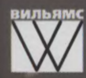
Компиляторы

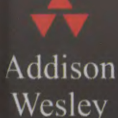
принципы, технологии и инструментарий

Второе издание

Сложность разработки компилятора



Альфред В. Ахо
Моника С. Лам
Рави Сети
Джеффри Д. Ульман



Компиляторы

принципы, технологии и инструментарий

Второе издание

Compilers

Principles, Techniques, & Tools

Second Edition

Alfred V. Aho
Columbia University

Monica S. Lam
Stanford University

Ravi Sethi
Avaya

Jeffrey D. Ullman
Stanford University

Boston • San Francisco • New York
London • Toronto • Sydney • Tokyo • Singapore • Madrid
Mexico City • Munich • Paris • Cape Town • Hong Kong • Montreal

Компиляторы

принципы, технологии и инструментарий

Второе издание

Альфред В. Ахо
Колумбийский университет

Моника С. Лам
Стэнфордский университет

Рави Сети
Avaya

Джеффри Д. Ульман
Стэнфордский университет

Москва • Санкт-Петербург • Киев
2008

ББК 32.973.26-018.2.75
К63
УДК 681.3.07

Издательский дом "Вильямс"

Зав. редакцией *С.Н. Тригуб*

Перевод с английского и редакция канд. техн. наук *И.В. Красикова*

По общим вопросам обращайтесь в Издательский дом "Вильямс" по адресу:
info@williamspublishing.com, http://www.williamspublishing.com
115419, Москва, а/я 783; 03150, Киев, а/я 152

Ахо, Альфред В., **Лам**, Моника С., **Сети**, Рави, **Ульман**, Джеффри Д.

К63 Компиляторы: принципы, технологии и инструментарий, 2-е изд. : Пер. с англ. — М. : ООО "И.Д. Вильямс", 2008. — 1184 с. : ил. — Парал. тит. англ.

ISBN 978–5–8459–1349–4 (рус.)

Эта книга, как и предыдущее издание, начинается с изложения основных принципов разработки компиляторов, включая детальное рассмотрение лексического и синтаксического анализа и генерации кода. Особенностью данного издания является широкое освещение вопросов оптимизации кода, в том числе для работы в многопроцессорных системах.

Строгость изложения материала смягчается большим количеством практических примеров. Написание компиляторов охватывает такие области знаний, как языки программирования, архитектура вычислительных систем, теория языков, алгоритмы и технология создания программного обеспечения. Помочь в освоении этих технологий и соответствующего инструментария и призвана данная книга. Несмотря на ее учебную ориентацию — в первую очередь, она предназначена для студентов и преподавателей соответствующих специальностей — книга будет полезна всем, кто работает над созданием компиляторов или просто интересуется данной темой.

ББК 32.973.26–018.2.75

Все названия программных продуктов являются зарегистрированными торговыми марками соответствующих фирм.

Никакая часть настоящего издания ни в каких целях не может быть воспроизведена в какой бы то ни было форме и какими бы то ни было средствами, будь то электронные или механические, включая фотокопирование и запись на магнитный носитель, если на это нет письменного разрешения издательства Addison-Wesley Publishing Company, Inc.

Authorized translation from the English language edition published by Addison-Wesley Publishing Company, Inc, Copyright © 2007

All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior written permission of the publisher.

Russian language edition published by Williams Publishing House according to the Agreement with R&I Enterprises International, Copyright © 2008

ISBN 978–5–8459–1349–4 (рус.)
ISBN 0–321–48681–1 (англ.)

© Издательский дом "Вильямс", 2008
© Pearson Education, Inc., 2007

Оглавление

Предисловие 24

Глава 1. Введение в компиляцию 29

Глава 2. Простой синтаксически управляемый транслятор 75

Глава 3. Лексический анализ 155

Глава 4. Синтаксический анализ 251

Глава 5. Синтаксически управляемая трансляция 383

Глава 6. Генерация промежуточного кода 443

Глава 7. Среды времени выполнения 525

Глава 8. Генерация кода 617

Глава 9. Машинно-независимые оптимизации 705

Глава 10. Параллелизм на уровне команд 841

Глава 11. Оптимизация параллелизма и локальности 911

Глава 12. Межпроцедурный анализ 1061

Приложение А. Завершенный пример начальной стадии компилятора 1133

Приложение Б. Поиск линейно независимых решений 1163

Предметный указатель 1167

Содержание

Предисловие 24

Глава 1. Введение в компиляцию 29

1.1 Компиляторы 29

1.1.1 Упражнения к разделу 1.1 32

1.2 Структура компилятора 32

1.2.1 Лексический анализ 33

1.2.2 Синтаксический анализ 35

1.2.3 Семантический анализ 37

1.2.4 Генерация промежуточного кода 38

1.2.5 Оптимизация кода 39

1.2.6 Генерация кода 39

1.2.7 Управление таблицей символов 40

1.2.8 Объединение фаз в проходы 41

1.2.9 Инструментарий для создания компиляторов 41

1.3 Эволюция языков программирования 42

1.3.1 Переход к языкам высокого уровня 42

1.3.2 Влияние на компиляторы 44

1.3.3 Упражнения к разделу 1.3 45

1.4 "Компилятороведение" 45

1.4.1 Моделирование при проектировании и реализации компилятора 45

1.4.2 Изучение оптимизации кода 46

1.5 Применения технологий компиляторов 48

1.5.1 Реализация высокоуровневых языков программирования 48

1.5.2 Оптимизация для архитектуры компьютера 50

1.5.3 Разработка новых архитектур компьютеров 52

1.5.4 Трансляции программ 54

1.5.5 Инструментарий для повышения производительности программного обеспечения 55

1.6 Азы языков программирования 57
1.6.1 Понятия статического и динамического 58
1.6.2 Среды и состояния 58
1.6.3 Статическая область видимости и блочная структура 61
1.6.4 Явное управление доступом 65
1.6.5 Динамическая область видимости 65
1.6.6 Механизмы передачи параметров 68
1.6.7 Псевдонимы 70
1.6.8 Упражнения к разделу 1.6 70
1.7 Резюме к главе 1 72
1.8 Список литературы к главе 1 73

Глава 2. Простой синтаксически управляемый транслятор 75
2.1 Введение 76
2.2 Определение синтаксиса 78
2.2.1 Определения грамматик 79
2.2.2 Выведение 81
2.2.3 Деревья разбора 82
2.2.4 Неоднозначности 84
2.2.5 Ассоциативность операторов 85
2.2.6 Приоритет операторов 86
2.2.7 Упражнения к разделу 2.2 89
2.3 Синтаксически управляемая трансляция 90
2.3.1 Постфиксная запись 91
2.3.2 Синтезированные атрибуты 92
2.3.3 Простые синтаксически управляемые определения 94
2.3.4 Обходы дерева 95
2.3.5 Схемы трансляции 97
2.3.6 Упражнения к разделу 2.3 99
2.4 Разбор 100
2.4.1 Нисходящий анализ 101
2.4.2 Предиктивный анализ 104
2.4.3 Использование пустых продукций 106
2.4.4 Разработка предиктивного анализатора 106
2.4.5 Левая рекурсия 107
2.4.6 Упражнения к разделу 2.4 109
2.5 Транслятор простых выражений 109
2.5.1 Абстрактный и конкретный синтаксис 110
2.5.2 Адаптация схемы трансляции 111
2.5.3 Процедуры для нетерминалов 113
2.5.4 Упрощение транслятора 114

2.5.5 Завершенная программа 115
2.6 Лексический анализ 118
2.6.1 Удаление пробельных символов и комментариев 119
2.6.2 Опережающее чтение 120
2.6.3 Константы 121
2.6.4 Распознавание ключевых слов и идентификаторов 121
2.6.5 Лексический анализатор 123
2.6.6 Упражнения к разделу 2.6 128
2.7 Таблицы символов 128
2.7.1 Таблица символов для области видимости 129
2.7.2 Использование таблиц символов 133
2.8 Генерация промежуточного кода 136
2.8.1 Два вида промежуточных представлений 136
2.8.2 Построение синтаксических деревьев 137
2.8.3 Статические проверки 142
2.8.4 Трехадресный код 144
2.8.5 Упражнения к разделу 2.8 151
2.9 Резюме к главе 2 152

Глава 3. Лексический анализ 155
3.1 Роль лексического анализатора 155
3.1.1 Лексический и синтаксический анализ 157
3.1.2 Токены, шаблоны и лексемы 157
3.1.3 Атрибуты токенов 159
3.1.4 Лексические ошибки 160
3.1.5 Упражнения к разделу 3.1 161
3.2 Буферизация ввода 162
3.2.1 Пары буферов 162
3.2.2 Ограничители 163
3.3 Спецификация токенов 165
3.3.1 Строки и языки 165
3.3.2 Операции над языками 166
3.3.3 Регулярные выражения 168
3.3.4 Регулярные определения 170
3.3.5 Расширения регулярных выражений 172
3.3.6 Упражнения к разделу 3.3 173
3.4 Распознавание токенов 177
3.4.1 Диаграммы переходов 179
3.4.2 Распознавание зарезервированных слов и идентификаторов 181
3.4.3 Завершение примера 183

3.4.4 Архитектура лексического анализатора на основе диаграммы переходов 184
3.4.5 Упражнения к разделу 3.4 187
3.5 Генератор лексических анализаторов Lex 191
3.5.1 Использование Lex 191
3.5.2 Структура программ Lex 192
3.5.3 Разрешение конфликтов в Lex 196
3.5.4 Прогностический оператор 197
3.5.5 Упражнения к разделу 3.5 198
3.6 Конечные автоматы 199
3.6.1 Недетерминированные конечные автоматы 200
3.6.2 Таблицы переходов 201
3.6.3 Принятие входной строки автоматом 201
3.6.4 Детерминированный конечный автомат 203
3.6.5 Упражнения к разделу 3.6 204
3.7 От регулярных выражений к автоматам 205
3.7.1 Преобразование НКА в ДКА 206
3.7.2 Моделирование НКА 210
3.7.3 Эффективность моделирования НКА 210
3.7.4 Построение НКА из регулярного выражения 213
3.7.5 Эффективность алгоритма обработки строк 218
3.7.6 Упражнения к разделу 3.7 221
3.8 Разработка генератора лексических анализаторов 221
3.8.1 Структура генерируемого анализатора 221
3.8.2 Распознавание шаблонов на основе НКА 223
3.8.3 ДКА для лексических анализаторов 225
3.8.4 Реализация прогностического оператора 226
3.8.5 Упражнения к разделу 3.8 228
3.9 Оптимизация распознавателей на основе ДКА 229
3.9.1 Важные состояния НКА 229
3.9.2 Функции, вычисляемые на синтаксическом дереве 231
3.9.3 Вычисление nullable, firstpos и lastpos 232
3.9.4 Вычисление followpos 234
3.9.5 Преобразование регулярного выражения непосредственно в ДКА 235
3.9.6 Минимизация количества состояний ДКА 237
3.9.7 Минимизация состояний в лексических анализаторах 242
3.9.8 Компромисс между скоростью и используемой памятью при моделировании ДКА 242
3.9.9 Упражнения к разделу 3.9 244

3.10 Резюме к главе 3 245
3.11 Список литературы к главе 3 247

Глава 4. Синтаксический анализ 251
4.1 Введение 252
4.1.1 Роль синтаксического анализатора 252
4.1.2 Образцы грамматик 253
4.1.3 Обработка синтаксических ошибок 254
4.1.4 Стратегии восстановления после ошибок 256
4.2 Контекстно-свободные грамматики 258
4.2.1 Формальное определение контекстно-свободной грамматики 258
4.2.2 Соглашения об обозначениях 260
4.2.3 Порождения 261
4.2.4 Деревья разбора и порождения 263
4.2.5 Неоднозначность 265
4.2.6 Проверка языка, сгенерированного грамматикой 266
4.2.7 Контекстно-свободные грамматики и регулярные выражения 267
4.2.8 Упражнения к разделу 4.2 269
4.3 Разработка грамматики 272
4.3.1 Лексический и синтаксический анализ 273
4.3.2 Устранение неоднозначности 273
4.3.3 Устранение левой рекурсии 275
4.3.4 Левая факторизация 278
4.3.5 Не контекстно-свободные языковые конструкции 279
4.3.6 Упражнения к разделу 4.3 280
4.4 Нисходящий синтаксический анализ 281
4.4.1 Синтаксический анализ методом рекурсивного спуска 283
4.4.2 FIRST и FOLLOW 285
4.4.3 LL(1)-грамматики 288
4.4.4 Нерекурсивный предиктивный синтаксический анализ 292
4.4.5 Восстановление после ошибок в предиктивном синтаксическом анализе 295
4.4.6 Упражнения к разделу 4.4 298
4.5 Восходящий синтаксический анализ 301
4.5.1 Свертки 302
4.5.2 Обрезка основ 302
4.5.3 Синтаксический анализ “перенос/свертка” 304
4.5.4 Конфликты в процессе ПС-анализа 306
4.5.5 Упражнения к разделу 4.5 309

4.6 Введение в LR-анализ: простой LR 309
4.6.1 Обоснование использования LR-анализаторов 310
4.6.2 Пункты и LR(0)-автомат 311
4.6.3 Алгоритм LR-анализа 317
4.6.4 Построение таблиц SLR-анализа 322
4.6.5 Активные префиксы 326
4.6.6 Упражнения к разделу 4.6 328
4.7 Более мощные LR-анализаторы 330
4.7.1 Канонические LR(1)-пункты 331
4.7.2 Построение множеств LR(1)-пунктов 332
4.7.3 Канонические таблицы LR(1)-анализа 336
4.7.4 Построение LALR-таблиц синтаксического анализа 338
4.7.5 Эффективное построение таблиц LALR-анализа 344
4.7.6 Уплотнение таблиц LR-анализа 349
4.7.7 Упражнения к разделу 4.7 352
4.8 Использование неоднозначных грамматик 353
4.8.1 Использование приоритетов и ассоциативности для разрешения конфликтов 353
4.8.2 Неоднозначность "висящего else" 357
4.8.3 Восстановление после ошибок в LR-анализе 358
4.8.4 Упражнения к разделу 4.8 361
4.9 Генераторы синтаксических анализаторов 363
4.9.1 Генератор синтаксических анализаторов `Yacc` 364
4.9.2 Использование `Yacc` с неоднозначной грамматикой 368
4.9.3 Создание лексического анализатора в `Yacc` с помощью `Lex` 371
4.9.4 Восстановление после ошибок в `Yacc` 372
4.9.5 Упражнения к разделу 4.9 375
4.10 Резюме к главе 4 375
4.11 Список литературы к главе 4 378

Глава 5. Синтаксически управляемая трансляция 383
5.1 Синтаксически управляемые определения 384
5.1.1 Наследуемые и синтезируемые атрибуты 384
5.1.2 Вычисление СУО в узлах дерева разбора 387
5.1.3 Упражнения к разделу 5.1 390
5.2 Порядок вычисления в СУО 391
5.2.1 Графы зависимостей 391
5.2.2 Упорядочение вычисления атрибутов 393
5.2.3 S-атрибутные определения 394
5.2.4 L-атрибутные определения 394

5.2.5 Семантические правила с контролируемыми побочными действиями 396
5.2.6 Упражнения к разделу 5.2 398
5.3 Применения синтаксически управляемой трансляции 399
5.3.1 Построение синтаксических деревьев 400
5.3.2 Структура типа 404
5.3.3 Упражнения к разделу 5.3 406
5.4 Синтаксически управляемые схемы трансляции 406
5.4.1 Постфиксные схемы трансляции 407
5.4.2 Реализация постфиксной СУТ с использованием стека синтаксического анализатора 407
5.4.3 СУТ с действиями внутри продукций 410
5.4.4 Устранение левой рекурсии из СУТ 411
5.4.5 СУТ для L-атрибутных определений 414
5.4.6 Упражнения к разделу 5.4 421
5.5 Реализация L-атрибутных СУО 422
5.5.1 Трансляция в процессе синтаксического анализа методом рекурсивного спуска 423
5.5.2 Генерация кода “на лету” 425
5.5.3 L-атрибутные СУО и LL-синтаксический анализ 428
5.5.4 Восходящий синтаксический анализ L-атрибутных СУО 434
5.5.5 Упражнения к разделу 5.5 439
5.6 Резюме к главе 5 439
5.7 Список литературы к главе 5 441

Глава 6. Генерация промежуточного кода 443
6.1 Варианты синтаксических деревьев 445
6.1.1 Ориентированные ациклические графы для выражений 445
6.1.2 Метод номера значения для построения ориентированных ациклических графов 447
6.1.3 Упражнения к разделу 6.1 450
6.2 Трехадресный код 450
6.2.1 Адреса и команды 450
6.2.2 Четверки 454
6.2.3 Тройки 455
6.2.4 Представление в виде статических единственных присваиваний 457
6.2.5 Упражнения к разделу 6.2 458
6.3 Типы и объявления 459
6.3.1 Выражения типов 459
6.3.2 Эквивалентность типов 461

6.3.3 Объявления 462
6.3.4 Размещение локальных имен в памяти 462
6.3.5 Последовательности объявлений 464
6.3.6 Поля в записях и классах 466
6.3.7 Упражнения к разделу 6.3 467
6.4 Трансляция выражений 468
6.4.1 Операции в выражениях 468
6.4.2 Инкрементная трансляция 470
6.4.3 Адресация элементов массива 471
6.4.4 Трансляция обращений к массиву 473
6.4.5 Упражнения к разделу 6.4 475
6.5 Проверка типов 477
6.5.1 Правила проверки типов 477
6.5.2 Преобразования типов 478
6.5.3 Перегрузка функций и операторов 481
6.5.4 Выведение типа и полиморфные функции 482
6.5.5 Алгоритм унификации 487
6.5.6 Упражнения к разделу 6.5 490
6.6 Поток управления 491
6.6.1 Булевы выражения 492
6.6.2 Код сокращенного вычисления 492
6.6.3 Инструкции потока управления 493
6.6.4 Трансляция логических выражений с помощью потока управления 496
6.6.5 Устранение излишних команд перехода 499
6.6.6 Булевы значения и код с переходами 501
6.6.7 Упражнения к разделу 6.6 502
6.7 Обратные поправки 504
6.7.1 Однопроходная генерация кода с использованием обратных поправок 504
6.7.2 Обратные поправки для булевых выражений 505
6.7.3 Инструкции потока управления 508
6.7.4 Инструкции break, continue и goto 511
6.7.5 Упражнения к разделу 6.7 512
6.8 Инструкции выбора 514
6.8.1 Трансляция инструкций выбора 514
6.8.2 Синтаксически управляемая трансляция инструкций выбора 515
6.8.3 Упражнения к разделу 6.8 517
6.9 Промежуточный код процедур 517

6.10 Резюме к главе 6 520
6.11 Список литературы к главе 6 521

Глава 7. Среды времени выполнения 525
7.1 Организация памяти 525
7.1.1 Статическое и динамическое распределение памяти 527
7.2 Выделение памяти в стеке 528
7.2.1 Деревья активации 529
7.2.2 Записи активации 532
7.2.3 Последовательности вызовов 535
7.2.4 Данные переменной длины в стеке 539
7.2.5 Упражнения к разделу 7.2 540
7.3 Доступ к нелокальным данным в стеке 542
7.3.1 Доступ к данным при отсутствии вложенных процедур 542
7.3.2 Вложенные процедуры 543
7.3.3 Язык с вложенными объявлениями процедур 544
7.3.4 Глубина вложенности 545
7.3.5 Связи доступа 547
7.3.6 Работа со связями доступа 548
7.3.7 Связи доступа для процедур, являющихся параметрами 550
7.3.8 Дисплеи 551
7.3.9 Упражнения к разделу 7.3 554
7.4 Управление кучей 555
7.4.1 Диспетчер памяти 555
7.4.2 Иерархия памяти компьютера 557
7.4.3 Локальность в программах 559
7.4.4 Снижение фрагментации 561
7.4.5 Освобождение памяти вручную 565
7.4.6 Упражнения к разделу 7.4 568
7.5 Введение в сборку мусора 568
7.5.1 Цели проектирования сборщиков мусора 569
7.5.2 Достижимость 572
7.5.3 Сборщики мусора с подсчетом ссылок 574
7.5.4 Упражнения к разделу 7.5 576
7.6 Введение в сборку на основе отслеживания 576
7.6.1 Базовый сборщик мусора 577
7.6.2 Базовая абстракция 580
7.6.3 Оптимизация алгоритма “пометить и подмести” 582
7.6.4 Сборщики мусора “пометить и сжать” 583
7.6.5 Копирующие сборщики 587
7.6.6 Сравнение стоимости 589

7.6.7 Упражнения к разделу 7.6 590

7.7 Распределенная сборка мусора 591

7.7.1 Инкрементная сборка мусора 591

7.7.2 Инкрементный анализ достижимости 593

7.7.3 Основы частичной сборки 596

7.7.4 Сборка мусора по поколениям 597

7.7.5 Алгоритм поезда 599

7.7.6 Упражнения к разделу 7.7 603

7.8 Дополнительные вопросы сборки мусора 604

7.8.1 Параллельная сборка мусора 605

7.8.2 Частичное перемещение объектов 608

7.8.3 Консервативная сборка мусора для небезопасных языков программирования 608

7.8.4 Слабые ссылки 609

7.8.5 Упражнения к разделу 7.8 610

7.9 Резюме к главе 7 611

7.10 Список литературы к главе 7 614

Глава 8. Генерация кода 617

8.1 Вопросы проектирования генератора кода 619

8.1.1 Вход генератора кода 619

8.1.2 Целевая программа 620

8.1.3 Выбор команд 621

8.1.4 Распределение регистров 623

8.1.5 Порядок вычислений 625

8.2 Целевой язык 625

8.2.1 Простая модель целевой машины 625

8.2.2 Стоимость программ и команд 629

8.2.3 Упражнения к разделу 8.2 630

8.3 Адреса в целевом коде 632

8.3.1 Статическое выделение памяти 632

8.3.2 Выделение памяти в стеке 635

8.3.3 Адреса имен времени выполнения 638

8.3.4 Упражнения к разделу 8.3 639

8.4 Базовые блоки и графы потоков 640

8.4.1 Базовые блоки 641

8.4.2 Информация о дальнейшем использовании 643

8.4.3 Графы потоков 644

8.4.4 Представление графов потоков 646

8.4.5 Циклы 646

8.4.6 Упражнения к разделу 8.4 647

8.5 Оптимизация базовых блоков 648
8.5.1 Представление базовых блоков с использованием ориентированных ациклических графов 648
8.5.2 Поиск локальных общих подвыражений 649
8.5.3 Устранение неиспользуемого кода 651
8.5.4 Применение алгебраических тождеств 652
8.5.5 Представление обращений к массивам 653
8.5.6 Присваивание указателей и вызовы процедур 655
8.5.7 Сборка базового блока из ориентированного ациклического графа 656
8.5.8 Упражнения к разделу 8.5 658
8.6 Простой генератор кода 660
8.6.1 Дескрипторы регистров и адресов 661
8.6.2 Алгоритм генерации кода 661
8.6.3 Разработка функции *getReg* 665
8.6.4 Упражнения к разделу 8.6 667
8.7 Локальная оптимизация 668
8.7.1 Устранение излишних загрузок и сохранений 668
8.7.2 Устранение недостижимого кода 669
8.7.3 Оптимизация потока управления 670
8.7.4 Алгебраические упрощения и снижение стоимости 671
8.7.5 Использование машинных идиом 671
8.7.6 Упражнения к разделу 8.7 671
8.8 Распределение и назначение регистров 672
8.8.1 Глобальное распределение регистров 672
8.8.2 Счетчики использований 673
8.8.3 Назначение регистров для внешних циклов 676
8.8.4 Распределение регистров путем раскраски графа 676
8.8.5 Упражнения к разделу 8.8 677
8.9 Выбор команд путем переписывания дерева 677
8.9.1 Схемы трансляции деревьев 678
8.9.2 Генерация кода путем замощения входного дерева 681
8.9.3 Поиск соответствий с использованием синтаксического анализа 684
8.9.4 Программы семантической проверки 685
8.9.5 Обобщенный поиск соответствий 686
8.9.6 Упражнения к разделу 8.9 688
8.10 Генерация оптимального кода для выражений 688
8.10.1 Числа Ершова 689
8.10.2 Генерация кода на основе помеченных деревьев выражений 690

8.10.3 Вычисление выражений при недостаточном количестве регистров 692
8.10.4 Упражнения к разделу 8.10 694
8.11 Генерация кода с использованием динамического программирования 695
8.11.1 Последовательные вычисления 696
8.11.2 Алгоритм динамического программирования 697
8.11.3 Упражнения к разделу 8.11 700
8.12 Резюме к главе 8 700
8.13 Список литературы к главе 8 702

Глава 9. Машинно-независимые оптимизации 705
9.1 Основные источники оптимизации 706
9.1.1 Причины избыточности 706
9.1.2 Конкретный пример: быстрая сортировка 707
9.1.3 Трансформации, сохраняющие семантику 710
9.1.4 Глобальные общие подвыражения 710
9.1.5 Распространение копий 712
9.1.6 Удаление бесполезного кода 713
9.1.7 Перемещение кода 714
9.1.8 Переменные индукции и снижение стоимости 715
9.1.9 Упражнения к разделу 9.1 718
9.2 Введение в анализ потоков данных 719
9.2.1 Абстракция потока данных 720
9.2.2 Схема анализа потока данных 722
9.2.3 Схемы потоков данных в базовых блоках 723
9.2.4 Достигающие определения 725
9.2.5 Анализ активных переменных 733
9.2.6 Доступные выражения 735
9.2.7 Резюме 740
9.2.8 Упражнения к разделу 9.2 740
9.3 Основы анализа потока данных 743
9.3.1 Полурешетки 744
9.3.2 Передаточные функции 750
9.3.3 Итеративный алгоритм в обобщенной структуре 753
9.3.4 Смысл решения потока данных 755
9.3.5 Упражнения к разделу 9.3 759
9.4 Распространение констант 760
9.4.1 Значения потока данных для структуры распространения констант 761
9.4.2 Сбор в структуре распространения констант 762

9.4.3 Передаточные функции для структуры распространения констант 762
9.4.4 Монотонность структуры распространения констант 763
9.4.5 Недистрибутивность структуры распространения констант 764
9.4.6 Интерпретация результатов 765
9.4.7 Упражнения к разделу 9.4 767
9.5 Устранение частичной избыточности 768
9.5.1 Источники избыточности 769
9.5.2 Все ли избыточные вычисления могут быть устранены? 771
9.5.3 Отложенное перемещение кода 773
9.5.4 Ожидаемость выражений 774
9.5.5 Алгоритм отложенного перемещения кода 775
9.5.6 Упражнения к разделу 9.5 785
9.6 Циклы в графах потоков 787
9.6.1 Доминаторы 787
9.6.2 Упорядочение в глубину 790
9.6.3 Ребра в глубинном остовном дереве 792
9.6.4 Обратные ребра и приводимость 794
9.6.5 Глубина графа потока 795
9.6.6 Естественные циклы 796
9.6.7 Скорость сходимости итеративных алгоритмов потоков данных 798
9.6.8 Упражнения к разделу 9.6 801
9.7 Анализ на основе областей 803
9.7.1 Области 804
9.7.2 Иерархии областей для приводимых графов потоков 805
9.7.3 Обзор анализа на основании областей 808
9.7.4 Необходимые предположения о передаточных функциях 810
9.7.5 Алгоритм анализа на основе областей 811
9.7.6 Обработка неприводимых графов потоков 816
9.7.7 Упражнения к разделу 9.7 818
9.8 Символический анализ 819
9.8.1 Аффинные выражения ссылочных переменных 819
9.8.2 Формулировка задачи потока данных 822
9.8.3 Символический анализ на основе областей 827
9.8.4 Упражнения к разделу 9.8 832
9.9 Резюме к главе 9 833
9.10 Список литературы к главе 9 838

Глава 10. Параллелизм на уровне команд 841
10.1 Архитектуры процессоров 842
10.1.1 Конвейерная обработка команд и задержки ветвления 842
10.1.2 Конвейерное выполнение 843
10.1.3 Многоадресные команды 844
10.2 Ограничения планирования кода 845
10.2.1 Зависимость через данные 846
10.2.2 Поиск зависимостей среди обращений к памяти 846
10.2.3 Компромиссы между использованием регистров и параллелизмом 848
10.2.4 Упорядочение фаз распределения регистров и планирования кода 851
10.2.5 Зависимость от управления 852
10.2.6 Поддержка опережающего выполнения 853
10.2.7 Базовая модель машины 855
10.2.8 Упражнения к разделу 10.2 856
10.3 Планирование базовых блоков 857
10.3.1 Графы зависимости данных 858
10.3.2 Планирование списков базовых блоков 859
10.3.3 Приоритетные топологические порядки 861
10.3.4 Упражнения к разделу 10.3 863
10.4 Глобальное планирование кода 864
10.4.1 Примитивное перемещение кода 865
10.4.2 Восходящее перемещение кода 867
10.4.3 Нисходящее перемещение кода 868
10.4.4 Обновление зависимостей данных 870
10.4.5 Алгоритм глобального планирования 870
10.4.6 Усовершенствованные методы перемещения кода 874
10.4.7 Взаимодействие с динамическими планировщиками 875
10.4.8 Упражнения к разделу 10.4 876
10.5 Программная конвейеризация 876
10.5.1 Введение 877
10.5.2 Программная конвейеризация циклов 879
10.5.3 Распределение регистров и генерация кода 882
10.5.4 Циклы с зависимыми итерациями 883
10.5.5 Цели и ограничения программной конвейеризации 884
10.5.6 Алгоритм программной конвейеризации 888
10.5.7 Планирование ациклических графов зависимости данных 889
10.5.8 Планирование графов с циклическими зависимостями 891
10.5.9 Усовершенствования алгоритма конвейеризации 899

10.5.10 Модульное расширение переменных 900
10.5.11 Условные инструкции 903
10.5.12 Аппаратная поддержка программной конвейеризации 904
10.5.13 Упражнения к разделу 10.5 904
10.6 Резюме к главе 10 907
10.7 Список литературы к главе 10 909

Глава 11. Оптимизация параллелизма и локальности 911
11.1 Фундаментальные концепции 914
11.1.1 Многопроцессорность 914
11.1.2 Параллелизм в приложениях 917
11.1.3 Параллелизм на уровне циклов 918
11.1.4 Локальность данных 920
11.1.5 Введение в теорию аффинных преобразований 922
11.2 Пример посерьезнее: умножение матриц 927
11.2.1 Алгоритм умножения матриц 927
11.2.2 Оптимизации 930
11.2.3 Интерференция кэша 933
11.2.4 Упражнения к разделу 11.2 933
11.3 Пространства итераций 934
11.3.1 Построение пространств итераций вложений циклов 934
11.3.2 Порядок выполнения вложенности циклов 936
11.3.3 Матричная запись неравенств 938
11.3.4 Добавление символьных констант 938
11.3.5 Управление порядком выполнения 939
11.3.6 Изменение осей 944
11.3.7 Упражнения к разделу 11.3 945
11.4 Аффинные индексы массивов 948
11.4.1 Аффинные обращения к данным 948
11.4.2 Аффинное и неаффинное обращения на практике 949
11.4.3 Упражнения к разделу 11.4 950
11.5 Повторное использование данных 951
11.5.1 Типы повторных использований 952
11.5.2 Собственные повторные использования 953
11.5.3 Собственное пространственное повторное использование 958
11.5.4 Групповое повторное использование 959
11.5.5 Упражнения к разделу 11.5 962
11.6 Анализ зависимости данных в массивах 964
11.6.1 Определение зависимостей данных доступов к массивам 965
11.6.2 Целочисленное линейное программирование 966

11.6.3 НОД 967
11.6.4 Эвристики для решения задачи целочисленного линейного программирования 969
11.6.5 Решение обобщенной задачи целочисленного линейного программирования 973
11.6.6 Резюме 975
11.6.7 Упражнения к разделу 11.6 976
11.7 Поиск параллельности, не требующей синхронизации 978
11.7.1 Вводный пример 978
11.7.2 Разбиения аффинного пространства 981
11.7.3 Ограничения разбиений пространства 982
11.7.4 Решение ограничений разбиений пространств 986
11.7.5 Простой алгоритм генерации кода 990
11.7.6 Устранение пустых итераций 993
11.7.7 Устранение проверок из внутреннего цикла 996
11.7.8 Преобразования исходного кода 998
11.7.9 Упражнения к разделу 11.7 1003
11.8 Синхронизация между параллельными циклами 1005
11.8.1 Постоянное количество синхронизаций 1006
11.8.2 Графы зависимостей программ 1007
11.8.3 Иерархическое время 1009
11.8.4 Алгоритм распараллеливания 1012
11.8.5 Упражнения к разделу 11.8 1013
11.9 Конвейеризация 1013
11.9.1 Что такое конвейеризация 1014
11.9.2 Последовательная сверхрелаксация: пример 1016
11.9.3 Полностью переставляемые циклы 1017
11.9.4 Конвейеризация полностью переставляемых циклов 1018
11.9.5 Общая теория 1021
11.9.6 Ограничения временного разбиения 1022
11.9.7 Решение временны́х ограничений с использованием леммы Фаркаша 1026
11.9.8 Преобразования кода 1029
11.9.9 Параллелизм с минимальной синхронизацией 1035
11.9.10 Упражнения к разделу 11.9 1037
11.10 Оптимизации локальности 1039
11.10.1 Временна́я локальность вычисляемых данных 1040
11.10.2 Сжатие массива 1041
11.10.3 Чередование частей 1044
11.10.4 Алгоритмы оптимизации локальности 1047
11.10.5 Упражнения к разделу 11.10 1050

11.11 Прочие применения аффинных преобразований 1050
11.11.1 Машины с распределенной памятью 1050
11.11.2 Процессоры с одновременным выполнением нескольких команд 1051
11.11.3 Векторные и SIMD-команды 1052
11.11.4 Предвыборка 1053
11.12 Резюме к главе 11 1054
11.13 Список литературы к главе 11 1057

Глава 12. Межпроцедурный анализ 1061
12.1 Базовые концепции 1062
12.1.1 Графы вызовов 1062
12.1.2 Чувствительность к контексту 1064
12.1.3 Строки вызовов 1067
12.1.4 Контекстно-чувствительный анализ на основе клонирования 1069
12.1.5 Контекстно-чувствительный анализ на основе резюме 1070
12.1.6 Упражнения к разделу 12.1 1073
12.2 Необходимость межпроцедурного анализа 1075
12.2.1 Вызовы виртуальных методов 1076
12.2.2 Анализ псевдонимов указателей 1076
12.2.3 Параллельность 1077
12.2.4 Поиск программных ошибок и уязвимых мест 1077
12.2.5 SQL-ввод 1078
12.2.6 Переполнение буфера 1080
12.3 Логическое представление потока данных 1081
12.3.1 Введение в Datalog 1082
12.3.2 Правила Datalog 1083
12.3.3 Интенсиональные и экстенсиональные предикаты 1085
12.3.4 Выполнение программы Datalog 1088
12.3.5 Инкрементное вычисление программ Datalog 1090
12.3.6 Проблематичные правила Datalog 1091
12.3.7 Упражнения к разделу 12.3 1093
12.4 Простой алгоритм анализа указателей 1095
12.4.1 Сложность анализа указателей 1096
12.4.2 Модель указателей и ссылок 1097
12.4.3 Нечувствительность к потоку 1098
12.4.4 Формулировка с применением Datalog 1099
12.4.5 Использование информации о типе 1101
12.4.6 Упражнения к разделу 12.4 1102
12.5 Контекстно-нечувствительный межпроцедурный анализ 1105

12.5.1 Влияние вызовов методов 1105
12.5.2 Построение графа вызовов в Datalog 1107
12.5.3 Динамическая загрузка и отражение 1108
12.5.4 Упражнения к разделу 12.5 1109
12.6 Контекстно-чувствительный анализ указателей 1109
12.6.1 Контексты и строки вызовов 1110
12.6.2 Добавление контекста в правила Datalog 1113
12.6.3 Дополнительные наблюдения о чувствительности 1114
12.6.4 Упражнения к разделу 12.6 1115
12.7 Реализация Datalog с применением BDD 1115
12.7.1 Диаграммы бинарного выбора 1116
12.7.2 Преобразования диаграмм бинарного выбора 1118
12.7.3 Представление отношений при помощи BDD 1119
12.7.4 Операции отношений и BDD-операции 1120
12.7.5 Использование диаграмм бинарного выбора для анализа целей указателей 1123
12.7.6 Упражнения к разделу 12.7 1124
12.8 Резюме к главе 12 1124
12.9 Список литературы к главе 12 1128

Приложение А. Завершенный пример начальной стадии компилятора 1133
A.1 Исходный язык 1133
A.2 Main 1135
A.3 Лексический анализатор 1135
A.4 Таблицы символов и типы 1139
A.5 Промежуточный код выражений 1140
A.6 Переходы для булевых выражений 1144
A.7 Промежуточный код для инструкций 1149
A.8 Синтаксический анализатор 1153
A.9 Построение начальной стадии 1160

Приложение Б. Поиск линейно независимых решений 1163

Предметный указатель 1167

Предисловие

Со времени написания первого издания книги в 1986 году в мире разработки компиляторов произошли значительные изменения. Языки программирования эволюционировали и перед разработчиками возникли новые проблемы. Архитектура компьютеров предлагает массу различных вычислительных ресурсов, которые для получения наивысших результатов должен использовать разработчик компилятора. Пожалуй, наиболее интересно то, что методы оптимизации кода нашли применение не только в компиляторах. В настоящее время они используются в инструментарии для поиска ошибок в программном обеспечении и, что особенно важно, при проверке безопасности существующего кода. Большинство других методов — грамматика, регулярные выражения, синтаксические анализаторы и синтаксически управляемые трансляторы — также широко используются не только при разработке компиляторов.

Таким образом, наш подход остался тем же, что и в предыдущих версиях книги. Мы отдаем себе отчет в том, что лишь малая доля наших читателей будет заниматься разработкой или хотя бы поддержкой компилятора для какого-то из основных языков программирования. Однако модели, теория и алгоритмы, связанные с компиляторами, могут применяться для решения широкого диапазона задач проектирования и разработки программного обеспечения. Таким образом, мы уделяем особое внимание задачам, которые обычно встречаются при разработке языковых процессоров, независимо от исходного языка или целевой машины.

Использование данной книги

Требуется примерно два семестра, чтобы охватить весь (или, по крайней мере, бóльшую часть) материал данной книги. Обычно первая половина книги служит предметом изучения студентов на первых курсах, в то время как вторая, посвященная оптимизации кода, изучается на старших курсах.

Вот краткое содержание глав книги.

В главе 1 содержится, в основном, мотивационный материал, и при этом раскрываются некоторые базовые вопросы архитектуры компьютера и принципов языков программирования.

В главе 2 разрабатывается миниатюрный компилятор и вводится множество новых концепций, которые будут затем детально рассмотрены в последующих главах. Сам компилятор вы найдете в приложении А.

Глава 3 посвящена лексическому анализу, регулярным выражениям, конечным автоматам и инструментарию для создания сканеров входного потока. Материал этой главы может быть широко использован при разработке текстовых редакторов.

В главе 4 приведены различные методы синтаксического анализа, нисходящие (рекурсивного спуска, LL) и восходящие (LR и его варианты).

В главе 5 вы познакомитесь с принципиальными идеями синтаксически управляемых определений и трансляции.

В главе 6 показано, как теория из главы 5 применяется для генерации промежуточного кода в типичном языке программирования.

В главе 7 обсуждаются вопросы сред времени выполнения, в частности управление стеком и сборка мусора.

Глава 8 посвящена генерации объектного кода. В ней рассматриваются построение базовых блоков, генерация кода из выражений и базовых блоков и методы распределения регистров.

Глава 9 служит введением в методы оптимизации кода, включая потоковые графы, структуры потоков данных и итеративные алгоритмы для их разрешения.

В главе 10 рассматривается оптимизация на уровне команд. Основной упор делается на поиск возможностей параллельного вычисления в небольших последовательностях команд и их диспетчеризация для процессоров, способных выполнять несколько команд одновременно.

Глава 11 посвящена поиску крупномасштабной параллельности вычислений и ее использованию. Здесь упор делается на числовой код, который содержит связанные циклы, работающие с многомерными массивами.

В главе 12 речь идет о межпроцедурном анализе: анализе указателей, использовании псевдонимов и анализе потоков данных, которые принимают во внимание последовательность вызовов процедур, достигающих данной точки кода.

Материал данной книги изучался в нескольких университетах — Колумбии, Гарварде и Стенфорде. В Колумбии в учебных курсах для первокурсников по языкам программирования и трансляторам регулярно использовался материал глав 1–8. Основным заданием данного курса являлось написание курсовой работы, выполняющейся небольшими группами студентов и состоящей в реализации небольшого языка программирования, спроектированного ими самостоятельно. Разработанные таким образом языки ориентированы на применение в различных областях, включая квантовые вычисления, генерацию музыки, компьютерную графику, игры, операции с матрицами и многое другое. В своей работе студенты использовали генераторы компонентов компиляторов, такие как ANTLR, Lex и Yacc, и методы синтаксически управляемой трансляции, рассматривающиеся в главах 2 и 5. Курс для старшекурсников основывался на материале глав 9–12, и основ-

ной упор делался на генерацию кода и оптимизацию для современных машин, включая сетевую и многопроцессорную архитектуры.

В Стенфорде краткий вводный курс охватывал материал приблизительно глав 1–8, хотя в нем было и введение в вопросы глобальной оптимизации, основанное на материале главы 9. Второй курс, посвященный компиляторам, охватывал главы 9–12, а также материал по сборке мусора из главы 7. Студенты использовали местную систему Joeq на основе Java для реализации алгоритмов анализа потоков данных.

Необходимые требования

Читатель должен владеть определенными знаниями в области информатики, включая, по меньшей мере, курс по программированию и курсы по структурам данных и дискретной математике. Полезно также знать несколько различных языков программирования.

Упражнения

Практически в каждом разделе книги содержится большое количество упражнений. Более сложные упражнения отмечены одним восклицательным знаком, а самые сложные — двумя.

Поддержка в Интернете

Начальная страница книги находится по адресу

dragonbook.stanford.edu

Здесь вы можете найти список обнаруженных ошибок и исправлений (http://www-db.stanford.edu/~ullman/dragon/errata.html) и дополнительные материалы к книге. Мы надеемся сделать общедоступной информацию по всем курсам, посвященным компиляторам, которые мы ведем, включая домашние задания, решения задач и экзаменационные вопросы. Мы также планируем сделать доступной информацию о важных компиляторах, предоставленную их разработчиками.

Благодарности

Благодарим дизайнера обложки — С.Д. Ульман (S.D. Ullman) из *Strange Tonic Productions*.

Джон Бентли (Jon Bentley) основательно помог своими комментариями ряда глав книги, когда она находилась еще на стадии черновика.

Полезные комментарии и исправления ошибок были сделаны следующими людьми.

Доменико Бьянкулли (Domenico Bianculli), Питер Бош (Peter Bosch), Марцио Басс (Marcio Buss), Марк Идди (Marc Eaddy), Стефен Эдвардс (Stephen Edwards), Вибхав Гарг (Vibhav Garg), Ким Хазельвуд (Kim Hazelwood), Гурав Кц (Gaurav Kc), Вей Ли (Wei Li), Майк Смит (Mike Smith), Арт Стамнесс (Art Stamness), Криста Свор (Krysta Svore), Оливер Тардью (Olivier Tardieu) и Джия Зенг (Jia Zeng).

Мы благодарим всех их за помощь. Все оставшиеся в книге ошибки остаются на нашей совести.

Кроме того, Моника благодарит своих коллег по работе над компилятором SUIF за 18-летнее обучение. Это Джеральд Айнер (Gerald Aigner), Дзинтарс Авотс (Dzintars Avots), Саман Амарасингх (Saman Amarasinghe), Дженнифер Андерсон (Jennifer Anderson), Майкл Карбин (Michael Carbin), Джеральд Чинг (Gerald Cheong), Амер Диван (Amer Diwan), Роберт Френч (Robert French), Анвар Гулом (Anwar Ghuloum), Мери Холл (Mary Hall), Джон Хеннеси (John Hennessy), Девид Гейне (David Heine), Ших-Вей Лао (Shih-Wei Liao), Ами Лим (Amy Lim), Бенджамин Лившиц (Benjamin Livshits), Майкл Мартин (Michael Martin), Дрор Майдан (Dror Maydan), Тодд Моури (Todd Mowry), Брайан Мерфи (Brian Murphy), Джеффри Оплингер (Jeffrey Oplinger), Карен Пипер (Karen Pieper), Мартин Ринард (Martin Rinard), Олатунджи Рувасе (Olatunji Ruwase), Константине Сапунцакис (Constantine Sapuntzakis), Патрик Сатьянатан (Patrick Sathyanathan), Майкл Смит (Michael Smith), Стивен Тжианг (Steven Tjiang), Чау-Вен Ценг (Chau-Wen Tseng), Кристофер Анкель (Christopher Unkel), Джон Воли (John Whaley), Роберт Вильсон (Robert Wilson), Кристофер Вильсон (Christopher Wilson) и Майкл Вольф (Michael Wolf).

A. V. A., Чатмен, Нью-Джерси

M. S. L., Менло-Парк, Калифорния

R. S., Фар-Хиллс, Нью-Джерси

J. D. U., Стенфорд, Калифорния

Июнь, 2006

От издательства

Вы, читатель этой книги, и есть главный ее критик и комментатор. Мы ценим ваше мнение и хотим знать, что было сделано нами правильно, что можно было сделать лучше и что еще вы хотели бы увидеть изданным нами. Нам интересно услышать и любые другие замечания, которые вам хотелось бы высказать в наш адрес.

Мы ждем ваших комментариев и надеемся на них. Вы можете прислать нам бумажное или электронное письмо, либо просто посетить наш Web-сервер и оставить свои замечания там. Одним словом, любым удобным для вас способом дайте нам знать, нравится или нет вам эта книга, а также выскажите свое мнение о том, как сделать наши книги более интересными для вас.

Посылая письмо или сообщение, не забудьте указать название книги и ее авторов, а также ваш обратный адрес. Мы внимательно ознакомимся с вашим мнением и обязательно учтем его при отборе и подготовке к изданию последующих книг. Наши координаты:

E-mail: `info@williamspublishing.com`

WWW: `http://www.williamspublishing.com`

Информация для писем

из России: 115419, Москва, а/я 783

из Украины: 03150, Киев, а/я 152

Глава 1

Введение в компиляцию

Языки программирования представляют собой средство описания вычислений для людей и машин. Современный мир зависит от языков программирования, поскольку все программное обеспечение на всех компьютерах мира написано на том или ином языке программирования. Однако, прежде чем запустить программу, ее необходимо преобразовать в форму, которая может выполняться на компьютере.

Программные системы, выполняющие такое преобразование, называются *компиляторами*.

Эта книга о том, как разрабатывать и реализовывать компиляторы. Вы узнаете, что для построения трансляторов для широкого диапазона языков и машин могут использоваться всего несколько базовых идей. Принципы и методы проектирования компиляторов применимы в таком количестве многих других областей, что только редкий ученый-кибернетик не столкнется с ними множество раз в процессе своей деятельности. Изучение написания компиляторов требует обращения к таким темам, как языки программирования, архитектура компьютера, теория языков, алгоритмы, разработка программного обеспечения.

В этой вступительной главе вы познакомитесь с различными видами трансляторов, структурой типичного компилятора и узнаете о тенденциях развития языков программирования и архитектуры вычислительных систем и их влиянии на компиляторы. Здесь также рассматриваются вопросы взаимодействия проектирования компиляторов и теоретической информатики и применения методов разработки компиляторов, выходящих за рамки компиляции. Завершится этот краткий обзор рассмотрением ключевых концепций языков программирования, которые будут необходимы при изучении компиляторов.

1.1 Компиляторы

Попросту говоря, компилятор — это программа, которая считывает текст программы, написанной на одном языке — *исходном*, и транслирует (переводит) его в эквивалентный текст на другом языке — *целевом* (рис. 1.1). Одна из важных ролей компилятора состоит в сообщении об ошибках в исходной программе, обнаруженных в процессе трансляции.

Если целевая программа представляет собой программу на машинном языке, она затем может быть вызвана пользователем для обработки некоторых входных данных и получения некоторых выходных данных (рис. 1.2).

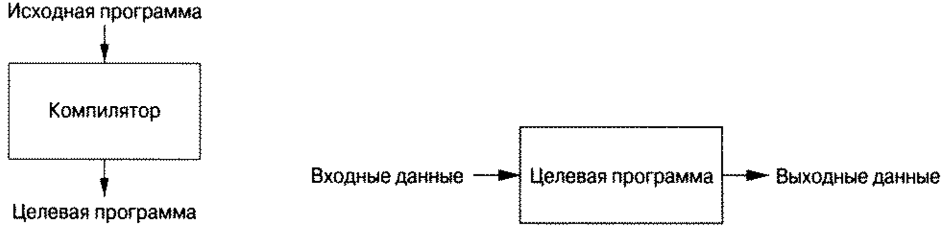


Рис. 1.1. Компилятор

Рис. 1.2. Выполнение целевой программы

Интерпретатор представляет собой еще один распространенный вид языкового процессора. Вместо получения целевой программы, как в случае транслятора, интерпретатор непосредственно выполняет операции, указанные в исходной программе, над входными данными, предоставляемыми пользователем (рис. 1.3).

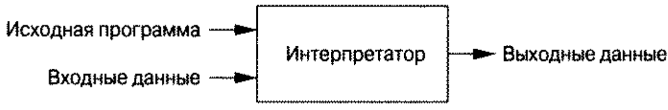


Рис. 1.3. Интерпретатор

Целевая программа на машинном языке, производимая компилятором, обычно гораздо быстрее, чем интерпретатор, получает выходные данные на основании входных. Однако интерпретатор обычно обладает лучшими способностями к диагностике ошибок, чем компилятор, поскольку он выполняет исходную программу инструкция за инструкцией.[1]

Пример 1.1. Языковой процессор Java объединяет в себе и компиляцию, и интерпретацию (рис. 1.4). Исходная программа на Java может сначала компилироваться в промежуточный вид, именуемый байт-кодом (bytecode). Затем байт-код интерпретируется виртуальной машиной. Преимущество такого решения в том, что скомпилированный на одной машине байт-код может быть выполнен на другой, например, будучи передан по сети.

[1]Обращаем ваше внимание на перевод термина *statement* в данной книге. Дословно он означает *высказывание*, *утверждение*, однако в применении к компьютерной тематике это не совсем удачно. Обычно при переводе используется термин *оператор*, но в данной книге, посвященной формальным языкам, этот термин имеет собственное значение. Поэтому при переводе термина *statement* за редкими исключениями было использовано понятие *инструкция* — как наиболее близкое по смыслу, так и использовавшееся в переводе предыдущего издания книги (А. Ахо, Р. Сети, Д. Ульман. *Компиляторы: принципы, технологии и инструменты*. — М.: Издательский дом "Вильямс", 2001). — *Прим. пер.*

Для более быстрой обработки входных данных некоторые компиляторы Java, именуемые *just-in-time*-компиляторами, транслируют байт-код в машинный язык непосредственно перед запуском промежуточной программы для обработки входных данных. □

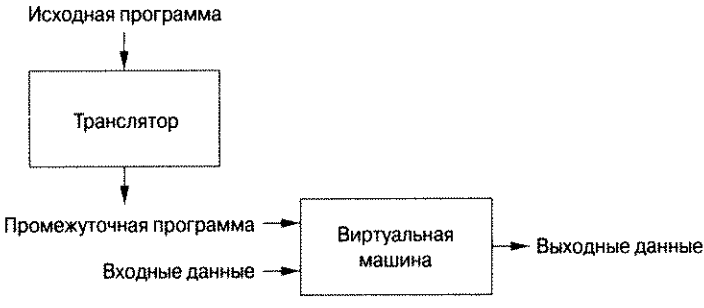


Рис. 1.4. Гибридный компилятор

Кроме компиляторов, потребовать создания выполнимой целевой программы могут и другие программы (рис. 1.5). Исходная программа может быть разделена на модули, находящиеся в различных файлах. Сборка исходной программы иногда поручается отдельной программе, именуемой *препроцессором*. Препроцес-

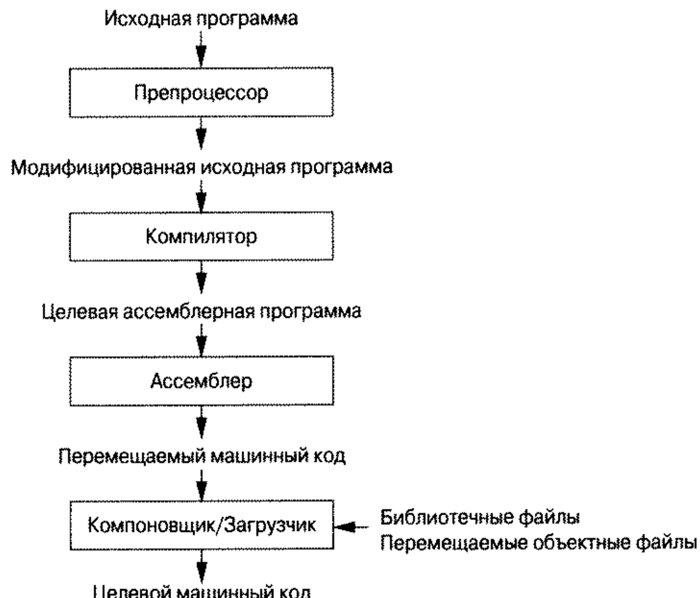


Рис. 1.5. Система обработки языка

сор может также раскрывать сокращения, именуемые макросами, в инструкции исходного языка.

Модифицированная исходная программа затем передается компилятору. Компилятор может выдать в качестве выходных данных программу на языке ассемблера, поскольку ассемблерный код легче создать и проще отлаживать. Язык ассемблера затем обрабатывается программой, которая называется *ассемблер*, и дает в качестве выходных данных перемещаемый машинный код.

Большие программы зачастую компилируются по частям, так что перемещаемый машинный код должен быть скомпонован совместно с другими перемещаемыми объектными файлами и библиотечными файлами в код, который можно будет выполнять на данной машине. *Компоновщик* ("линкер") выполняет разрешение внешних адресов памяти, по которым код из одного файла может обращаться к информации из другого файла. *Загрузчик* затем помещает все выполнимые объектные файлы в память для выполнения.

1.1.1 Упражнения к разделу 1.1

Упражнение 1.1.1. В чем заключается разница между компилятором и интерпретатором?

Упражнение 1.1.2. Каковы преимущества (а) компилятора перед интерпретатором и (б) интерпретатора перед компилятором?

Упражнение 1.1.3. Каковы преимущества системы обработки языка, в которой компилятор дает выход на языке ассемблера, по сравнению с системой, в которой компилятор дает выход на машинном языке?

Упражнение 1.1.4. Компилятор, который транслирует программу на высокоуровневом языке программирования в программу на другом высокоуровневом языке программирования, называется транслятором из исходного текста в исходный текст (source-to-source). Каковы преимущества использования языка программирования C в качестве целевого для такого компилятора?

Упражнение 1.1.5. Опишите некоторые из задач, которые должен выполнять ассемблер.

1.2 Структура компилятора

Пока что мы рассматривали компилятор как "черный ящик", отображающий исходную программу в семантически эквивалентную ей целевую программу. Если мы немного приоткроем этот ящик, то увидим, что это отображение разделяется на две части: анализ и синтез.

Анализ разбивает исходную программу на составные части и накладывает на них грамматическую структуру. Затем он использует эту структуру для создания

промежуточного представления исходной программы. Если анализ обнаруживает, что исходная программа неверно составлена синтаксически либо дефектна семантически, он должен выдать информативные сообщения об этом, чтобы пользователь мог исправить обнаруженные ошибки. Анализ также собирает информацию об исходной программе и сохраняет ее в структуре данных, именуемой *таблицей символов*, которая передается вместе с промежуточным представлением синтезу.

Синтез строит требуемую целевую программу на основе промежуточного представления и информации из таблицы символов. Анализ часто называют начальной стадией (front end), а синтез — заключительной (back end).

Если рассмотреть процесс компиляции более детально, можно увидеть, что он представляет собой последовательность *фаз*, каждая из которых преобразует одно из представлений исходной программы в другое. Типичное разложение компилятора на фазы приведено на рис. 1.6. На практике некоторые фазы могут объединяться, а межфазное промежуточное представление может не строиться явно. Таблица символов, в которой хранится информация обо всей исходной программе, используется всеми фазами компилятора.

Некоторые компиляторы содержат фазу машинно-независимой оптимизации между анализом и синтезом. Назначение этой оптимизации — преобразовать промежуточное представление, чтобы синтез мог получить более качественную целевую программу по сравнению с той, которая может быть получена из неоптимизированного промежуточного представления. Поскольку оптимизация необязательна, некоторые фазы оптимизации, показанные на рис. 1.6, в компиляторе могут отсутствовать.

1.2.1 Лексический анализ

Первая фаза компиляции называется *лексическим анализом* или *сканированием*. Лексический анализатор читает поток символов, составляющих исходную программу, и группирует эти символы в значащие последовательности, называющиеся *лексемами*. Для каждой лексемы анализатор строит выходной *токен* (token) вида

$$\langle \textit{имя_токена}, \textit{значение_атрибута} \rangle$$

Он передается последующей фазе, синтаксическому анализу. Первый компонент токена, *имя_токена*, представляет собой абстрактный символ, использующийся во время синтаксического анализа, а второй компонент, *значение_атрибута*, указывает на запись в таблице символов, соответствующую данному токену. Информация из записи в таблице символов необходима для семантического анализа и генерации кода.

Предположим, например, что исходная программа содержит инструкцию присваивания

```
position = initial + rate * 60        (1.1)
```

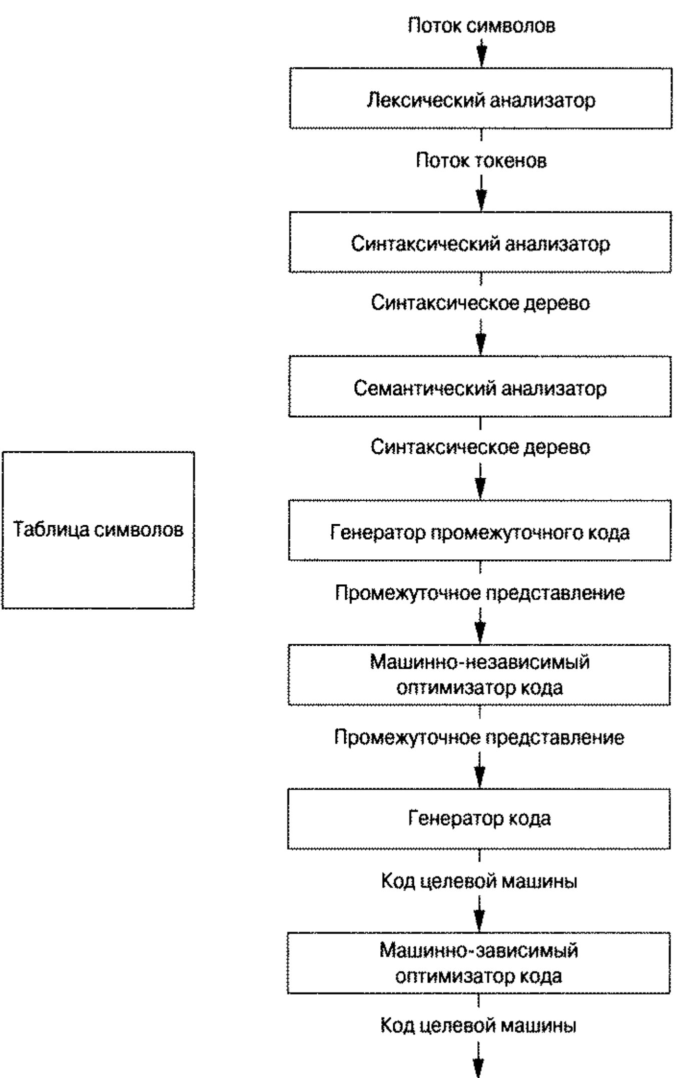


Рис. 1.6. Фазы компилятора

Символы в этом присваивании могут быть сгруппированы в следующие лексемы и отображены в следующие токены, передаваемые синтаксическому анализатору.

1. `position` представляет собой лексему, которая может отображаться в токен $\langle \mathbf{id}, 1\rangle$, где **id** — абстрактный символ, обозначающий *идентификатор*, а 1 указывает запись в таблице символов для `position`. Запись таблицы символов для некоторого идентификатора хранит информацию о нем, такую как его имя и тип.

2. Символ присваивания = представляет собой лексему, которая отображается в токен $\langle = \rangle$. Поскольку этот токен не требует значения атрибута, второй компонент данного токена опущен. В качестве имени токена может быть использован любой абстрактный символ, например такой, как **assign**, но для удобства записи мы будем использовать в качестве имени абстрактного символа саму лексему.

3. `initial` представляет собой лексему, которая отображается в токен $\langle \mathbf{id}, 2 \rangle$, где 2 указывает на запись в таблице символов для `initial`.

4. + является лексемой, отображаемой в токен $\langle + \rangle$.

5. `rate` — лексема, отображаемая в токен $\langle \mathbf{id}, 3 \rangle$, где 3 указывает на запись в таблице символов для `rate`.

6. `*` — лексема, отображаемая в токен $\langle * \rangle$.

7. `60` — лексема, отображаемая в токен $\langle 60 \rangle$.[2]

Пробелы, разделяющие лексемы, лексическим анализатором отбрасываются.

На рис. 1.7 показано представление инструкции присваивания (1.1) после лексического анализа в виде последовательности токенов

$$\mathbf{id}1\ \langle = \rangle\ \mathbf{id}2\ \langle + \rangle\ \mathbf{id}3\ \langle * \rangle\ \langle 60 \rangle \tag{1.2}$$

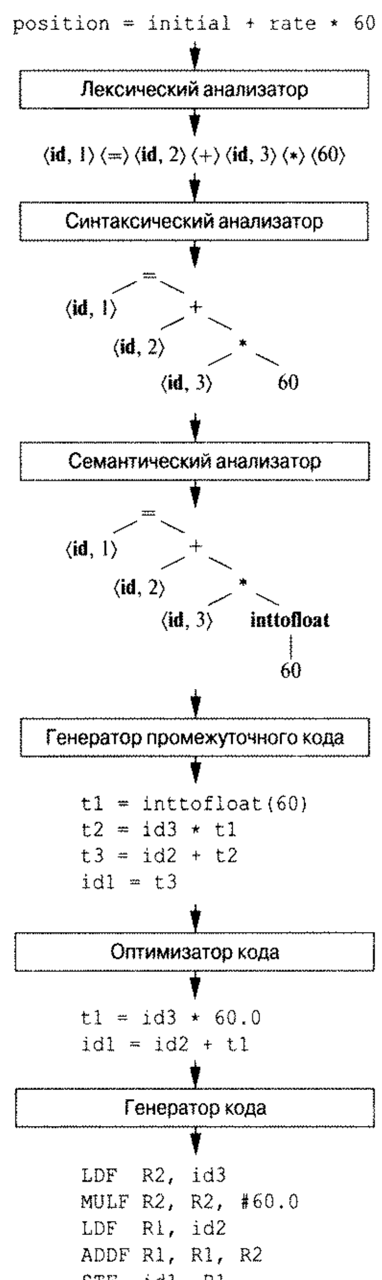
При этом представлении имена токенов =, + и * представляют собой абстрактные символы для операторов присваивания, сложения и умножения соответственно.

1.2.2 Синтаксический анализ

Вторая фаза компилятора — *синтаксический анализ* или *разбор* (parsing). Анализатор использует первые компоненты токенов, полученных при лексическом анализе, для создания древовидного промежуточного представления, которое описывает грамматическую структуру потока токенов. Типичным представлением является *синтаксическое дерево*, в котором каждый внутренний узел представляет операцию, а дочерние узлы — аргументы этой операции. Синтаксическое дерево для потока токенов (1.2) показано на выходе синтаксического анализатора на рис. 1.7.

---

[2]Технически говоря, для лексемы `60` мы должны создать токен наподобие $\langle \mathbf{number}, 4 \rangle$, где 4 указывает на запись таблицы символов для внутреннего представления целого числа 60, но обсуждение токенов для чисел мы отложим до главы 2. В главе 3 будут рассмотрены методы построения лексических анализаторов.



1	position	...
2	initial	...
3	rate	...

Таблица символов

Рис. 1.7. Трансляция инструкции присваивания

Это дерево указывает порядок, в котором выполняются операции в присваивании

```
position = initial + rate * 60
```

Дерево имеет внутренний узел, помеченный *, левым дочерним узлом которого является $\langle \mathbf{id}, 3\rangle$, а правым — 60. Узел $\langle \mathbf{id}, 3\rangle$ представляет идентификатор `rate`. Узел, помеченный *, явно указывает, что сначала мы должны умножить значение `rate` на 60. Узел, помеченный +, указывает, что мы должны прибавить результат умножения к значению `initial`. Корень дерева с меткой = говорит о том, что следует присвоить результат этого сложения в позиции памяти, отведенной идентификатору `position`. Порядок операций согласуется с обычными арифметическими правилами, которые говорят о том, что умножение имеет более высокий приоритет, чем сложение, и должно быть выполнено до сложения.

Последующие фазы компилятора используют грамматическую структуру, которая помогает проанализировать исходную и сгенерировать целевую программу. В главе 4 мы будем использовать контекстно-свободные грамматики для определения грамматической структуры языков программирования и рассмотрим алгоритмы автоматического построения эффективных синтаксических анализаторов для определенных классов грамматик. Из глав 2 и 5 вы узнаете, что синтаксически управляемые определения могут помочь уточнить трансляцию конструкций языка программирования.

1.2.3 Семантический анализ

Семантический анализатор использует синтаксическое дерево и информацию из таблицы символов для проверки исходной программы на семантическую согласованность с определением языка. Он также собирает информацию о типах и сохраняет ее в синтаксическом дереве или в таблице символов для последующего использования в процессе генерации промежуточного кода.

Важной частью семантического анализа является *проверка типов*, когда компилятор проверяет, имеет ли каждый оператор операнды соответствующего типа. Например, многие определения языков программирования требуют, чтобы индекс массива был целым числом; компилятор должен сообщить об ошибке, если в качестве индекса массива используется число с плавающей точкой.

Спецификация языка может разрешать определенные преобразования типов, именуемые *приведениями* (coercion). Например, бинарный арифметический оператор может быть применен либо к паре целых чисел, либо к паре чисел с плавающей точкой. Если такой оператор применен к числу с плавающей точкой и целому числу, то компилятор может выполнить преобразование целого числа в число с плавающей точкой.

Такое приведение показано на рис. 1.7. Предположим, что `position`, `initial` и `rate` были объявлены как числа с плавающей точкой и что лексема 60 образует целое число. Проверка типов в семантическом анализаторе на рис. 1.7 определяет, что оператор * применяется к числу с плавающей точкой `rate` и целому числу 60. В этом случае целое число может быть преобразовано в число с плавающей точкой. Обратите внимание, что на рис. 1.7 в синтаксическом дереве, полученном на выходе семантического анализатора, имеется дополнительный узел для оператора **inttofloat**, который явным образом преобразует свой целый аргумент в число с плавающей точкой. Проверка типов и семантический анализ рассматриваются в главе 6.

1.2.4 Генерация промежуточного кода

В процессе трансляции исходной программы в целевой код компилятор может создавать одно или несколько промежуточных представлений различного вида. Синтаксические деревья являются видом промежуточного представления; обычно они используются в процессе синтаксического и семантического анализа.

После синтаксического и семантического анализа исходной программы многие компиляторы генерируют явное низкоуровневое или машинное промежуточное представление исходной программы, которое можно рассматривать как программу для абстрактной вычислительной машины. Такое промежуточное представление должно обладать двумя важными свойствами: оно должно легко генерироваться и легко транслироваться в целевой машинный язык.

В главе 6 мы рассмотрим промежуточное представление, которое называется *трехадресным кодом* и состоит из последовательности команд, напоминающих ассемблерные, причем в каждой команде имеется три операнда и каждый операнд может действовать, как регистр. Выход генератора промежуточного кода на рис. 1.7 состоит из последовательности трехадресных кодов

```
t1 = inttofloat(60)
t2 = id3 * t1
t3 = id2 + t2
id1 = t3
```
(1.3)

Следует сделать несколько замечаний по поводу трехадресных команд. Во-первых, каждая трехадресная команда присваивания содержит как минимум один оператор справа. Таким образом, приведенные команды определяют порядок выполнения операций — в исходной программе (1.1) умножение выполняется раньше сложения. Во-вторых, компилятор должен генерировать временные имена для хранения значений, вычисляемых трехадресными командами. В-третьих, некоторые "трехадресные команды" наподобие первой и последней в последовательности (1.3) содержат менее трех операндов.

В главе 6 мы рассмотрим основные промежуточные представления, используемые компиляторами. В главе 5 вы познакомитесь с методами синтаксически управляемой трансляции, которые будут применены в главе 6 для проверки типов и генерации промежуточного кода для типичных конструкций языка программирования, таких как выражения, конструкции управления потоком выполнения и вызовы процедур.

1.2.5 Оптимизация кода

Фаза машинно-независимой оптимизации кода пытается улучшить промежуточный код, чтобы затем получить более качественный целевой код. Обычно "более качественный", "лучший" означает "более быстрый", но могут применяться и другие критерии сравнения, как, например, "более короткий код" или "код, использующий меньшее количество ресурсов". Например, непосредственный алгоритм генерирует промежуточный код (1.3), используя по команде для каждого оператора в синтаксическом дереве, полученном на выходе семантического анализатора.

Простой алгоритм генерации промежуточного кода с последующим оптимизатором кода представляет собой рациональный способ генерации хорошего целевого кода. Оптимизатор может определить, что преобразование 60 из целого числа в число с плавающей точкой может быть выполнено единственный раз во время компиляции, так что операция **inttofloat** может быть устранена путем замены целого числа 60 числом с плавающей точкой 60.0. Кроме того, `t3` используется только один раз — для передачи значения в `id1`, так что оптимизатор может преобразовать (1.3) в более короткую последовательность

```
t1 = id3 * 60.0
id1 = id2 + t1
```
(1.4)

Имеется большой разброс количества усилий, затрачиваемых различными компиляторами на оптимизацию на этом этапе. Так называемые "оптимизирующие компиляторы" затрачивают на эту фазу достаточно много времени, в то время как другие компиляторы применяют здесь только простые методы оптимизации, которые существенно повышают скорость работы целевой программы, при этом не слишком замедляя процесс компиляции. В главах, начиная с 8, детально рассматриваются методы машинно-зависимой и машинно-независимой оптимизации.

1.2.6 Генерация кода

Генератор кода получает в качестве входных данных промежуточное представление исходной программы и отображает его в целевой язык. Если целевой язык

представляет собой машинный код, для каждой переменной, используемой программой, выбираются соответствующие регистры или ячейки памяти. Затем промежуточные команды транслируются в последовательности машинных команд, выполняющих те же действия. Ключевым моментом генерации кода является аккуратное распределение регистров для хранения переменных.

Например, при использовании регистров R1 и R2 промежуточный код (1.4) может транслироваться в машинный код

```
LDF  R2,  id3
MULF R2,  R2, #60.0
LDF  R1,  id2                 (1.5)
ADDF R1,  R1, R2
STF  id1, R1
```

Первый операнд каждой команды определяет приемник. F в каждой команде говорит о том, что команда работает с числами с плавающей точкой. Код (1.5) загружает содержимое адреса id3 в регистр R2, затем умножает его на константу с плавающей точкой 60.0. # указывает, что 60.0 следует рассматривать как непосредственное значение. Третья команда помещает id2 в регистр R1, а четвертая прибавляет к нему предварительно вычисленное и сохраненное в регистре R2 значение. Наконец, значение регистра R1 сохраняется в адресе id1, так что код корректно реализует инструкцию присваивания (1.1). Подробнее генерация кода рассматривается в главе 8.

Это беглое знакомство с генерацией кода полностью игнорирует важный вопрос о распределении памяти для идентификаторов в исходной программе. Как вы увидите в главе 7, организация памяти во время выполнения программы зависит от компилируемого языка. Решения о распределении памяти принимаются либо в процессе генерации промежуточного кода, либо при генерации целевого кода.

1.2.7 Управление таблицей символов

Важная функция компилятора состоит в том, чтобы записывать имена переменных в исходной программе и накапливать информацию о разных атрибутах каждого имени. Эти атрибуты могут предоставлять информацию о выделенной памяти для данного имени, его типе, области видимости (где именно в программе может использоваться его значение) и, в случае имен процедур, такие сведения, как количество и типы их аргументов, метод передачи каждого аргумента (например, по значению или по ссылке), а также возвращаемый тип.

Таблица символов представляет собой структуру данных, содержащую записи для каждого имени переменной, с полями для атрибутов имени. Структура данных

должна быть разработана таким образом, чтобы позволять компилятору быстро находить запись для каждого имени, а также быстро сохранять данные в записи и получать их из нее. Таблицы символов рассматриваются в главе 2.

1.2.8 Объединение фаз в проходы

Фазы связаны с логической организацией компилятора. При реализации работа разных фаз может быть сгруппирована в *проходы* (pass), которые считывают входной файл и записывают выходной. Например, фазы анализа — лексический анализ, синтаксический анализ, семантический анализ и генерация промежуточного кода — могут быть объединены в один проход. Оптимизация кода может представлять собой необязательный проход. Затем может быть еще один проход, заключающийся в генерации кода для конкретной целевой машины.

Некоторые наборы компиляторов созданы вокруг тщательно разработанного промежуточного представления, которое позволяет начальной стадии для некоторого языка программирования взаимодействовать с заключительной стадией для определенной целевой машины. При наличии таких наборов можно создавать компиляторы для различных исходных языков и одной целевой машины, комбинируя различные начальные стадии с заключительной стадией для этой целевой машины. Аналогично можно разрабатывать компиляторы для различных целевых машин, комбинируя начальную стадию с заключительными стадиями для различных целевых машин.

1.2.9 Инструментарий для создания компиляторов

Разработчики компиляторов, как и разработчики любого другого программного обеспечения, могут с успехом использовать современные среды разработки программного обеспечения, содержащие такие инструменты, как редакторы языков, отладчики, средства контроля версий, профайлеры, средства тестирования и т.п. В дополнение к этим средствам общего назначения может использоваться ряд более специализированных инструментов, созданных для помощи в реализации различных фаз компилятора.

Эти инструменты используют собственные специализированные языки для описания и реализации отдельных компонентов, и многие из них основаны на весьма сложных алгоритмах. Наиболее успешными являются те инструменты, которые скрывают детали алгоритма генерации и создают компоненты, легко интегрируемые в компилятор. К широко используемым инструментам для создания компиляторов относятся следующие.

1. *Генераторы синтаксических анализаторов*, которые автоматически создают синтаксические анализаторы на основе грамматического описания языка программирования.

2. *Генераторы сканеров*, которые создают лексические анализаторы на основе описания токенов языка с использованием регулярных выражений.

3. *Средства синтаксически управляемой трансляции*, которые создают наборы подпрограмм для обхода синтаксического дерева и генерации промежуточного кода.

4. *Генераторы генераторов кода*, которые создают генераторы кода на основе набора правил трансляции каждой операции промежуточного языка в машинный язык для целевой машины.

5. *Средства работы с потоком данных*, которые облегчают сбор информации о передаче значений от одной части программы ко всем другим. Анализ потоков данных представляет собой ключевую часть оптимизации кода.

6. *Наборы для построения компиляторов*, которые представляют собой интегрированные множества подпрограмм для построения разных фаз компиляторов.

В этой книге будут описаны многие из перечисленных инструментов.

1.3 Эволюция языков программирования

Первые ЭВМ появились в 1940-х годах и программировались исключительно на машинных языках путем последовательностей нулей и единиц, которые явно указывали компьютеру, какие операции и в каком порядке должны быть выполнены. Сами операции были очень низкоуровневыми: переместить данные из одной ячейки памяти в другую, сложить содержимое двух регистров, сравнить два значения и т.д. Надо отметить, что такое программирование было очень медленным, утомительным и подверженным ошибкам. Однажды написанную программу было очень трудно понять и модифицировать.

1.3.1 Переход к языкам высокого уровня

Первым шагом в создании более дружественных языков программирования была разработка мнемонических ассемблерных языков в начале 1950-х годов.

Изначально команды ассемблера являлись всего лишь мнемоническими представлениями машинных команд. Позже в языки ассемблера были введены макросы, так что программист мог определять параметризованные сокращения для часто использующихся последовательностей машинных команд.

Большим шагом к высокоуровневым языкам программирования стала разработка во второй половине 1950-х годов языка программирования Fortran — для

научных вычислений, Cobol — для обработки бизнес-данных и Lisp — для символьных вычислений. Философия, стоящая за этими языками, заключается в создании высокоуровневой системы обозначений, облегчающей программисту написание программ для численных вычислений, бизнес-приложений и символьных программ. Эти языки были столь успешны, что применяются и сегодня.

В последующие десятилетия было создано множество языков программирования с новыми возможностями, которые делали написание программ более простым, естественным и надежным. Позже в этой главе мы рассмотрим некоторые ключевые возможности многих современных языков программирования.

В настоящее время существуют тысячи языков программирования. Их можно классифицировать различными способами. Один из способов классификации — по поколениям. *Языки первого поколения* — это машинные языки; *языки второго поколения* — языки ассемблера, а к *языкам третьего поколения* относятся высокоуровневые языки программирования, такие как Fortran, Cobol, Lisp, C, C++, C# и Java. *Языки четвертого поколения* — это языки программирования, разработанные для конкретных применений, например NOMAD — для генерации отчетов, SQL — для запросов к базам данных и Postscript — для форматирования текстов. Термин *языки пятого поколения* применяется к языкам программирования, основанным на логике или ограничениях, таким как Prolog и OPS5.

Еще в одной классификации языков программирования используются термин *императивный* (imperative) по отношению к языкам программирования, в которых программа указывает, *как* должны выполняться вычисления, и термин *декларативный* (declarative) по отношению к языкам, которые указывают, *какие* вычисления должны быть выполнены. Языки программирования наподобие C, C++, C# и Java являются императивными. В императивных языках имеется понятие состояния программы и инструкции, которые изменяют это состояние. Функциональные языки, такие как ML и Haskell, а также логические языки типа Prolog, часто рассматриваются как декларативные.

Термин *язык фон Неймана* (von Neumann language) применим к языкам программирования, вычислительная модель которых основана на вычислительной архитектуре фон Неймана. Многие современные языки программирования, такие как Fortran и C, являются языками фон Неймана.

Объектно-ориентированные языки — это языки программирования, поддерживающие объектно-ориентированное программирование, стиль программирования, в котором программа состоит из набора объектов, взаимодействующих друг с другом. Главными наиболее ранними объектно-ориентированными языками программирования стали Simula 67 и Smalltalk; примерами более поздних объектно-ориентированных языков программирования являются C++, C#, Java и Ruby.

Языки сценариев ("скриптов") — это интерпретируемые языки с высокоуровневыми операторами, разработанными для "склеивания" вычислений. Такие вычисления изначально назывались *сценариями* (script). Популярными примерами

языков сценариев являются Awk, JavaScript, Perl, PHP, Python, Ruby, REXX и Tcl. Программы, написанные на языках сценариев, чаще всего существенно короче эквивалентных программ, написанных на языках программирования наподобие С.

1.3.2 Влияние на компиляторы

Поскольку разработка языков программирования и разработка компиляторов тесно связаны между собой, новые достижения в области языков программирования приводят к новым требованиям, возникающим перед разработчиками компиляторов, которые должны придумывать алгоритмы и представления для трансляции и поддержки новых возможностей языка. Кроме того, с 1940-х годов произошли существенные изменения и в архитектуре вычислительных систем, так что разработчики компиляторов должны не только учитывать новые свойства языков программирования, но и разрабатывать такие алгоритмы трансляции, которые смогут максимально использовать преимущества новых аппаратных возможностей.

Компиляторы могут способствовать использованию высокоуровневых языков программирования, минимизируя накладные расходы времени выполнения программ, написанных на этих языках. Они играют важную роль в эффективном использовании высокопроизводительной архитектуры компьютера пользовательскими приложениями. В действительности производительность вычислительной системы настолько зависит от технологии компиляции, что компиляторы используются в качестве инструмента для оценки архитектурных концепций перед созданием компьютера.

Написание компилятора представляет настоящий вызов для программиста. Компилятор сам по себе — большая и сложная программа. Кроме того, многие современные системы обработки языков работают с разными языками и целевыми машинами в пределах одного пакета, т.е. представляют собой набор компиляторов, состоящих, возможно, из миллионов строк кода. Соответственно, при разработке и создании современных языковых процессоров определяющую роль играют методы программотехники.

Компилятор обязан корректно транслировать потенциально бесконечное множество программ, которые могут быть написаны на соответствующем языке программирования. Задача генерации оптимального целевого кода из исходной программы в общем случае неразрешима; таким образом, разработчики компиляторов должны искать компромиссные решения о том, какие эвристики следует использовать для генерации эффективного кода.

Изучение компиляторов — это также изучение вопроса о том, насколько теория соответствует практике, как вы увидите в разделе 1.4.

Цель данной книги — изучение методологии и фундаментальных идей, использующихся при разработке компиляторов. Мы не ставили задачу обучить читателя

всем имеющимся алгоритмам и методам, которые могут быть использованы для создания современной системы обработки языка. Однако прочитавшие эту книгу получат базовые знания в количестве, достаточном для понимания того, как относительно просто создать собственный компилятор.

1.3.3 Упражнения к разделу 1.3

Упражнение 1.3.1. Укажите, какие термины из списка

а) императивный	б) декларативный	в) фон Неймана
г) объектно-ориентированный	д) функциональный	е) третьего поколения
ж) четвертого поколения	з) сценариев	

применимы к языкам программирования из списка

1) C	2) C++	3) Cobol	4) Fortran	5) Java
6) Lisp	7) ML	8) Perl	9) Python	10) VB

1.4 "Компиляторореведение"

Разработка компиляторов полна красивых примеров решения сложных задач, возникающих при реальной работе над компиляторами, путем математического абстрагирования. Они служат прекрасной иллюстрацией того, как для решения задач могут быть применены абстракции: для этого надо для конкретной задачи сформулировать математическую абстракцию, которая содержит ключевые характеристики задачи, и решить ее с применением математических методов. Формулировка задачи должна основываться на четком понимании характеристик компьютерных программ, а решение должно быть проверено и уточнено эмпирически.

Компилятор должен принимать все исходные программы, которые соответствуют спецификации языка; множество исходных программ бесконечно, а сама программа может быть очень большой, состоящей, возможно, из миллионов строк. Любые преобразования, выполняемые компилятором при трансляции исходной программы, должны сохранить неизменным смысл компилируемой программы. Таким образом, разработчики компиляторов оказывают влияние не только на создаваемые ими компиляторы, но и на все программы, которые будут скомпилированы их компилятором. Это делает написание компиляторов особенно достойным занятием, но и налагает на программистов особую ответственность.

1.4.1 Моделирование при проектировании и реализации компилятора

Изучение компиляторов, в основном, заключается в изучении способов разработки математических моделей и выбора правильных алгоритмов, балансируя

между обобщенностью и мощью, с одной стороны, и простотой и эффективностью — с другой.

Некоторые из наиболее фундаментальных моделей — конечные автоматы и регулярные выражения, о которых пойдет речь в главе 3. Эти модели полезны для описания лексических единиц программы (ключевых слов, идентификаторов и т.п.) и алгоритмов, используемых компилятором для распознавания этих единиц. К фундаментальным моделям относятся и контекстно-свободные грамматики, используемые для описания синтаксических структур языков программирования, таких как вложенные скобки и управляющие конструкции. Грамматики будут изучаться в главе 4. Аналогично деревья являются важной моделью для представления структуры программы и ее трансляции в объектный код, как вы узнаете из главы 5.

1.4.2 Изучение оптимизации кода

Термин "оптимизация" в проектировании компиляторов означает попытки компилятора получить код, более эффективный, чем обычно. "Оптимизация", таким образом, является некорректным названием, поскольку нет способа гарантированно получить код столь же быстрый (или более быстрый), как любой другой код, выполняющий те же задачи.

В настоящее время оптимизация кода, выполняемая компилятором, становится все более важной и более сложной. Она становится более сложной, поскольку архитектуры процессоров также становятся все более сложными, предоставляя все больше возможностей для улучшения способа выполнения кода. Оптимизация становится более важной, поскольку массовое наступление компьютеров с возможностью параллельных вычислений требует существенной оптимизации, иначе их производительность падает на порядки. Вероятное преобладание многоядерных машин (компьютеров с большим количеством процессоров) требует от компиляторов использования преимуществ многопроцессорности.

Очень трудно, если вообще возможно, построить мощный компилятор без "хакерства". Соответственно, вокруг задачи оптимизации кода построена богатая и полезная теория. Используя точные математические основы, можно убедиться в корректности оптимизации и получить желаемый эффект для всех возможных входных данных. Начиная с главы 9, вы увидите, насколько необходимы такие математические модели, как графы, матрицы и линейное программирование, для построения компилятором хорошо оптимизированного кода.

С другой стороны, одной голой теории недостаточно. Как и у многих других задач реального мира, здесь нет окончательного идеального ответа. В действительности большинство вопросов, задаваемых при оптимизации кода компилятором, оказываются неразрешимыми. Одно из важных проявлений профессионализма в проектировании компиляторов состоит в умении корректно сформулировать ре-

шаемую задачу. Для начала требуются хорошее понимание поведения программ и всестороннее экспериментирование и вычисления для подтверждения интуитивных предположений.

Оптимизации, выполняемые компилятором, должны отвечать трем требованиям проектирования:

- оптимизация должна быть корректной, т.е. сохранять смысл компилируемой программы;
- оптимизация должна повышать производительность многих программ;
- время компиляции должно оставаться в разумных пределах;
- требуемая для реализации оптимизации инженерная работа должна быть осуществима.

Невозможно переоценить важность корректности. Написать компилятор, генерирующий быстрый код, — тривиальная задача, если генерируемый код может быть неверным! Оптимизирующие компиляторы настолько сложны, что мы готовы даже заявить, что нет ни одного полностью безошибочного оптимизирующего компилятора! Итак, наиболее важная цель при написании компилятора — это его корректность.

Вторая цель состоит в том, чтобы компилятор мог повышать производительность многих программ. Обычно производительность означает скорость выполнения программы. Если говорить о встраиваемых приложениях, то тут может оказаться желательным также малый размер сгенерированного кода. В случае мобильных устройств желательно, чтобы код минимизировал потребление энергии. Обычно та же оптимизация, которая повышает скорость выполнения, приводит и к экономии энергии. Помимо производительности, важны и такие потребительские аспекты, как сообщения об ошибках и отладка.

Время компиляции должно оставаться достаточно небольшим, чтобы поддерживать быстрый цикл разработки и отладки. Выполнить это требование становится все проще с ростом быстродействия машин. Зачастую сначала программа разрабатывается и отлаживается без оптимизации. Это делается не только для снижения времени компиляции, но и, что более важно, потому что неоптимизированную программу легче отлаживать (оптимизация компилятором зачастую ухудшает связь между исходным и объектным кодами). Включение оптимизации компилятора иногда выявляет новые проблемы в исходной программе; таким образом, тестирование должно выполняться для оптимизированного кода заново. Требование дополнительного тестирования иногда удерживает от использования оптимизации в приложениях, в особенности если их производительность не критична.

И наконец, компилятор представляет собой сложную систему, которая при этом должна быть достаточно простой, чтобы стоимость ее разработки и поддержки оставалась в разумных пределах. Имеется бесконечное число оптимизаций, которые могут быть реализованы, но для получения корректной и эффективной оптимизации часто требуются выходящие из ряда вон усилия. Приоритет в реализации отдается тем оптимизациям, которые приводят к большему выигрышу для встречающихся на практике исходных программ.

Таким образом, при изучении компиляторов следует не только научиться строить компилятор, но и освоить общую методологию решения сложных и не ограниченных определенными рамками задач. Подход, используемый при разработке компиляторов, включает как теорию, так и эксперимент. Обычно работа начинается с постановки задачи, основанной на интуитивном представлении о важности тех или иных вопросов.

1.5 Применения технологий компиляторов

Многие программисты используют методы, с которыми они познакомились при изучении компиляторов, не только для написания компиляторов или каких-то их частей. Эти методы находят широкое применение и в других областях. Проектирование компиляторов затрагивает ряд других областей информатики, и в этом разделе мы рассмотрим некоторые наиболее важные взаимодействия и приложения компиляторных технологий.

1.5.1 Реализация высокоуровневых языков программирования

Высокоуровневый язык программирования определяет программную абстракцию: программист выражает алгоритм с использованием языка, а компилятор должен транслировать эту программу в целевой язык. Вообще говоря, высокоуровневые языки программирования проще для программирования, но менее эффективны, т.е. целевые программы работают более медленно. Программисты, использующие низкоуровневые языки, обладают большими возможностями контроля над выполняемыми вычислениями и могут, в принципе, получать более эффективный код. К сожалению, на низкоуровневых языках труднее писать программы, и, что еще хуже, эти языки менее переносимы, программы на них более подвержены ошибкам и их труднее сопровождать. Оптимизирующие компиляторы включают технологии для повышения производительности генерируемого кода, тем самым компенсируя неэффективность высокоуровневых абстракций.

Пример 1.2. Ключевое слово **register** в языке программирования C представляет собой пример взаимодействия между методами компиляции и эволюцией языка.

Когда язык C создавался в средине 1970-х годов, считалось необходимым позволить программисту определять, какие переменные должны размещаться в регистрах. Однако после разработки эффективных методов распределения регистров такое управление стало излишним, и большинство современных программ не используют эту возможность языка.

Более того, программы, использующие ключевое слово **register**, могут проигрывать в эффективности, поскольку программисты — не лучшие знатоки в низкоуровневых вопросах наподобие распределения регистров. Выбор оптимального распределения регистров очень сильно зависит от архитектуры конкретной машины. Следование принятым программистом решениям о низкоуровневом распределении ресурсов может в действительности привести к падению производительности, в особенности при работе программы на других машинах, а не на той, для которой она создавалась. □

Многие распространенные языки программирования смещаются в сторону увеличения уровня абстракции. Доминирующим языком программирования в 1980-е годы был C, но уже в 1990-х годах во многих новых проектах использовался язык программирования C++; разработанный в 1995 году язык программирования Java быстро приобрел популярность к концу 1990-х годов. Новые возможности языков программирования приводили к новым исследованиям в области оптимизации кода. Далее будут рассмотрены основные возможности языков программирования, которые стимулировали значительное развитие технологий компиляции.

Практически все распространенные языки программирования, включая C, Fortran и Cobol, поддерживают определяемые пользователем агрегированные типы данных, такие как массивы и структуры, а также высокоуровневые средства управления потоком выполнения, такие как циклы и вызовы процедур. Если ограничиться непосредственной трансляцией каждой высокоуровневой конструкции или операции обращения к данным, результат получится очень неэффективным. Разработанная *оптимизация потоков данных* (data-flow optimization) анализирует потоки данных в программе и удаляет избыточность таких конструкций. Она генерирует код, который напоминает код, написанный на низком уровне профессиональным программистом.

Объектная ориентированность впервые появилась в языке Simula в 1967 году и вошла в такие языки, как Smalltalk, C++, C# и Java. Ключевыми идеями, лежащими в основе объектной ориентированности, являются

1) абстракция данных;

2) наследование свойств.

Они делают программы более модульными и упрощают их поддержку. Объектно-ориентированные программы отличаются от программ, написанных на других

языках программирования, тем, что они состоят из большего количества процедур меньшего размера (которые в объектно-ориентированном программировании называются *методами*). Таким образом, оптимизация должна быть способна перешагнуть границы процедур в исходной программе. Здесь оказалось особенно полезным встраивание процедур, представляющее собой замену вызова процедуры ее телом. Была также разработана оптимизация, ускоряющая диспетчеризацию виртуальных методов.

Java обладает многими облегчающими программирование возможностями, которые появились ранее в других языках программирования. Язык Java безопасен с точки зрения типов, т.е. объект не может использоваться вместо объекта несвязанного типа. Обращение к массивам выполняется с проверкой выхода за пределы массива. Java не имеет указателей и не использует соответствующую арифметику. Этот язык программирования имеет встроенную систему сборки мусора, которая автоматически освобождает выделенную для переменных память, которая больше не используется. Все эти возможности упрощают программирование, но приводят к дополнительным накладным расходам времени выполнения. Разработанная для языка программирования Java оптимизация снижает накладные расходы, например, устраняя излишние проверки диапазонов и выделяя память для объектов, недоступных извне процедуры, в стеке, а не в куче. Разработаны также эффективные алгоритмы для минимизации накладных расходов по сборке мусора.

Кроме прочего, язык программирования Java разработан для поддержки переносимости и мобильного кода. Программы распространяются в виде байт-кода Java, который либо интерпретируется, либо компилируется в машинный код динамически, т.е. в процессе выполнения. Динамическая компиляция изучалась также и в других контекстах, когда информация извлекается динамически, во время выполнения, и используется для получения более оптимизированного кода. При такой динамической оптимизации важно минимизировать время компиляции, которое является частью накладных расходов выполнения программы. Распространенная методика состоит в компиляции и оптимизации только тех частей программы, которые будут часто выполняться.

1.5.2 Оптимизация для архитектуры компьютера

Быстрое развитие архитектур вычислительных систем привело к развитию новых технологий компиляции. Практически все высокопроизводительные системы используют преимущества двух основных технологий: *параллелизм* (parallelism) и *иерархии памяти*. Параллелизм можно найти на нескольких уровнях: на *уровне команд*, когда одновременно могут выполняться несколько команд, и на *уровне процессора*, когда различные потоки одного приложения выполняются на разных процессорах. Иерархии памяти представляют собой ответ на основное ограниче-

ние, накладываемое на память: можно иметь очень быструю или очень большую память, но нельзя иметь память одновременно и большую, и быструю.

Параллелизм

Все современные микропроцессоры используют параллелизм на уровне команд, однако этот параллелизм может быть скрыт от программиста. Программы пишутся так, как если бы все их команды выполнялись последовательно; аппаратное обеспечение динамически проверяет зависимости в потоке последовательных команд и по возможности выполняет их параллельно. В некоторых случаях машина включает аппаратный планировщик, который может изменять порядок команд для увеличения параллелизма программ. Независимо от того, используется ли такое аппаратное переупорядочение, компиляторы могут так генерировать целевой код, чтобы более эффективно использовать параллелизм на уровне команд.

Параллелизм на уровне команд может проявляться в наборе команд и явным образом. Машины VLIW (Very Long Instruction Word — очень длинное слово команды) имеют команды, которые могут выполнять несколько операций параллельно. Примером такой архитектуры может служить процессор Intel IA64. Все высокопроизводительные микропроцессоры общего назначения также включают команды, которые могут работать одновременно с векторами данных. Современные методы компиляции позволяют автоматически генерировать из последовательных программ код для таких машин.

Постепенно становятся преобладающими многопроцессорные системы — даже персональные компьютеры все чаще имеют несколько процессоров. Многопоточный код для работы в многопроцессорной среде может как явно писаться программистом, так и автоматически генерироваться компилятором из обычной последовательной программы. Такой компилятор скрывает от программистов детали обнаруженного в программе параллелизма, распределяя вычисления по процессорам компьютера и минимизируя необходимые синхронизацию и межпроцессорный обмен. Многие научно-вычислительные и инженерные приложения содержат интенсивные вычисления и могут эффективно использовать преимущества параллельной работы процессоров. В настоящее время имеются методы распараллеливания, которые автоматически транслируют последовательные научные программы в многопроцессорный код.

Иерархии памяти

Иерархия памяти состоит из нескольких уровней с различными скоростями и размерами, причем чем уровень ближе к процессору, тем выше его скорость и меньше размер. Среднее время обращения программы к памяти снижается, если большинство ее обращений удовлетворяется более быстрым уровнем в иерархии. Параллелизм и наличие иерархии памяти повышают потенциальную производительность компьютера, но компилятор должен приложить усилия, чтобы эта по-

тенциальная производительность машины стала реальной производительностью приложения.

Иерархии памяти имеются на всех машинах. Процессор обычно использует небольшое количество регистров объемом в десятки байтов, несколько уровней кэш-памяти объемом от килобайтов до мегабайтов, физической памяти — от мегабайтов до гигабайтов и вторичную память — от гигабайтов и выше. Соответственно, скорость доступа к соседним в иерархии уровням памяти может отличаться на два или три порядка. Зачастую производительность системы ограничена не скоростью процессора, а производительностью подсистемы памяти. В то время как традиционно компиляторы уделяли большее внимание оптимизации работы процессора, сегодня больший упор следует делать на более эффективное использование иерархии памяти.

Эффективное использование регистров — пожалуй, самая важная задача оптимизации программы. Но в отличие от регистров, которые должны явно управляться программным обеспечением, кэши и физическая память скрыты от набора команд и управляются аппаратным обеспечением. Как выяснилось, политики управления кэшами, реализуемые аппаратным обеспечением, в ряде случаев неэффективны, в частности в научных вычислениях с большими структурами данных (обычно массивами). Повысить эффективность иерархии памяти можно путем изменения размещения данных или порядка команд, обращающихся к данным. Можно также изменить размещение кода для повышения эффективности кэшей команд.

1.5.3 Разработка новых архитектур компьютеров

В свое время, на заре компьютерной эры, компиляторы разрабатывались после того, как создавались компьютеры. С тех пор ситуация изменилась. Поскольку программирование на языках высокого уровня стало нормой, производительность вычислительных систем определяется не просто скоростью процессоров, но и тем, насколько хорошо компиляторы могут использовать их новые возможности. Таким образом, при разработке архитектур современных вычислительных систем компиляторы разрабатываются на стадии проектирования процессора, и скомпилированный код, выполняемый на имитаторах, используется для оценки предлагаемых архитектурных возможностей.

RISC

Одним из хорошо известных примеров, когда компиляторы повлияли на разработку архитектуры компьютера, было изобретение архитектуры RISC (Reduced Instruction-Set Computer — компьютер с сокращенным набором команд). До этого изобретения основная тенденция состояла в разработке все более сложных наборов команд, предназначенных для упрощения программирования на языке

ассемблера; такие архитектуры известны как CISC (Complex Instruction-Set Computer — компьютер со сложным набором команд). Например, набор команд CISC включает сложные режимы адресации памяти для поддержки обращения к структурам, а также команды вызова процедур с сохранением регистров и передачей параметров в стекс.

Методы оптимизации зачастую способны свести такие команды к небольшому количеству более простых операций, устраняя избыточность сложных команд. Таким образом, желательно построить простые наборы команд, которые способны эффективно использоваться компиляторами и существенно облегчить оптимизацию аппаратного обеспечения.

Большинство архитектур процессоров общего назначения, включая PowerPC, SPARC, MIPS, Alpha и PA-RISC, основаны на концепции RISC. Хотя архитектура x86 — наиболее популярного процессора — имеет набор команд CISC, при реализации этого процессора применены многие идеи, разработанные для RISC-машин. Более того, самый эффективный способ использования высокопроизводительных компьютеров x86 состоит в использовании только простейших команд процессора.

Специализированные архитектуры

За последние три десятилетия было предложено множество архитектурных концепций, включающих потоковые машины (data flow machine), векторные машины, машины VLIW (Very Long Instruction Word — архитектура с командными словами очень большой длины), SIMD (Single Instruction, Multiple Data — один поток команд, много потоков данных), массивы процессоров, систолические матрицы, мультипроцессоры с разделяемой памятью и мультипроцессоры с распределенной памятью. Разработка каждой из этих архитектурных концепций сопровождалась исследованиями и разработкой соответствующих технологий компиляции.

Некоторые из перечисленных идей нашли свое место во встраиваемых машинах. Поскольку целая система может быть собрана в виде единой схемы, процессоры больше не рассматриваются как отдельные товарные единицы, зато могут изготавливаться на заказ для достижения наивысшей эффективности решения тех или иных задач. Таким образом, в отличие от процессоров общего назначения, когда экономика заставляет использовать, по сути, единые архитектурные решения, специализированные процессоры для конкретных приложений демонстрируют огромное разнообразие архитектурных решений. Технологии компиляции должны не только поддерживать программирование для таких архитектур, но и участвовать в принятии решения об использовании той или иной архитектуры.

1.5.4 Трансляции программ

Хотя обычно мы рассматриваем компиляцию как трансляцию с высокоуровневого языка программирования на машинный уровень, та же технология применима и для трансляции между различными видами языков программирования. Далее приведены некоторые важные применения технологии трансляции.

Бинарная трансляция

Методы компиляции могут использоваться для трансляции бинарного кода для одной машины в код для другой, обеспечивая выполнение машиной программы, изначально скомпилированной для другого набора машинных команд. Методы бинарной трансляции использовались различными компьютерными компаниями для повышения доступности программного обеспечения для их машин. В частности, доминирование рынка персональных компьютеров с процессорами x86 привело к тому, что бóльшая часть программного обеспечения оказалась доступна в виде кода для x86. Были разработаны бинарные трансляторы для конвертации кода x86 в коды Alpha и Sparc. Бинарная трансляция использовалась также фирмой *Transmeta, Inc.* в реализации набора команд x86. Процессор Transmeta Crusoe представлял собой LVIW-процессор, который вместо выполнения сложных команд x86 непосредственно аппаратным обеспечением использовал трансляцию для конвертации кода x86 в код LVIW.

Бинарная трансляция может также использоваться для обеспечения обратной совместимости. Когда процессоры в Apple Macintosh в 1994 году были сменены с Motorola MC 68040 на PowerPC, то для того, чтобы процессоры PowerPC могли выполнять старый код MC 68040, использовалась бинарная трансляция.

Аппаратный синтез

На высокоуровневых языках пишется не только программное обеспечение; даже проектирование аппаратного обеспечения, в основном, описывается с помощью специализированных высокоуровневых языков, предназначенных для описания аппаратного обеспечения, таких как Verilog и VHDL (Very high speed integrated circuit Hardware Description Language — язык описания очень высокоскоростных интегральных схем аппаратного обеспечения). Дизайн аппаратного обеспечения обычно описывается на уровне регистровых передач (RTL — Register Transfer Level), где переменные представляют регистры, а выражения — комбинационные логические схемы. Инструментарий аппаратного синтеза автоматически транслирует RTL-описания в логические вентили, которые отображаются в транзисторы и в конечном счете — в физические схемы. В отличие от компиляторов для языков программирования, такие инструменты зачастую выполняют оптимизацию очень длительное время — часами. Существуют также методы трансляции проектов на более высоких уровнях, таких как, например, функциональный.

Интерпретаторы запросов к базам данных

Языки полезны не только в программном и аппаратном обеспечении, но и во многих приложениях, например языки запросов, в особенности SQL (Structured Query Language — язык структурированных запросов), использующиеся для выполнения поиска в базах данных. Запросы к базам данных состоят из предикатов, содержащих реляционные и булевы операторы. Они могут интерпретироваться или компилироваться в команды для поиска в базе данных записей, удовлетворяющих указанному предикату.

Компилируемое моделирование

Имитационное моделирование (simulation) представляет собой метод, используемый во многих научных и инженерных дисциплинах, например, для лучшего понимания явлений или проверки конструкций. Входными данными для имитатора обычно включают описание проекта и конкретные входные параметры для запуска моделирования определенной ситуации. Имитация может быть очень дорогой. Обычно требуется имитировать множество возможных альтернативных проектов для разных входных данных, причем каждый эксперимент может занимать дни вычислений на высокопроизводительных машинах. Вместо написания имитатора, интерпретирующего проект, стоит разработать компилятор, который даст на выходе машинный код, что приведет к выигрышу времени. Такое компилируемое моделирование может ускорить работу на порядки по сравнению с интерпретатором. Этот метод используется во многих реальных инструментах, имитирующих проекты, написанные на Verilog или VHDL.

1.5.5 Инструментарий для повышения производительности программного обеспечения

Программы, вероятно, являются наиболее сложным продуктом, производимым человеком. Они состоят из множества фрагментов, каждый из которых должен быть совершенно безошибочным, чтобы программа работала корректно. Ошибки в программах — настоящий бич; они могут приводить к краху системы, получению неверных результатов, делать систему уязвимой для внешних атак и даже приводить к катастрофическим последствиям, таким как выход из строя техники и даже гибель людей. Главный способ выявления ошибок в программах — тестирование.

Интересным и многообещающим дополнительным подходом является использование анализа потока данных для статического (до запуска программы) обнаружения ошибок. В ходе анализа потоков данных можно обнаружить ошибки во всех возможных путях выполнения, а не только в тех, которые реально выполняются для данного входного набора данных, как в случае тестирования программы. Множество методов анализа потоков данных, изначально разработанных

для оптимизации, могут использоваться для создания инструментов, помогающих программисту при разработке программного обеспечения.

Задача поиска всех ошибок в программе неразрешима. Анализ потоков данных может предупредить программиста обо всех возможных инструкциях, которые могут приводить к ошибкам определенной категории. Однако, если большинство таких предупреждений окажутся ложной тревогой, программист не будет пользоваться подобным инструментом. Поэтому на практике такие инструменты обнаружения ошибок оказываются не надежными и не всеохватывающим, т.е. они не обнаруживают все ошибки в программе и не все ошибки, которые они обнаруживают, на самом деле являются ошибками. Тем не менее различные методы статического анализа продемонстрировали свою эффективность в поиске ошибок, таких как разыменование нулевого и освобожденного указателя, в реальных программах. Факт ненадежности таких детекторов ошибок существенно отличает их от оптимизации компилятором. Оптимизация обязана быть консервативной и не может изменять семантику программы ни при каких условиях.

В оставшейся части данного подраздела мы упомянем несколько путей, которыми методы анализа программ (основанные на технологиях, изначально предназначенных для оптимизации кода компиляторами) могут повысить производительность программного обеспечения. Особо важны методы, позволяющие статически выявить уязвимые места в системе безопасности программ.

Проверка типов

Проверка типов — эффективная и хорошо обоснованная технология поиска несогласованностей в программе. Она может использоваться для поиска ошибок, например, когда операция применяется к объекту неверного типа или когда фактические параметры процедуры не соответствуют ее сигнатуре. Анализ программы может выполняться после поиска ошибок типов путем анализа потока данных в программе. Например, если указателю присваивается значение `null` и тут же выполняется его разыменование, очевидно, что программа содержит ошибку.

Тот же метод может использоваться для поиска различных "дыр" в системе безопасности, когда программе может быть передана строка или иные данные, которые неосторожно используются программой. Передаваемая пользователем строка может быть помечена как имеющая тип "опасный", и если не выполнена проверка корректности формата такой строки, она остается "опасной". Если строка с таким типом может влиять на поток управления в некоторой точке программы, то это потенциальное нарушение безопасности программы.

Проверка диапазона

Оказывается, гораздо легче допустить ошибку при программировании на низком уровне, чем на высоком. Например, многие нарушения безопасности вызываются переполнением буфера в программе, написанной на C. Поскольку в C

нет средств проверки диапазона массива, программист должен сам обеспечить невозможность обращения к памяти за пределами массива. Если в программе не выполняются проверки того, что предоставляемые пользователем данные не вызывают переполнения буфера, то программа может оказаться взломанной путем размещения части данных за пределами отведенного буфера. Может быть предпринята атака путем манипуляции входными данными таким образом, что программа начинает вести себя неверно и подвергает риску безопасность системы. Имеются методы, разработанные для поиска возможных переполнений буфера в программах, но пока что их успешность весьма ограничена.

Если же программа написана на безопасном языке с автоматической проверкой диапазона, то такая проблема никогда в ней не возникнет. Тот же анализ потоков данных, который используется для устранения излишних проверок диапазона, может использоваться и для поиска переполнений буфера. Основное отличие, однако, заключается в том, что если ошибка при удалении проверки диапазона может привести к небольшому замедлению работы программы, то ошибка при обнаружении потенциального переполнения буфера может привести к нарушению безопасности системы. Таким образом, в то время как для оптимизации проверки диапазона вполне адекватны простые методы, для получения высококачественных результатов в инструментах для поиска ошибок должны использоваться сложные технологии анализа, такие как отслеживание значений указателей при выполнении процедур.

Инструментарий для управления памятью

Сборка мусора — еще один прекрасный пример компромисса между эффективностью и сочетанием простоты программирования и надежности программного обеспечения. Автоматическое управление памятью устраняет все ошибки работы с памятью (например, утечки памяти), которые являются основным источником проблем в программах C и C++. Для помощи программисту в поисках ошибок управления памятью разработаны разнообразные инструменты. Например, широко используется такой инструмент, как Purify, который динамически отлавливает ошибки управления памятью. Разработаны также инструменты, которые могут выявлять некоторые проблемы статически, без выполнения программы.

1.6 Азы языков программирования

В этом разделе мы рассмотрим наиболее важные термины и понятия, встречающиеся при изучении языков программирования. Нашей целью не является рассмотрение всех концепций всех распространенных языков программирования. Мы полагаем, что читатель знаком как минимум с одним из языков программирования, C, C++ или Java, и сталкивался с другими языками программирования.

1.6.1 Понятия статического и динамического

Среди наиболее важных вопросов, с которыми мы встречаемся при разработке компилятора для языка программирования, является вопрос о том, какие решения может принимать компилятор. Если язык использует стратегию, позволяющую компилятору принимать решения, то мы говорим, что язык использует *статическую* стратегию или что вопросы могут решаться *во время компиляции*. С другой стороны, стратегия, которая позволяет принимать решения только в процессе выполнения программы, называется *динамической* или принимающей решения *во время выполнения*.

Один из вопросов, которые мы хотим рассмотреть, — это вопрос об области видимости объявлений. *Областью видимости* (scope) объявления x называется область программы, в которой используется x из этого объявления. Язык использует *статическую область видимости* или *лексическую область видимости*, если он может определить область видимости объявления при рассмотрении исходной программы. В противном случае язык использует *динамическую область видимости*. При динамической области видимости в процессе выполнения программы одинаковое использование x может работать с несколькими различными объявлениями x.

Большинство языков программирования, такие как C и Java, используют статическую область видимости. Мы рассмотрим ее в подразделе 1.6.3.

Пример 1.3. В качестве еще одного примера понятия статического и динамического рассмотрим использование термина "статический" в применении к данным в объявлении класса Java. В Java переменная представляет собой имя для ячейки памяти, использующейся для хранения ее значения. Здесь "статический" означает не область видимости переменной, а способность компилятора определить ячейку памяти, в которой может быть найдена объявленная переменная. Объявление наподобие

```
public static int x;
```

делает x *переменной класса* и гласит, что имеется только одна копия x, независимо от общего количества созданных объектов класса. Более того, компилятор может определить ячейку памяти, в которой будет храниться целое значение x. Напротив, если убрать из этого объявления `static`, то каждый объект класса будет иметь собственную ячейку памяти, в которой будет храниться значение x, и компилятор не в состоянии заранее определить все эти ячейки в программе, которая будет выполняться. □

1.6.2 Среды и состояния

Еще один важный вопрос, который следует рассмотреть при обсуждении языков программирования, — влияют ли изменения во время выполнения программы

на значения элементов данных, или на интерпретацию имен этих данных. Например, выполнение присваивания `x = y + 1` изменяет значение, обозначаемое как x. Говоря более строго, присваивание изменяет значение в ячейке памяти, обозначенной как x.

Может оказаться менее понятным, что сама ячейка, обозначаемая как x, может измениться при выполнении программы. Например, как было в примере 1.3, если x не является статической переменной (переменной класса), то каждый объект класса имеет собственную ячейку памяти для экземпляра переменной x. В этом случае присваивание переменной x может изменить любой из этих экземпляров переменных в зависимости от того, какому объекту принадлежит метод, содержащий это присваивание.

Связь имен с местоположением в памяти и затем со значениями может быть описана как два отображения, изменяющиеся по ходу выполнения программы (рис. 1.8).

1. *Среда* (environment) — это отображение имен на ячейки памяти. Поскольку переменные указывают на местоположения в памяти ("lvalue" в терминологии C), можно также определить среду как отображение имен на переменные.

2. *Состояние* (state) представляет собой отображение местоположений в памяти на их значения, т.е. в терминологии C состояние отображает lvalue на соответствующие rvalue.

Среды изменяются в соответствии с правилами областей видимости языка.

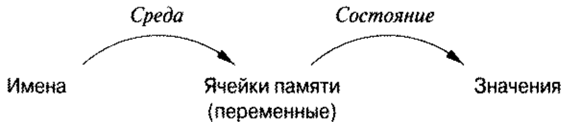


Рис. 1.8. Двухстадийное отображение имен на значения

Пример 1.4. Рассмотрим фрагмент программы на C (рис. 1.9). Целое i объявлено как глобальная переменная, а затем — как локальная переменная функции f. При выполнении f среда изменяется так, что имя i ссылается на ячейку памяти, зарезервированную для i, локального для функции f, так что явно показанное присваивание `i = 3` обращается к этому месту в памяти. Обычно такая локальная переменная i располагается в стеке времени выполнения.

При выполнении функции g, отличной от f, использование i не может означать обращение к локальной переменной i функции f. Это использование i в функции g должно находиться в области видимости некоторого другого объявления i. Явно

указанное присваивание `x = i + 1` располагается в некоторой иной процедуре, определение которой здесь не показано. По-видимому, i в выражении `i + 1` означает глобальную переменную i. Как и в большинстве языков программирования, объявления имен в C должны предшествовать их использованию, так что функция, расположенная до объявления глобальной переменной i, не может обращаться к ней. □

```
...
int i;                  /* Глобальное i              */
...
void f(···) {
    int i;              /* Локальное i               */
    ...
    i = 3;              /* Использование локального i */
    ...
}
...
    x = i + 1;          /* Использование глобального i */
```

Рис. 1.9. Два объявления имени i

Отображения среды и состояния на рис. 1.8 динамические, но из этого правила имеется несколько исключений.

1. *Статическое и динамическое связывание* имен с ячейками памяти. В основном, связывание имен с ячейками памяти динамическое, и мы рассматривали несколько подходов к такому связыванию на протяжении раздела. Некоторые объявления, такие как глобальная переменная i на рис. 1.9, могут определять местоположение в памяти раз и навсегда при генерации объектного кода компилятором.[3]

2. *Статическое и динамическое связывание* ячеек памяти со значениями. Связывание ячеек памяти со значениями (вторая стадия на рис. 1.8) также в общем случае динамическое, поскольку до выполнения программы мы

[3]Технически компилятор C назначает глобальной переменной i местоположение в виртуальной памяти, оставляя задачу определения конкретного местоположения i в физической памяти машины загрузчику и операционной системе. Однако мы не должны беспокоиться о вопросах такого "перемещения", которое не оказывает никакого влияния на процесс компиляции. Вместо этого мы рассматриваем адресное пространство, используемое компилятором для выходного кода, как если бы это была физическая память.

Имена, идентификаторы и переменные

Хотя термины "имя" и "переменная" часто означают одно и то же, мы аккуратно используем их для того, чтобы отличать имена времени компиляции от местоположений в памяти во время выполнения программы, обозначаемых этими именами.

Идентификатор (identifier) представляет собой строку символов, обычно букв или цифр, которая ссылается (идентифицирует) на некоторую сущность, такую как объект данных, процедура, класс или тип. Все идентификаторы являются именами, но не все имена — идентификаторы. Имена могут также быть выражениями. Например, имя $x.y$ может означать поле y структуры, обозначенной как x. Здесь x и y — идентификаторы, в то время как $x.y$ — имя, но не идентификатор. Составные имена наподобие $x.y$ называются квалифицированными именами (qualified names).

Переменная (variable) ссылается на конкретное местоположение в памяти. Достаточно распространена ситуация, когда один и тот же идентификатор объявлен более одного раза; каждое такое объявление вводит новую переменную. Даже если каждый идентификатор объявлен только один раз, идентификатор, локальный для рекурсивной процедуры, в различные моменты времени обращается к различным местоположениям в памяти.

не можем сказать, какое значение находится в конкретной ячейке памяти. Исключением является объявление констант. Например, определение языка программирования C

```
#define ARRAYSIZE 1000
```

статически связывает имя ARRAYSIZE со значением 1000. Мы можем определить это связывание, взглянув на указанную инструкцию, и мы знаем, что изменение данного связывания в процессе выполнения программы невозможно.

1.6.3 Статическая область видимости и блочная структура

В большинстве языков программирования, включая C и его семейство, используется статическая область видимости. Правила области видимости для C основаны на структуре программы; область видимости объявления явно определяется тем, где в программе находится это объявление. Более поздние языки

программирования, такие как C++, Java и C#, предоставляют явные средства управления областями видимости при помощи ключевых слов наподобие **public**, **private** и **protected**.

В этом разделе мы рассмотрим правила статических областей видимости для языков программирования с блоками, где *блок* (block) представляет собой сгруппированные объявления и инструкции. В языке программирования C для ограничения блока используются фигурные скобки { и }; той же цели служили ключевые слова **begin** и **end** в Algol.

Пример 1.5. В качестве первого приближения стратегия статических областей видимости языка программирования C следующая.

1. Программа на C состоит из последовательности объявлений переменных и функций верхнего уровня.

2. Функции могут содержать в себе объявления переменных, где переменные включают локальные переменные и параметры. Область видимости каждого такого объявления ограничена функцией, в котором оно появляется.

3. Область видимости объявления верхнего уровня имени x состоит из всей программы, следующей после объявления, за исключением инструкций, которые располагаются в функции, также содержащей объявление x.

Дополнительные детали стратегии касаются объявлений переменных в инструкциях. Такие объявления будут рассмотрены в упражнении 1.6. □

В языке программирования C синтаксис блока определяется следующими правилами.

1. Инструкция одного типа является блоком. Блоки могут быть везде, где могут быть инструкции другого типа, такие как инструкции присваивания.

2. Блок представляет собой последовательность объявлений, за которой следует последовательность инструкций, и все вместе они заключены в фигурные скобки.

Обратите внимание на то, что данный синтаксис позволяет блокам быть вложенными друг в друга. Об этом свойстве вложенности говорят как о *блочной структуре* (block structure). Семейство языков программирования C имеет блочную структуру, с тем исключением, что одна функция не может быть определена внутри другой функции.

Мы говорим, что объявление D "принадлежит" блоку B, если B — наиболее близкий вложенный блок, содержащий D; т.е. D находится в B, но не в блоке, вложенном в B.

Процедуры, функции и методы

Чтобы избежать повторения "процедуры, функции и методы" всякий раз, когда мы будем говорить о могущих быть вызванными подпрограммах, мы будем обычно употреблять термин "процедуры". Исключением является упоминание о программах на языках программирования наподобие C, в которых есть только функции — здесь мы будем говорить о них с использованием термина "функции". Когда речь пойдет о языке наподобие Java, в котором есть только методы, мы, соответственно, будем использовать именно этот термин.

В общем случае функция возвращает значение некоторого типа (тип возвращаемого значения), в то время как процедура никаких значений не возвращает. C и аналогичные ему языки программирования, в которых имеются только функции, рассматривают процедуры как функции со специальным типом возвращаемого значения "`void`", чтобы указать, что они не возвращают никаких значений. Объектно-ориентированные языки программирования наподобие Java и C++ используют термин "методы". Они могут вести себя, как функции или процедуры, но при этом они связаны с конкретным классом.

Правило статической области видимости для объявлений переменных в языках программирования с блочной структурой следующее. Если объявление D имени x принадлежит блоку B, то областью видимости D является весь блок B, за исключением любого блока B', вложенного в блок B на произвольной глубине, в котором повторно объявлено имя x. Имя x повторно объявлено в блоке B', если этому блоку принадлежит некоторое иное объявление D' того же имени x.

Эквивалентный способ выразить это правило заключается в том, чтобы сосредоточиться на использовании имени x. Пусть $B_1, B_2, \ldots, B_k$ — блоки, окружающие некоторое использование x, причем B_k — наименьший блок, вложенный в B_{k-1}, который, в свою очередь, вложен в блок B_{k-2}, и т.д. Найдем наибольшее i, для которого имеется объявление x, принадлежащее блоку B_i. В таком случае рассматриваемое использование x ссылается на объявление в блоке B_i. Можно сказать, что рассматриваемое использование x находится в области видимости объявления в блоке B_i.

Пример 1.6. Программа на языке программирования C++, приведенная на рис. 1.10, имеет четыре блока, содержащие ряд определений переменных a и b. Для облегчения запоминания каждое объявление инициализирует переменную значением, равным номеру блока, которому оно принадлежит.

Рассмотрим, например, объявление `int a = 1` в блоке B_1. Областью его видимости является весь блок B_1, за исключением тех блоков, вложенных (воз-

```
main() {
    int a = 1;                                  B1
    int b = 1;
    {
        int b = 2;                              B2
        {
            int a = 3;                          B3
            cout << a << b;
        }
        {
            int b = 4;                          B4
            cout << a << b;
        }
        cout << a << b;
    }
    cout << a << b;
}
```

Рис. 1.10. Блоки в программе на языке C++

можно, глубоко) в блок B_1, которые содержат собственное объявление a. Блок B_2, вложенный непосредственно в блок B_1, не содержит объявления a; такое объявление содержит блок B_3. Блок B_4 не содержит объявления a, так что блок B_3 – единственное место в программе, находящееся вне области видимости объявления a, принадлежащего блоку B_1. Иными словами, область видимости включает блок B_4 и весь блок B_2, за исключением части B_2, находящейся в блоке B_3. Области видимости всех пяти объявлений приведены на рис. 1.11.

Объявление	Область видимости
`int a = 1;`	$B_1 - B_3$
`int b = 1;`	$B_1 - B_2$
`int b = 2;`	$B_2 - B_4$
`int a = 3;`	B_3
`int b = 4;`	B_4

Рис. 1.11. Области видимости объявлений из примера 1.6

С другой точки зрения, рассмотрим инструкцию вывода в блоке B_4 и определим, какие объявления действуют для использованных в этой инструкции имен a и b. Список охватывающих блоков в порядке увеличения их размера – B_4, B_2, B_1. Заметим, что блок B_3 не являются охватывающим для рассматриваемой точки. Блок B_4 содержит объявление b, так что инструкция в блоке B_4 использует имен-

но это объявление и выводит значение b, равное 4. Однако объявления переменной a в блоке B_4 нет, так что мы обращаемся к блоку B_2. В нем также нет объявления переменной a, и мы переходим к следующему блоку — B_1. К счастью, здесь имеется объявление `int a = 1`, так что изначально рассматриваемая инструкция в блоке B_4 выводит значение a, равное 1. Если бы этого объявления не было, программа была бы некорректной. □

1.6.4 Явное управление доступом

Классы и структуры вводят новую область видимости для своих членов. Если p — объект класса с полем (членом) x, то x в $p.x$ ссылается на поле x в определении класса. По аналогии с блочной структурой область видимости объявления члена x в классе C распространяется на любой подкласс C', за исключением ситуации, в которой C' содержит локальное объявление того же имени x.

При помощи ключевых слов наподобие **public**, **private** и **protected** объектно-ориентированные языки программирования, такие как C++ и Java, предоставляют возможность явно управлять доступом к именам членов в надклассе. Эти ключевые слова обеспечивают *инкапсуляцию* (encapsulation) путем ограничения доступа. Так, для закрытых (private) имен преднамеренно ограничивается область видимости, которая включает только объявления и определения методов данного класса и "дружественных" классов[4] (термин C++). Защищенные (protected) имена доступны подклассам, а открытые (public) имена доступны вне класса.[5]

В C++ определение класса может быть отделено от определения всех или некоторых его методов. Таким образом, имя x, связанное с классом C, может иметь область кода, который находится вне его области видимости и за которым следует другая область (определение метода), вновь входящая в область видимости. В реальной ситуации области внутри и вне области видимости могут чередоваться, пока не будут определены все методы.

1.6.5 Динамическая область видимости

Технически стратегия областей видимости динамическая, если она основана на факторах, которые могут быть известны только при выполнении программы. Однако термин *динамическая область видимости* обычно означает следующую

[4]А также дружественных функций. — *Прим. ред.*

[5]Здесь имеется расхождение между понятием области видимости в данной книге (которое звучит как "где именно в программе может использоваться значение имени") и в языке программирования; например, в C++ речь идет об области корректности имени, где под этим понимается возможность использования неквалифицированного имени (см. раздел 3.3 стандарта C++). Поэтому, например, закрытое имя в программе C++ может быть видимым (в частности, участвовать в разрешении перегрузки функции), но при этом недоступным (см., например, задачу 16 в книге Г. Саттер. *Новые сложные задачи на C++*. — М.: Издательский дом "Вильямс", 2005). — *Прим. ред.*

Объявления и определения

Несмотря на кажущуюся схожесть терминов "объявление" (declaration) и "определение" (definition), на самом деле эти концепции языков программирования существенно различаются. Объявления говорят нам о типах сущностей, в то время как определения — об их значениях. Так, `int i` — это объявление i, а `i = 1` — определение i.

Различие еще более существенно, когда мы имеем дело с методами или иными процедурами. В С++ метод объявляется в определении класса путем указания, помимо его имени, типов его аргументов и типа результата (о таком объявлении часто говорят как о *сигнатуре* метода). Затем, в другом месте, метод определяется, т.е. приводится код, выполняемый данным методом. В С весьма распространены аналогичные ситуации, когда определение и объявление функции находятся в разных файлах, отличных от файлов, в которых эта функция используется.

стратегию: использование имени x обращается к объявлению x в последней вызванной (но еще не завершенной) процедуре с таким объявлением. Динамическая область видимости такого вида возникает только в особых случаях. Мы рассмотрим два примера динамической стратегии: раскрытие макросов в препроцессоре С и разрешение методов в объектно-ориентированном программировании.

Пример 1.7. В программе С на рис. 1.12 идентификатор a представляет собой макрос, который обозначает выражение $(x+1)$. Но что такое x? Мы не можем разрешить x статически, т.е. в терминах текста программы.

```
#define a (x+1)

int x = 2;

void b() { int x = 1; printf("%d\n", a); }

void c() { printf("%d\n", a); }

void main() { b(); c(); }
```

Рис. 1.12. Динамические области видимости макросов

В действительности для интерпретации x мы должны использовать обычное правило динамической области видимости. Мы рассматриваем все активные в настоящий момент вызовы функций и выбираем из функций, у которых имеет-

Аналогия между динамическими и статическими видимостями

Хотя может иметься любое количество динамических и статических стратегий для видимостей, существует интересная взаимосвязь между обычным правилом области видимости (для блочной структуры) и обычной динамической стратегией. В определенном смысле динамическое правило — это время, а статическое — пространство. В то время как статическое правило требует от нас найти объявление, блок которого ближе всего к физическому местоположению использования, динамическое правило требует найти объявление, модуль (вызов процедуры) которого ближе всего по времени использования.

ся объявление x, ту, которая была вызвана последней. Именно это объявление x и имеется в виду при использовании x.

В примере на рис. 1.12 функция *main* сначала вызывает функцию b. При выполнении b выводит значение макроса a. Поскольку a должно быть заменено на $(x + 1)$, мы разрешаем это использование x к объявлению `int x = 1` в функции b. Причина такого разрешения в том, что функция b содержит объявление x, так что $(x + 1)$ в вызове `printf` в функции b ссылается на это объявление x. Следовательно, выводимое значение — 2.

После завершения b и вызова c мы снова сталкиваемся с выводом значения макроса a. Однако единственное имя x, доступное c, — это глобальная переменная x. Инструкция `printf` в функции c, таким образом, обращается к объявлению глобальной переменной x, и выводимое значение — 3. □

Разрешение динамической области видимости существенно также для полиморфных процедур, которые могут иметь два или более определения для одного и того же имени, зависящих только от типов аргументов. В некоторых языках, таких как ML (см. раздел 7.3.3), можно статически определить типы для всех применений имен, и в этом случае компилятор в состоянии заменить каждое использование имени процедуры p обращением к коду надлежащей процедуры. Однако в других языках, таких как Java и C++, бывают ситуации, когда компилятор не в состоянии решить такую задачу.

Пример 1.8. Отличительная особенность объектно-ориентированного программирования заключается в возможности каждого объекта вызывать в ответ на сообщение соответствующий метод. Другими словами, выполняемая процедура $x.m()$ зависит от класса объекта, обозначенного как x. Вот типичный пример.

1. Имеется класс C с методом $m()$.

2. D является подклассом C и имеет собственный метод $m()$.

3. Имеется использование m в виде $x.m\,()$, где x – объект класса C.

Обычно во время компиляции невозможно определить, принадлежит ли x классу C или подклассу D. Если метод вызывается несколько раз, весьма вероятно, что у части вызовов x будет принадлежать классу C, в то время как у остальных вызовов x будет принадлежать классу D. Какое именно определение m должно быть использовано, до выполнения программы выяснить невозможно. Таким образом, код, сгенерированный компилятором, должен определять класс объекта x и вызывать тот или иной метод с именем m. □

1.6.6 Механизмы передачи параметров

Все языки программирования имеют понятие процедуры, но они отличаются один от другого способами получения процедурой своих аргументов. В этом разделе мы рассмотрим, как *фактические параметры* (параметры, используемые в вызове процедуры) связываются с *формальными параметрами* (которые используются в определении процедуры). Как вызывающий процедуру код должен работать с параметрами, зависит от того, какой именно механизм передачи параметров применяется при вызове. Подавляющее большинство языков программирования использует либо "передачу по значению", либо "передачу по ссылке", либо оба эти механизма. Мы рассмотрим, что означают эти термины, а также еще один метод – "передачу по имени", который представляет, в первую очередь, исторический интерес.

Передача по значению

При *передаче по значению* фактический параметр вычисляется (если он представляет собой выражение) или копируется (если это переменная). Полученное значение помещается в местоположение, принадлежащее соответствующему формальному параметру вызываемой процедуры. Этот метод используется в C и Java и широко применяется в C++, так же, как и во многих других языках программирования. Передача по значению обладает той особенностью, что все вычисления, выполняемые над формальными параметрами вызываемой процедурой, являются локальными для этой процедуры, а сами фактические параметры при этом не могут быть изменены.

Заметим, однако, что в C можно передать указатель на переменную, что позволит вызываемой функции изменить ее значение. Аналогично имена массивов, передаваемые в качестве параметров в C, C++ или Java, по сути дают вызываемой процедуре указатель или ссылку на сам массив. Таким образом, если a – имя массива в вызывающей процедуре и оно передается по значению соответствующему формальному параметру x, то присваивание наподобие `x[i] = 2` в действительности изменяет элемент $a\,[i]$. Причина этого в том, что хотя x и по-

лучает копию значения *a*, это значение в действительности является указателем на начало области памяти, в которой располагается массив с именем *a*.

Аналогично в Java многие переменные на самом деле представляют собой ссылки, или указатели, на обозначаемые ими объекты. Это утверждение применимо к массивам, строкам и объектам всех классов. Несмотря на то что в Java используется только передача параметров по значению, при передаче имени объекта вызываемой процедуре значение, которое она получает, в действительности представляет собой указатель на объект. Таким образом, вызываемая процедура может влиять на значение самого объекта.

Передача по ссылке

При *передаче по ссылке* вызываемой процедуре в качестве значения соответствующего формального параметра передается значение адреса фактического параметра. Использование формального параметра в коде вызываемой процедуры реализуется путем следования по этому указателю к местоположению, указанному вызывающим кодом. Изменения формального параметра, таким образом, проявляются как изменения фактического параметра.

Однако если фактический параметр представляет собой выражение, то это выражение вычисляется перед вызовом, а результат вычисления помещается в собственное, отдельное местоположение в памяти. Изменения формального параметра приводят к изменениям значения в этом месте памяти, но никак не затрагивают данные вызывающей процедуры.

Передача по ссылке используется для ссылочных параметров в C++ и доступна во многих других языках программирования. Это практически единственный вариант передачи больших объектов, массивов или структур. Причина этого заключается в том, что при передаче по значению от вызывающего кода требуется копирование всего фактического параметра в место, принадлежащее соответствующему формальному параметру. Такое копирование становится слишком дорогостоящим с ростом размера параметра. Как уже отмечалось при рассмотрении передачи параметров по значению, языки программирования, такие как Java, решают задачу передачи массивов, строк и других объектов путем копирования ссылок на них. В результате Java ведет себя так, как если бы для всех типов, отличных от базовых (наподобие целых чисел или чисел с плавающей точкой), использовалась передача параметров по ссылке.

Передача по имени

Третий механизм — передача по имени — использовался в раннем языке программирования Algol 60. Этот способ требует, чтобы вызываемый код выполнялся, как если бы фактический параметр заменял формальный параметр в коде буквально, т.е. как если бы формальный параметр был макросом, обозначающим фактический параметр (с переименованием локальных имен в вызываемой проце-

дуре, чтобы они оставались различными). Если фактический параметр представлял собой выражение, а не переменную, применялось определенное интуитивное поведение кода, что и послужило одной из причин отказа от этого метода передачи параметров в современном программировании.

1.6.7 Псевдонимы

Передача параметров по ссылке (или ее имитация, как в случае Java, когда по значению передаются ссылки на объекты) имеет одно интересное следствие. Возможна ситуация, когда два формальных параметра будут ссылаться на одно и то же место в памяти; о таких переменных говорят, что они являются *псевдонимами* (aliases) друг друга. В результате любые две переменные, которые получают свои значения из двух различных формальных параметров, могут оказаться псевдонимами друг друга.

Пример 1.9. Предположим, что a — массив, принадлежащий процедуре p, и процедура p вызывает процедуру $q(x, y)$ при помощи вызова $q(a, a)$. Предположим также, что параметры передаются по значению, но имена массивов в действительности представляют собой ссылки на местоположения массивов в памяти, как в C или аналогичных языках программирования. Тогда x и y становятся псевдонимами друг друга. Важный момент заключается в том, что если внутри q имеется присваивание `x[10] = 2`, то значение $y[10]$ также становится равным 2. □

Оказывается, понимание псевдонимов и механизмов, создающих их, очень важно при оптимизации программ компилятором. Как вы увидите в начале главы 9, имеется много ситуаций, когда код может быть оптимизирован только при полной уверенности в том, что определенные переменные не являются псевдонимами. Например, можно определить, что `x = 2` — единственное место, где выполняется присваивание переменной x. Если это так, то можно заменить использование переменной x в программе на 2, например заменить `a = x + 3` более простым присваиванием `a = 5`. Но предположим, что у нас имеется другая переменная, y, являющаяся псевдонимом x. Тогда присваивание `y = 4` может вызвать неожиданное изменение значения x. Это также может означать, что замена `a = x + 3` присваиванием `a = 5` ошибочна и верное значение переменной a должно быть равно 7.

1.6.8 Упражнения к разделу 1.6

Упражнение 1.6.1. Для кода с блочной структурой на языке программирования C, приведенного на рис. 1.13, *а*, укажите значения, присваиваемые переменным w, x, y и z.

Упражнение 1.6.2. Для кода с блочной структурой на языке программирования C, приведенного на рис. 1.13, *б*, укажите значения, присваиваемые переменным w, x, y и z.

```
int w, x, y, z;
int i = 4; int j = 5;
{ int j = 7;
  i = 6;
  w = i + j;
}
x = i + j;
{ int i = 8;
  y = i + j;
}
z = i + j;
```

а) Код к упражнению 1.6.1

```
int w, x, y, z;
int i = 3; int j = 4;
{ int i = 5;
  w = i + j;
}
x = i + j;
{ int j = 6;
  i = 7;
  y = i + j;
}
z = i + j;
```

б) Код к упражнению 1.6.2

Рис. 1.13. Код с блочной структурой

Упражнение 1.6.3. Для кода с блочной структурой, приведенного на рис. 1.14, в предположении обычных статических правил областей видимости, укажите для каждого из двенадцати объявлений его область видимости.

```
{   int w, x, y, z;         /* Блок B1 */
    {   int x, z;           /* Блок B2 */
        {   int w, x;       /* Блок B3 */ }
    }
    {   int w, x;           /* Блок B4 */
        {   int y, z;       /* Блок B5 */ }
    }
}
```

Рис. 1.14. Код с блочной структурой к упражнению 1.6.3

Упражнение 1.6.4. Что будет выведено следующим фрагментом кода на языке программирования C?

```
#define a (x+1)
int x = 2;
```

```
void b() { x = a; printf("%d\n", x); }
void c() { int x = 1; printf("%d\n"), a; }
void main() { b(); c(); }
```

1.7 Резюме к главе 1

- *Языковые процессоры.* Интегрированная среда разработки программного обеспечения включает много различных видов языковых процессоров, таких как компиляторы, интерпретаторы, ассемблеры, компоновщики, загрузчики, отладчики, профайлеры.

- *Фазы компилятора.* Работа компилятора состоит из последовательности фаз, каждая из которых преобразует исходную программу из одного промежуточного представления в другое.

- *Машинные и ассемблерные языки.* Машинные языки представляют собой языки программирования первого поколения; за ними следуют языки ассемблера. Программирование на этих языках требует большого количества времени и чревато возникновением ошибок.

- *Моделирование в разработке компиляторов.* Проектирование компиляторов является одной из тех областей, в которых теория оказывает особо сильное влияние на практику. Модели, применяемые в этой области, включают автоматы, грамматики, регулярные выражения, деревья и многое другое.

- *Оптимизация кода.* Хотя код и не может быть "оптимизирован" в истинном понимании этого слова, наука о повышении эффективности кода очень важна (и очень сложна). Она представляет собой существенную часть изучения компиляции.

- *Высокоуровневые языки программирования.* С течением времени языки программирования берут на себя все бóльшую часть задач, ранее решавшихся программистом, таких как управление памятью, проверка согласованности типов и параллельное выполнение кода.

- *Компиляторы и архитектура вычислительных систем.* Технологии компиляции оказывают влияние на архитектуру компьютеров, так же как на них, в свою очередь, влияют достижения в области архитектуры вычислительных систем. Многие современные новшества в архитектуре зависят от способностей компиляторов эффективно использовать аппаратные возможности.

- *Производительность и безопасность программного обеспечения.* Те же методы, которые позволяют компилятору оптимизировать код, могут использоваться для решения ряда задач анализа программ, от поиска распространенных ошибок в программах до определения, насколько программы подвержены тому или иному из множества вариантов вторжения, разработанных "хакерами".

- *Правила областей видимости. Область видимости* объявления x представляет собой контекст, в котором использование x ссылается на это объявление. Язык, использующий *статическую*, или *лексическую* область видимости, в состоянии определить область видимости объявления на основе только исходного текста программы. В противном случае язык использует *динамические* области видимости.

- *Среды.* Связь имен с местоположениями в памяти и затем со значениями может быть описана в терминах *сред*, которые отображают имена на соответствующие им положения в памяти, и *состояний*, которые отображают местоположения на их значения.

- *Блочная структура.* Языки, которые допускают наличие вложенных блоков, называются имеющими *блочную структуру*. Имя x во вложенном блоке B находится в области видимости объявления D переменной x в охватывающем блоке, если не имеется другого объявления x в промежуточном блоке.

- *Передача параметров.* Параметры передаются от вызывающей процедуры к вызываемой по значению или по ссылке. При передаче больших объектов по значению передаваемые значения в действительности представляют собой ссылки на объекты, что в результате приводит к фактической передаче по ссылке.

- *Псевдонимы.* Когда параметры передаются по ссылке, два формальных параметра могут ссылаться на один и тот же объект. Такая возможность позволяет изменять одну переменную путем изменения другой переменной.

1.8 Список литературы к главе 1

Информацию о разработке языков программирования, которые были созданы и использовались до 1967 года, включая Fortran, Algol, Lisp и Simula, можно найти в [7]. Языки, созданные до 1982 года, включая C, C++, Pascal и Smalltalk, описаны в [1].

Набор компиляторов GNU (GNU Compiler Collection — gcc) — популярный источник компиляторов с открытым кодом для C, C++, Fortran, Java и других

языков программирования [2]. В [3] описывается Phoenix – набор инструментов для создания компиляторов, предоставляющий интегрированные средства для построения фаз анализа программ, генерации и оптимизации кода.

Дополнительную информацию о концепциях языков программирования можно найти в [5 и 6]; сведения об архитектуре компьютеров и ее влиянии на компиляцию содержатся в [4].

1. Bergin, T. J. and R. G. Gibson, *History of Programming Languages*, ACM Press, New York, 1996.

2. `http://gcc.gnu.org/`.

3. `http://research.microsoft.com/phoenix/default.aspx`.

4. Hennessy, J. L. and D. A. Patterson, *Computer Organization and Design: The Hardware/Software Interface*, Morgan-Kaufmann, San Francisco, CA, 2004.

5. Scott, M. L., *Programming Language Pragmatics, second edition*, Morgan-Kaufmann, San Francisco, CA, 2006.

6. Sethi, R., *Programming Languages: Concepts and Constructs*, Addison-Wesley, 1996.

7. Wexelblat, R. L., *History of Programming Languages*, Academic Press, New York, 1981.

Глава 2

Простой синтаксически управляемый транслятор

Данная глава представляет собой введение в методы компиляции, рассматриваемые в главах 3–6 данной книги. Эти методы проиллюстрированы в ней на примере создания работающей программы на языке программирования Java, которая транслирует типичные инструкции языка программирования в трехадресный код. В этой главе основной упор сделан на начальную стадию компиляции, в частности на лексический анализ, синтаксический анализ и генерацию промежуточного кода. В главах 7 и 8 рассматривается генерация машинных команд из трехадресного кода.

Мы начнем с создания синтаксически управляемого транслятора, который отображает инфиксные арифметические выражения в постфиксные. Затем мы расширим этот транслятор так, чтобы он мог отображать код, приведенный на рис. 2.1, в трехадресный код показанного на рис. 2.2 вида.

Работающий транслятор на языке программирования Java приведен в приложении А. Использовать Java удобно, но не необходимо. В действительности идеи этой главы предшествовали созданию как самого Java, так и С.

```
{
    int i; int j; float[100] a; float v; float x;
    while ( true ) {
        do i = i+1; while ( a[i] < v );
        do j = j-1; while ( a[j] > v );
        if ( i >= j ) break;
        x = a[i]; a[i] = a[j]; a[j] = x;
    }
}
```

Рис. 2.1. Фрагмент транслируемого кода

```
 1:  i = i + 1
 2:  t1 = a [ i ]
 3:  if t1 < v goto 1
 4:  j = j - 1
 5:  t2 = a [ j ]
 6:  if t2 > v goto 4
 7:  ifFalse i >= j goto 9
 8:  goto 14
 9:  x = a [ i ]
10:  t3 = a [ j ]
11:  a [ i ] = t3
12:  a [ j ] = x
13:  goto 1
14:
```

Рис. 2.2. Упрощенный промежуточный код для фрагмента программы, представленного на рис. 2.1

2.1 Введение

Фаза анализа компилятора разбивает исходную программу на составные части и производит ее внутреннее представление, называющееся промежуточным кодом. Фаза синтеза транслирует промежуточный код в целевую программу.

Анализ организован вокруг "синтаксиса" компилируемого языка. *Синтаксис* языка программирования описывает корректный вид его программ, в то время как *семантика* языка определяет смысл написанной на нем программы, т.е. что именно делает каждая программа при выполнении. Для определения синтаксиса в разделе 2.2 будет представлена широко используемая запись, именуемая контекстно-свободной грамматикой или BNF (Backus–Naur Form — форма Бэкуса–Наура). На сегодня описание семантики языка — гораздо более сложная задача, чем описание синтаксиса; для ее решения необходимо использовать неформальные описания и примеры.

Кроме определения синтаксиса языка, контекстно-свободная грамматика используется при трансляции программ. В разделе 2.3 будет рассмотрена грамматически управляемая технология компиляции, известная также как *синтаксически управляемая трансляция* (syntax-directed translation). Синтаксический анализ будет рассматриваться в разделе 2.4.

Остальная часть главы представляет собой краткий обзор модели начальной стадии компилятора, показанной на рис. 2.3. Мы начнем с синтаксического анализатора. Для простоты будет рассматриваться синтаксически управляемая трансляция инфиксных выражений в постфиксный вид, запись, в которой операторы следуют за их операндами. Например, постфиксной формой выражения $9 - 5 + 2$ является $95 - 2 +$. Трансляция в постфиксную форму достаточно богата для иллюстрации синтаксического анализа и достаточно проста, чтобы полностью привести транслятор в разделе 2.5. Простой транслятор обрабатывает выражения наподобие $9 - 5 + 2$, состоящие из цифр, разделенных знаками "плюс" и "минус". Одна из причин, по которой мы начинаем с таких простых выражений, состоит в том, что синтаксический анализатор может работать непосредственно с отдельными символами, представляющими операторы и операнды.

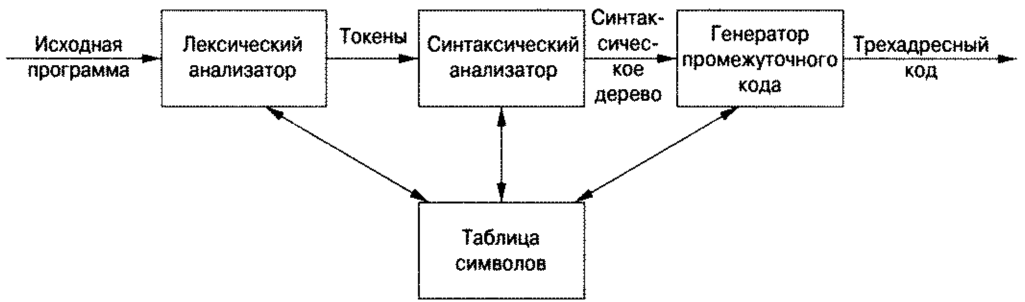


Рис. 2.3. Модель начальной стадии компилятора

Лексический анализатор позволяет транслятору в процессе синтаксического анализа работать с многосимвольными конструкциями наподобие идентификаторов, которые записываются как последовательность символов, но рассматриваются как единицы трансляции, называющиеся токенами. Например, в выражении `count + 1` идентификатор `count` рассматривается как отдельная единица трансляции. Лексический анализатор из раздела 2.6 позволяет выражениям состоять из чисел, идентификаторов и "пробельных символов" (пробелов, символов табуляции и новой строки).

Затем мы рассмотрим генерацию промежуточного кода. Два вида промежуточного кода продемонстрированы на рис. 2.4. Первый вид называется *абстрактными синтаксическими деревьями* или просто *синтаксическими деревьями*, представляющими иерархическую синтаксическую структуру исходной программы. В модели на рис. 2.3 синтаксический анализатор порождает синтаксическое дерево, которое позже транслируется в трехадресный код. Некоторые простые компиляторы объединяют синтаксический анализ и генерацию промежуточного кода в один компонент.

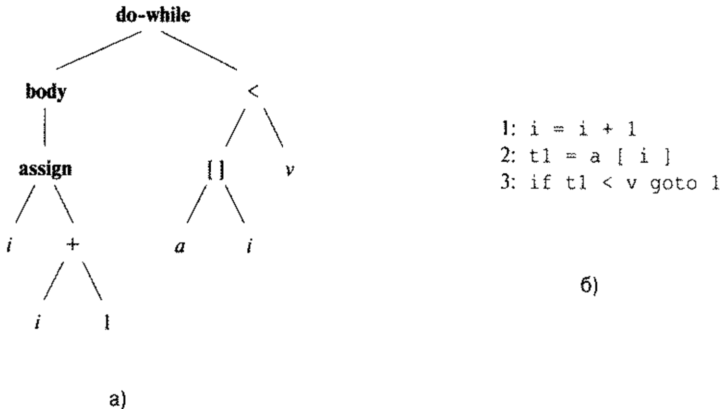


Рис. 2.4. Промежуточный код для фрагмента "`do i=i+1; while(a[i]<v);`"

Корень абстрактного синтаксического дерева на рис. 2.4, *а* представляет собой весь цикл **do-while**. Левый дочерний узел корня представляет тело цикла, состоящее из единственного присваивания `i=i+1;`. Правый дочерний узел корня представляет условие цикла `a[i]<v`. Реализация синтаксических деревьев рассматривается в разделе 2.8.

Второе распространенное промежуточное представление, показанное на рис. 2.4, *б*, является последовательностью "трехадресных" команд. Более полный пример приведен на рис. 2.2. Этот вид промежуточного кода получил свое название от команд вида $x = y$ **op** z, где **op** — бинарный оператор, y и z — адреса операндов, а x — адрес результата операции. Трехадресные команды выполняют не более одной операции, обычно вычисления, сравнения или ветвления.

В приложении А мы применим методы из этой главы для построения начальной стадии компилятора на языке программирования Java, которая транслирует инструкции в команды ассемблерного уровня.

2.2 Определение синтаксиса

В этом разделе мы рассмотрим запись — "контекстно-свободную грамматику" или для краткости просто "грамматику", — которая используется для определения синтаксиса языка программирования. Грамматики будут использоваться как часть спецификации начальной стадии компилятора на протяжении всей книги.

Грамматика естественным образом описывает иерархическую структуру множества конструкций языка программирования. Например, инструкция **if-else** в Java имеет вид

if (*выражение*) *инструкция* **else** *инструкция*

Таким образом, инструкция **if-else** представляет собой ключевое слово **if**, за которым следуют открывающая круглая скобка, выражение, закрывающая скобка, инструкция, ключевое слово **else** и еще одна инструкция. Используя переменную *expr* для обозначения выражения и переменную *stmt* для обозначения инструкции, можно записать это структурное правило как

$$stmt \to \textbf{if} \ (\ expr \ ) \ stmt \ \textbf{else} \ stmt$$

Здесь символ "→" можно прочесть как "может иметь вид". Такое правило называется *продукцией*[1] (production). В продукции лексические элементы наподобие ключевого слова **if** и скобок называются *терминалами* (terminal). Переменные наподобие *expr* и *stmt* представляют последовательности терминалов и называются *нетерминалами* (nonterminals).

2.2.1 Определения грамматик

Контекстно-свободная грамматика имеет четыре компонента.

1. Множество *терминальных* символов, иногда именуемых токенами. Терминалы представляют собой элементарные символы языка, определяемые грамматикой.

2. Множество *нетерминалов*, иногда называемых синтаксическими переменными. Каждый нетерминал представляет множество строк терминалов способом, который будет описан далее.

3. Множество *продукций*, каждая из которых состоит из нетерминала, называемого *заголовком* или *левой частью* продукции, стрелки и последовательности терминалов и/или нетерминалов, называемых *телом* или *правой частью* продукции. Интуитивно назначение продукции — определить один из возможных видов конструкции; если заглавный нетерминал представляет конструкцию, то тело представляет записываемый вид конструкции.

4. Один из нетерминальных символов, указываемый как *стартовый* или *начальный*.

Грамматика определяется перечислением ее продукций, причем первой указывается продукция для стартового символа. Цифры, знаки наподобие < или <= и выделенные полужирным шрифтом строки наподобие **while** являются терминальными символами. Выделенные курсивом имена являются нетерминалами,

[1]Возможно, это не самый удачный перевод данного термина, однако он традиционно используется в русскоязычной компьютерной литературе (см., например, В.Дж. Рейуорд-Смит. *Теория формальных языков*. — М.: Радио и связь, 1988). — *Прим. пер.*

Токены и терминалы

Лексический анализатор компилятора считывает символы исходной программы, группирует их в лексически осмысленные единицы, которые называются лексемами, и в качестве выходных данных возвращает токены, представляющие эти лексемы. Токен состоит из двух компонентов — имени токена и значения атрибута. Имена токенов — это абстрактные символы, используемые синтаксическим анализатором при выполнении анализа. Зачастую мы будем называть эти имена токенов *терминалами*, поскольку они появляются в грамматике языка программирования в виде терминальных символов. Значение атрибута, если таковое имеется, представляет собой указатель на запись в таблице символов, в которой содержится дополнительная информация о токене. Эта дополнительная информация не является частью грамматики, так что в нашем рассмотрении синтаксического анализа мы будем считать токены и терминалы синонимами.

а все имена или символы без выделения могут рассматриваться как терминалы.[2] Для удобства записи правые части продукций с одними и теми же нетерминалами слева могут быть сгруппированы с помощью символа | ("или").

Пример 2.1. В примерах этой главы используются выражения, состоящие из цифр и знаков "плюс" и "минус", например 9-5+2, 3-1 или 7. Поскольку знаки "плюс" и "минус" должны располагаться между двумя цифрами, такие выражения можно рассматривать как списки цифр, разделенных знаками "плюс" и "минус". Синтаксис используемых выражений описывает грамматика из следующих продукций:

$$list \rightarrow list + digit \quad (2.1)$$
$$list \rightarrow list - digit \quad (2.2)$$
$$list \rightarrow digit \quad (2.3)$$
$$digit \rightarrow \mathtt{0} \mid \mathtt{1} \mid \mathtt{2} \mid \mathtt{3} \mid \mathtt{4} \mid \mathtt{5} \mid \mathtt{6} \mid \mathtt{7} \mid \mathtt{8} \mid \mathtt{9} \quad (2.4)$$

Тела трех продукций с нетерминалом *list* в качестве заголовка могут быть сгруппированы:

$$list \rightarrow list + digit \mid list - digit \mid digit$$

[2]Отдельные символы, выделенные курсивом, будут использоваться и для других целей при детальном изучении грамматики в главе 4. Например, мы будем использовать X, Y и Z при указании символов, которые могут представлять собой как терминалы, так и нетерминалы. Однако выделенное курсивом имя из двух или более символов будет всегда означать нетерминал.

В соответствии с нашими соглашениями терминалами грамматики являются символы

`+ - 0 1 2 3 4 5 6 7 8 9`

Нетерминальными символами являются выделенные курсивом имена *list* и *digit*, при этом нетерминал *list* — стартовый; именно его продукция дана первой. □

Продукция называется *продукцией нетерминала* (for nonterminal), если ее заголовок представляет собой нетерминал. Строка терминалов является последовательностью из нуля или нескольких терминалов. Строка, содержащая нуль терминалов и записываемая как ϵ, называется *пустой* строкой.[3]

2.2.2 Выведение

Грамматика *выводит*, или *порождает* (derive), строки, начиная со стартового символа и неоднократно замещая нетерминалы телами продукций этих нетерминалов. Строки токенов, порождаемые из стартового символа, образуют *язык*, определяемый грамматикой.

Пример 2.2. Язык, определяемый грамматикой из примера 2.1, состоит из списков цифр, разделенных знаками "плюс" и "минус". Десять продукций для нетерминала *digit* позволяют ему быть любым из терминалов `0`, `1`, ..., `9`. Из продукции (2.3) следует, что цифра сама по себе является списком. Продукции (2.1) и (2.2) выражают то правило, что любой список, за которым следует знак "плюс" или "минус" с последующей цифрой, образует новый список.

Продукции (2.1)–(2.4) — это все, что нужно для определения требующегося языка. Например, можно вывести, что `9-5+2` является списком, следующим образом.

а) `9` — *list* в соответствии с продукцией (2.3), поскольку `9` — *digit*.

б) `9-5` — *list* в соответствии с продукцией (2.2), так как `9` — *list*, а `5` — *digit*.

в) `9-5+2` — *list* в соответствии с продукцией (2.1), поскольку `9-5` — *list*, а `2` — *digit*. □

Пример 2.3. Несколько отличающимся видом списков является список параметров в вызове функции. В Java параметры находятся внутри круглых скобок; так, `max(x,y)` означает вызов функции `max` с параметрами `x` и `y`. Единственный нюанс в таком списке заключается в том, что между терминалами `(` и `)` может находиться пустой список. Мы можем начать разработку грамматики для таких последовательностей с продукций

[3]Технически ϵ может быть строкой из нуля символов любого алфавита (коллекции символов).

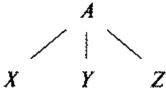
$$
\begin{array}{rcl}
call & \rightarrow & \textbf{id} \; (\; optparams \;) \\
optparams & \rightarrow & params \mid \epsilon \\
params & \rightarrow & params \; , \; param \mid param
\end{array}
$$

Обратите внимание, что второе возможное тело для *optparams* (optional parameter list — необязательный список параметров) представляет собой ϵ, что означает пустую строку символов. То есть *optparams* можно заменить пустой строкой, так что *call* может состоять из имени функции, за которым следует двухтерминальная строка `()`. Заметим, что продукции для *params* аналогичны продукциям для *list* из примера 2.1, если заменить арифметические операторы + и − запятой, а *digit* заменить на *param*. Мы не приводим продукции для *param*, поскольку параметры функции могут в действительности быть произвольными выражениями. Вскоре мы рассмотрим продукции для различных языковых конструкций, таких как выражения, инструкции и т.п. □

Синтаксический анализ, или *разбор* (parsing), представляет собой выяснение для полученной строки терминалов способа ее вывода из стартового символа грамматики. Если строка не может быть выведена из стартового символа, синтаксический анализатор сообщает об ошибке в строке. Синтаксический анализ — одна из наиболее фундаментальных задач компиляции; основные методы синтаксического анализа рассматриваются в главе 4. В этой главе для простоты мы начнем с исходных программ наподобие `9-5+2`, в которых каждый символ является терминалом; в общем случае исходная программа содержит многосимвольные лексемы, группируемые лексическим анализатором в токены, первые компоненты которых представляют собой терминалы, обрабатываемые синтаксическим анализатором.

2.2.3 Деревья разбора

Дерево разбора (parse tree) наглядно показывает, как стартовый символ грамматики порождает строку языка. Если нетерминал A имеет продукцию $A \rightarrow XYZ$, то дерево разбора может иметь внутренний узел, помеченный как A, с тремя потомками, помеченными слева направо как X, Y и Z:



Формально для данной контекстно-свободной грамматики *дерево разбора* представляет собой дерево со следующими свойствами.

1. Корень дерева помечен стартовым символом.

2. Каждый лист помечен терминалом или ϵ.

Терминология, связанная с деревьями

Древовидные структуры данных встречаются в компиляции очень часто.

- Дерево состоит из одного или нескольких *узлов* (node). Узлы могут иметь *метки* (label), которые в данной книге обычно являются символами грамматики. При изображении деревьев мы зачастую представляем узлы только их метками.

- Ровно один узел является *корнем* (root). Все узлы, за исключением корня, имеют единственный *родительский узел* (или просто *родитель*) (parent); корень не имеет родительского узла. При изображении деревьев мы размещаем родительский узел для данного узла над ним и проводим ребро между ними. Корень, таким образом, является самым верхним узлом.

- Если узел N родительский для узла M, то M является *дочерним* (child) по отношению к N. Дочерние узлы одного узла называются *родственными* или *сестринскими* (sibling). Они упорядочены *слева направо*, и когда мы изображаем дерево, то упорядочиваем дочерние узлы данного узла соответствующим образом.

- Узел без дочерних узлов называется *листом* (leaf). Другие узлы — у которых имеется один или несколько дочерних — представляют собой *внутренние узлы* (interior node).

- *Потомком* (descendant) узла N является сам N, дочерние по отношению к N узлы, узлы, дочерние по отношению к дочерним узлам N, и так далее для любого количества уровней. Мы говорим, что узел N является *предком* (ancestor) узла M, если M — потомок N.

3. Каждый внутренний узел помечен нетерминалом.

4. Если A является нетерминалом и помечает некоторый внутренний узел, а $X_1, X_2, \ldots, X_n$ — метки его дочерних узлов слева направо, то должна существовать продукция $A \to X_1 X_2 \cdots X_n$. Здесь каждое из обозначений $X_1, X_2, \ldots, X_n$ представляет собой либо терминальный, либо нетерминальный символ. В качестве частного случая продукции $A \to \epsilon$ соответствует узел A с единственным дочерним узлом ϵ.

Пример 2.4. Вывод 9-5+2 в примере 2.2 проиллюстрирован деревом на рис. 2.5. Каждый узел дерева помечен символом грамматики. Внутренний узел и его до-

черние узлы соответствуют продукции: внутренний узел соответствует заголовку продукции, а дочерние узлы — телу продукции.

На рис. 2.5 корневой узел помечен как *list* — стартовый символ грамматики из примера 2.1. Дочерние узлы слева направо помечены как *list*, + и *digit*. Заметьте, что

$$list \rightarrow list + digit$$

является продукцией грамматики из примера 2.1. Левый дочерний узел корня аналогичен корню, только его дочерний узел помечен как - вместо +. Все три узла, помеченные как *digit*, имеют по одному дочернему узлу с метками-цифрами. □

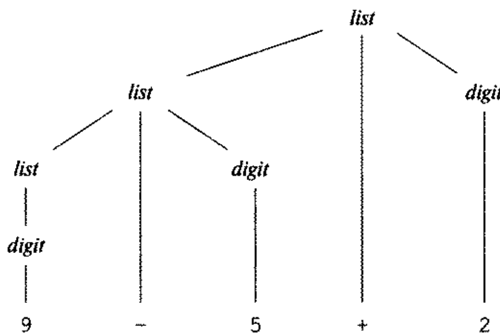


Рис. 2.5. Дерево разбора строки 9-5+2 в соответствии с грамматикой в примере 2.1

Листья дерева разбора слева направо образуют *крону* (yield) дерева, которая представляет собой строку, *выведенную* (derived), или *порожденную* (generated), из нетерминального символа в корне дерева разбора. На рис. 2.5 кроной является строка 9-5+2; для удобства все листья показаны на одном, нижнем, уровне. В дальнейшем выравнивать листья деревьев таким образом мы не будем. Любое дерево естественным образом упорядочивает свои листья слева направо, основываясь на том принципе, что если X и Y — два дочерних узла одного родителя и узел X находится слева от Y, то все потомки X будут находиться слева от любого потомка Y.

Другое определение языка, порожденного грамматикой, — это множество строк, которые могут быть сгенерированы некоторым деревом разбора. Процесс поиска дерева разбора для данной строки терминалов называется *разбором* (parsing) или *синтаксическим анализом* этой строки.

2.2.4 Неоднозначности

Следует быть предельно внимательным при рассмотрении структуры строки, соответствующей грамматике. Грамматика может иметь более одного дерева

разбора для данной строки терминалов. Такая грамматика называется *неоднозначной* (ambiguous). Чтобы показать неоднозначность грамматики, достаточно найти строку терминалов, которая дает более одного дерева разбора. Поскольку такая строка обычно имеет не единственный смысл, следует разрабатывать однозначные (непротиворечивые) грамматики либо неоднозначные грамматики с дополнительными правилами для разрешения неоднозначностей.

Пример 2.5. Предположим, мы используем только один нетерминал *string* и не различаем цифры и списки, как это делается в примере 2.1. Тогда грамматику можно записать следующим образом:

$$string \rightarrow string + string \mid string - string \mid 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$$

Объединение записей для *digit* и *list* в один нетерминальный символ *string* имеет кажущийся смысл, поскольку отдельная цифра является частным случаем списка.

Однако из рис. 2.6 видно, что, например, выражение 9-5+2 в такой грамматике имеет больше одного дерева разбора. Эти два дерева разбора соответствуют двум вариантам расстановки скобок в выражении: (9-5)+2 и 9-(5+2). Второе выражение дает в результате значение 2 вместо ожидаемого 6. Грамматика в примере 2.1 не допускает такой интерпретации. □

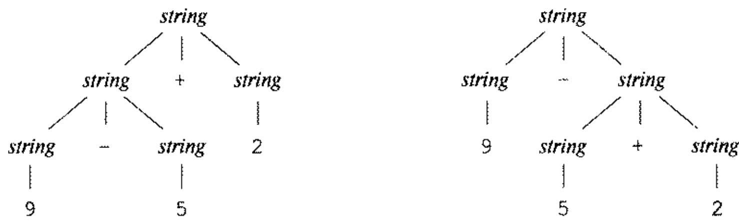


Рис. 2.6. Два дерева разбора для выражения 9-5+2

2.2.5 Ассоциативность операторов

По соглашению 9+5+2 эквивалентно (9+5)+2, а 9-5-2 эквивалентно (9-5)-2. Когда операнд типа 5 имеет операторы и слева, и справа, необходимы определенные соглашения, чтобы установить, какой именно оператор применяется к этому операнду. Мы говорим, что оператор + *левоассоциативен*, поскольку операнд со знаками "плюс" с обеих сторон используется левым оператором. В большинстве языков программирования четыре арифметических оператора — сложение, вычитание, умножение и деление — левоассоциативны.

Некоторые распространенные операторы, например возведение в степень, *правоассоциативны*. Другим примером правоассоциативного оператора может служить оператор присвоения (=) в С и его потомках; выражение `a=b=c` рассматривается как `a=(b=c)`.

Строки типа `a=b=c` с правоассоциативным оператором генерируются следующей грамматикой:

$$\begin{array}{lcl} \textit{right} & \rightarrow & \textit{letter} = \textit{right} \mid \textit{letter} \\ \textit{letter} & \rightarrow & \texttt{a} \mid \texttt{b} \mid \cdots \mid \texttt{z} \end{array}$$

Различия между деревьями разбора для левоассоциативных операторов наподобие - и правоассоциативных операторов наподобие = показаны на рис. 2.7. Обратите внимание, что дерево разбора для `9-5-2` растет вниз влево, в то время как дерево разбора для `a=b=c` — вниз вправо.

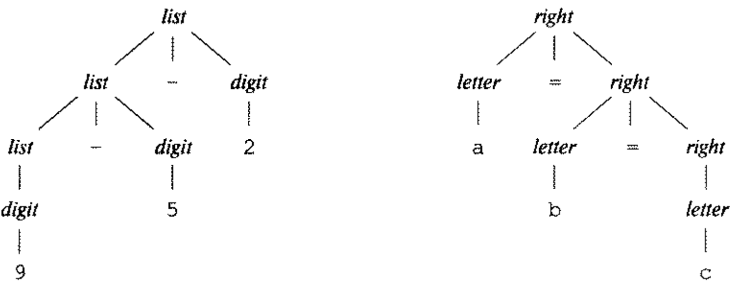


Рис. 2.7. Деревья разбора для лево- и правоассоциативных грамматик

2.2.6 Приоритет операторов

Рассмотрим выражение `9+5*2`. Есть два способа его интерпретации: как `(9+5)*2` и `9+(5*2)`. Ассоциативные правила для + и * применяются при наличии одного и того же оператора, так что они не в состоянии разрешить эту неоднозначность. Поэтому, если имеется более одного типа операторов, необходимы правила, определяющие относительный приоритет операторов.

Мы говорим, что оператор * имеет более *высокий приоритет*, чем +, если * получает свои операнды раньше +. В обычной арифметике умножение и деление имеют более высокий приоритет, чем сложение и вычитание. Таким образом, 5 является множителем в умножении в случае как с `9+5*2`, так и с `9*5+2`, т.е. эти выражения эквивалентны `9+(5*2)` и `(9*5)+2` соответственно.

Пример 2.6. Грамматика арифметических выражений может быть построена на основе таблицы, показывающей ассоциативность и приоритет операторов. Начнем с четырех основных арифметических операторов и таблицы приоритетов,

представляющей операторы в порядке увеличения приоритетов (операторы, расположенные в одной строке, имеют одинаковые приоритеты):

Левоассоциативные: + −

Левоассоциативные: * /

Создадим два нетерминала — *expr* и *term* — для этих уровней приоритета и дополнительный нетерминал *factor* для генерации базовых составляющих в выражениях, в данном случае — цифр и выражений в скобках:

$$factor \;\rightarrow\; \textbf{digit} \;|\; (\; expr\;)$$

Теперь рассмотрим бинарные операторы * и /, которые имеют наивысший приоритет. Поскольку эти операторы левоассоциативны, продукции подобны продукциям для списков, которые также левоассоциативны:

$$\begin{array}{lcl} term & \rightarrow & term * factor \\ & | & term\; /\; factor \\ & | & factor \end{array}$$

Точно так же *expr* рассматривается как список элементов *term*, разделенных операторами сложения или умножения:

$$\begin{array}{lcl} expr & \rightarrow & expr + term \\ & | & expr - term \\ & | & term \end{array}$$

В результате грамматика приобретает следующий вид:

$$\begin{array}{rcl} expr & \rightarrow & expr + term \;|\; expr - term \;|\; term \\ term & \rightarrow & term * factor \;|\; term\; /\; factor \;|\; factor \\ factor & \rightarrow & \textbf{digit} \;|\; (\; expr\;) \end{array}$$

Эта грамматика рассматривает выражение как список элементов, разделенных знаками + и −. При этом каждый элемент списка представляет собой список множителей, разделенных знаками * и /. Обратите внимание, что любое выражение в скобках является множителем, поэтому с помощью скобок можно создать выражение с любым уровнем вложенности (и, соответственно, произвольной высотой дерева). □

Пример 2.7. Ключевые слова позволяют распознавать инструкции, поскольку большинство из них начинается с ключевого слова или специального символа.

Обобщение грамматики выражений из примера 2.6

Можно рассматривать множитель (*factor*) как выражение, которое не может быть "разорвано" никаким оператором. Под этим мы понимаем то, что размещение оператора с любой стороны от множителя не может привести к тому, что операндом этого оператора может стать часть множителя – таким операндом может быть только оператор целиком. Если множителем является выражение в скобках, то от разрыва его защищают скобки; если множитель представляет собой отдельный операнд, то понятно, что он также не может быть разорван.

Член (*term*), не являющийся одновременно множителем, представляет собой выражение, которое может быть разорвано оператором с более высоким приоритетом, но не с менее высоким. Выражение, не являющееся ни членом, ни множителем, может быть разорвано любым оператором.

Эту идею можно обобщить для n уровней приоритетов. Нам нужно $n + 1$ нетерминалов. Первый, подобно *factor* в примере 2.6, не может быть разобран ни в коем случае. Обычно тела продукций для этого нетерминала являются отдельными операндами или выражениями в скобках. Затем, для каждого уровня приоритетов имеется по одному нетерминалу, представляющему выражения, которые могут быть разорваны операторами данного или более высокого уровня. Обычно продукции для этих нетерминалов имеют тела, представляющие использование операторов на данном уровне приоритета, плюс одно тело, которое представляет собой нетерминал для следующего по высоте уровня приоритета.

Исключениями из этого правила являются присваивания и вызовы процедур. Инструкции, определяемые (неоднозначной) грамматикой, показанной на рис. 2.8, являются корректными инструкциями Java.

В первой продукции для *stmt* терминал **id** представляет любой идентификатор. Продукции для *expression* не показаны. Инструкции присваивания, определяемые первой продукцией, являются корректными инструкциями Java, хотя Java рассматривает = как оператор присваивания, который может находиться внутри выражения. Например, Java разрешает использование присваивания `a=b=c`, запрещенное данной грамматикой.

Нетерминал *stmts* генерирует (возможно, пустой) список инструкций. Вторая продукция для *stmts* генерирует пустой список ϵ. Первая продукция генерирует (возможно, пустой) список инструкций, за которым следует инструкция.

Размещение точек с запятой – достаточно тонкий вопрос. Они располагаются в конце каждого тела продукции, которое не заканчивается *stmt*. Подобный под-

$$
\begin{array}{rcl}
stmt & \rightarrow & \textbf{id} = expression\ ; \\
& | & \textbf{if}\ (\ expression\ )\ stmt \\
& | & \textbf{if}\ (\ expression\ )\ stmt\ \textbf{else}\ stmt \\
& | & \textbf{while}\ (\ expression\ )\ stmt \\
& | & \textbf{do}\ stmt\ \textbf{while}\ (\ expression\ )\ ; \\
& | & \{\ stmts\ \} \\
\\
stmts & \rightarrow & stmts\ stmt \\
& | & \epsilon
\end{array}
$$

Рис. 2.8. Грамматика для подмножества инструкций Java

ход предотвращает накопление точек с запятой после таких инструкций, как **if** и **while**, которые заканчиваются вложенными подынструкциями. Когда вложенная инструкция представляет собой присваивание или цикл **do-while**, точка с запятой генерируется как часть подынструкции. □

2.2.7 Упражнения к разделу 2.2

Упражнение 2.2.1. Рассмотрим контекстно-свободную грамматику

$$S \rightarrow S\,S\,+ \mid S\,S\,* \mid \mathbf{a}$$

а) Покажите, как данная грамматика генерирует строку `aa+a*`.

б) Постройте дерево разбора для данной строки.

в) Какой язык генерирует данная грамматика? Обоснуйте свой ответ.

Упражнение 2.2.2. Какой язык генерируется каждой из следующих грамматик? В каждом случае обоснуйте свой ответ.

а) $S \rightarrow 0\,S\,1 \mid 0\,1$

б) $S \rightarrow +\,S\,S \mid -\,S\,S \mid \mathbf{a}$

в) $S \rightarrow S\ (\ S\ )\ S \mid \epsilon$

г) $S \rightarrow \mathbf{a}\,S\,\mathbf{b}\,S \mid \mathbf{b}\,S\,\mathbf{a}\,S \mid \epsilon$

д) $S \rightarrow \mathbf{a} \mid S + S \mid S\,S \mid S\,* \mid (\ S\ )$

Упражнение 2.2.3. Какие из грамматик в упражнении 2.2.2 неоднозначны?

Упражнение 2.2.4. Постройте однозначные контекстно-свободные грамматики для каждого из следующих языков. В каждом случае покажите корректность вашей грамматики.

а) Арифметические выражения в постфиксной записи.

б) Левоассоциативный список идентификаторов, разделенных запятыми.

в) Правоассоциативный список идентификаторов, разделенных запятыми.

г) Арифметические выражения, состоящие из целых чисел и идентификаторов с четырьмя бинарными операторами +, −, *, /.

! д) Добавьте унарные "плюс" и "минус" к арифметическим операторам из *г*.

Упражнение 2.2.5.

а) Покажите, что все бинарные строки, генерируемые приведенной далее грамматикой, имеют значения, делящиеся на 3. *Указание*: воспользуйтесь индукцией по количеству узлов в дереве разбора.

$$num \rightarrow \texttt{11} \mid \texttt{1001} \mid num\ \texttt{0} \mid num\ num$$

б) Генерирует ли эта грамматика все бинарные строки, значения которых делятся на 3?

Упражнение 2.2.6. Постройте контекстно-свободную грамматику для записи чисел римскими цифрами.

2.3 Синтаксически управляемая трансляция

Синтаксически управляемая трансляция выполняется путем присоединения правил или программных фрагментов к продукциям грамматики. Рассмотрим, например, выражение *expr*, генерируемое продукцией

$$expr \rightarrow expr_1 + term$$

Здесь *expr* представляет собой сумму подвыражений $expr_1$ и *term*. (Нижний индекс в $expr_1$ используется только для того, чтобы различать *expr* в теле продукции и в ее заголовке.) Можно транслировать *expr*, воспользовавшись его структурой, как показано в следующем псевдокоде:

Трансляция $expr_1$;

Трансляция *term*;

Обработка +;

Используя вариант такого псевдокода и обрабатывая + путем создания соответствующего узла, в разделе 2.8 мы построим синтаксическое дерево для *expr* путем построения синтаксических деревьев для $expr_1$ и *term*. Для удобства в качестве примера в этом разделе будет рассмотрена трансляция инфиксных выражений в постфиксные.

В этом разделе вводятся две концепции, связанные с синтаксически управляемой трансляцией.

- *Атрибуты.* Атрибут представляет собой некоторую величину, связанную с программной конструкцией. Примерами атрибутов являются типы данных выражений, количество команд в генерируемом коде, расположение первой команды в генерируемом для конструкции коде, а также многое другое. Поскольку мы используем грамматические символы (нетерминалы и терминалы) для представления программных конструкций, следует распространить понятие атрибутов не только на конструкции, но и на символы, их представляющие.

- (*Синтаксически управляемые*) *схемы трансляции.* Схема трансляции — это запись присоединенных к продукциям грамматики программных фрагментов. Эти фрагменты выполняются при использовании продукции в процессе синтаксического анализа. Объединенный результат выполнения всех этих фрагментов в порядке, определяемом синтаксическим анализом, и есть трансляция программы, к которой применяется этот процесс анализа/синтеза.

Синтаксически управляемая трансляция будет использоваться во всей этой главе для трансляции инфиксных выражений в постфиксные, для вычисления выражений и для построения синтаксических деревьев для программных конструкций. Более подробно синтаксически управляемый формализм рассматривается в главе 5.

2.3.1 Постфиксная запись

Примеры в этом разделе посвящены трансляции в постфиксную запись. *Постфиксная запись* (postfix notation) выражения E может быть индуктивно определена следующим образом.

1. Если E является переменной или константой, то постфиксная запись E представляет собой само E.

2. Если E — выражение вида E_1 **op** E_2, где **op** — произвольный бинарный оператор, то постфиксная запись E представляет собой $E'_1\, E'_2$ **op**, где E'_1 и E'_2 — постфиксные записи для E_1 и E_2 соответственно.

3. Если E — выражение в скобках вида (E_1), то постфиксная запись для E такова же, как и постфиксная запись для E_1.

Пример 2.8. Постфиксная запись для (9-5)+2 представляет собой 95-2+. Согласно правилу (1) трансляция 9, 5 и 2 представляет собой сами эти константы. По правилу (2) 9-5 транслируется в 95-. Тот же вид согласно правилу (3) имеет и трансляция (9-5). Получив результат трансляции подвыражения в скобках, можно применить правило (2) для трансляции всего выражения, где (9-5) выступает в роли E_1, а 2 — в роли E_2, и получить результат 95-2+.

В качестве другого примера для 9-(5+2) постфиксная запись имеет вид 952+-, т.е. 5+2 сначала транслируется в 52+, а затем это выражение становится вторым аргументом знака "минус". □

Скобки в постфиксной записи не используются, поскольку последовательность и *арность* (arity) — количество аргументов — операторов допускают только единственный способ декодирования постфиксного выражения. "Фокус" заключается в последовательном сканировании постфиксной строки слева направо, пока не встретится оператор. Затем выполняется поиск слева соответствующего количества операндов и найденный оператор выполняется с этими операндами. Результат выполнения замещает операнды и оператор, после чего процесс поиска слева направо очередного оператора продолжается.

Пример 2.9. Рассмотрим постфиксное выражение 952+-3*. Сканируя слева направо, первым мы встретим знак "плюс". Слева от него мы обнаруживаем операнды 5 и 2. Их сумма, 7, заменяет 52+, и мы получаем строку 97-3*. Теперь крайний слева оператор — знак "минус", а его операндами являются 9 и 7. Заменяя их результатом вычитания, мы получаем строку 23*. Символ умножения применяется к 2 и 3 и дает окончательный результат, равный 6. □

2.3.2 Синтезированные атрибуты

Идея назначения значений программным конструкциям — например, значений и типов выражений — может быть выражена в терминах грамматики. Мы назначаем атрибуты терминалам и нетерминалам, затем присоединяем к продукциям грамматики правила, которые описывают, каким образом происходит вычисление атрибутов в узлах дерева разбора, где рассматриваемая продукция используется для связи узла с дочерними узлами.

Синтаксически управляемое определение связывается

1) с каждым грамматическим символом, множеством атрибутов;

2) с каждой продукцией, множеством *семантических правил* для вычисления значений атрибутов, связанных с символами продукции.

Атрибуты могут вычисляться следующим образом. Для данной входной строки x строится дерево разбора для x. Затем для вычисления атрибутов в каждом узле дерева разбора применяются семантические правила описанным далее образом.

Предположим, что узел N в дереве разбора помечен грамматическим символом X. Обозначим как $X.a$ значение атрибута a символа X в этом узле. Дерево разбора с указанием значений атрибутов в каждом узле называется *аннотированным* (annotated) деревом разбора. Например, на рис. 2.9 показано аннотированное дерево разбора для 9-5+2 с атрибутом *t*, связанным с нетерминалами *expr* и *term*. Значение атрибута корня дерева 95-2+ представляет собой постфиксную запись для 9-5+2. Вскоре мы увидим, как вычисляются эти выражения.

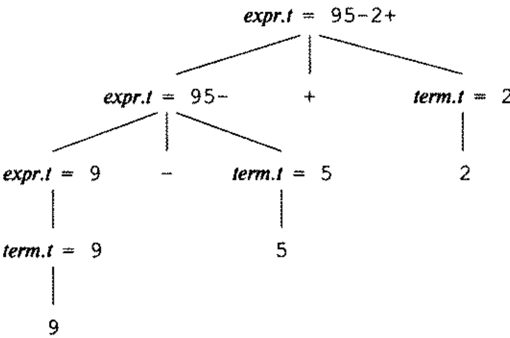


Рис. 2.9. Значения атрибутов в узлах дерева разбора

Атрибут называется *синтезированным*, если его значение в узле дерева разбора N определяется на основании атрибутов дочерних по отношению к N узлов и самого узла N. Синтезированный атрибут обладает тем требующимся свойством, что он может быть вычислен путем единственного восходящего прохода по дереву разбора. В разделе 5.1.1 мы рассмотрим другой важный тип атрибутов — наследуемые атрибуты. Говоря неформально, значения наследуемых атрибутов вычисляются на основании значений атрибутов в самом узле, родительском и сестринских узлах в дереве разбора.

Пример 2.10. Аннотированное дерево разбора на рис. 2.9 основано на синтаксически управляемых определениях для трансляции выражений, состоящих из цифр, разделенных знаками "плюс" и "минус", в постфиксную запись. Каждый нетерминал имеет строковый атрибут *t*, который представляет постфиксную запись для выражения, генерируемого этим нетерминалом в дереве разбора. Символ $\|$ в семантическом правиле представляет собой оператор конкатенации строк.

Постфиксная форма цифры — это сама цифра; например, семантическое правило продукции *term* → 9 определяет, что при ее использовании *term.t* в узле дерева

Продукция	Семантическое правило
$expr \rightarrow expr_1 + term$	$expr.t = expr_1.t \,\|\, term.t \,\|\, '+'$
$expr \rightarrow expr_1 - term$	$expr.t = expr_1.t \,\|\, term.t \,\|\, '-'$
$expr \rightarrow term$	$expr.t = term.t$
$term \rightarrow 0$	$term.t = '0'$
$term \rightarrow 1$	$term.t = '1'$
...	...
$term \rightarrow 9$	$term.t = '9'$

Рис. 2.10. Синтаксически управляемые определения для трансляции инфиксных выражений в постфиксные

разбора представляет собой просто 9. Все остальные цифры транслируются аналогично. В качестве другого примера при использовании продукции $expr \rightarrow term$ значение $term.t$ становится значением $expr.t$.

Продукция $expr \rightarrow expr_1 + term$ порождает выражение, содержащее оператор "плюс".[4] Левый операнд оператора сложения — $expr_1$, а правый — $term$. Семантическое правило данной продукции

$$expr.t = expr_1.t \,\|\, term.t \,\|\, '+'$$

определяет значение атрибута $expr.t$ путем конкатенации постфиксных форм $expr_1.t$ и $term.t$ левого и правого операндов, к которым затем добавляется знак "плюс". Это правило является формализацией определения постфиксного выражения. □

2.3.3 Простые синтаксически управляемые определения

Синтаксически управляемое определение в примере 2.10 имеет следующее важное свойство: строка, представляющая трансляцию нетерминала в заголовке каждой продукции, является конкатенацией трансляций нетерминалов в теле продукции в том же порядке, в котором они встречаются в продукции, с небольшими необязательными дополнительными строками. Синтаксически управляемое определение с такими свойствами называется *простым*.

[4] В этом и многих других правилах один и тот же нетерминал (здесь — $expr$) встречается несколько раз. Цель нижнего индекса 1 в $expr_1$ — помочь отличить два появления $expr$ в продукции; "1" не является частью нетерминала (см. врезку "Соглашения для различения используемых нетерминалов").

Соглашения для различения используемых нетерминалов

В правилах нам часто необходимо различать несколько использований одного и того же нетерминала в заголовке и в теле продукции (см. пример 2.10). Причина заключается в том, что в дереве разбора различные узлы, помеченные одним и тем же нетерминалом, обычно имеют разные значения для трансляции. Мы примем следующее соглашение: нетерминалы используются в заголовке продукции без индексов, а в теле продукции — с различными индексами. Это все один и тот же нетерминал, и индекс не является частью его имени. Однако вы должны понимать разницу между примерами конкретной трансляции, в которых используется указанное соглашение, и обобщенными продукциями наподобие $A \to X_1, X_2, \ldots, X_n$, где символы X с индексами представляют произвольный список грамматических символов, а *не* экземпляров одного конкретного нетерминала с именем X.

Пример 2.11. Рассмотрим первую продукцию и семантическое правило, представленные на рис. 2.10:

Продукция	Семантическое правило
$expr \to expr_1 + term$	$expr.t = expr_1.t \,\|\|\, term.t \,\|\|\, '+'$

(2.5)

Здесь трансляция *expr.t* представляет собой конкатенацию трансляций $expr_1$ и *term*, за которыми следует символ +. Обратите внимание, что $expr_1$ и *term* находятся в одном и том же порядке и в продукции, и в семантическом правиле. Перед и между их трансляциями нет никаких дополнительных символов; в данном примере имеется только один дополнительный символ за ними. □

При рассмотрении схем трансляции мы увидим, что простое синтаксически управляемое определение может быть реализовано путем печати только дополнительных строк в порядке их появления в определении.

2.3.4 Обходы дерева

Обходы дерева будут использоваться для описания вычисления атрибутов и для указания выполнения фрагментов кода в схеме трансляции. *Обход* (traversal) дерева начинается с его корня и посещает каждый узел дерева в некотором порядке.

Обход *в глубину* (depth-first) начинается с корня и рекурсивно посещает дочерние узлы каждого узла в любом порядке, не обязательно слева направо. Такой обход называется обходом “в глубину”, так как он сперва посещает все непосе-

щенные дочерние узлы, насколько это возможно, т.е. посещает сначала все узлы, находящиеся на наибольшем удалении ("глубине") от корневого.

Процедура $visit\,(N)$ на рис. 2.11 представляет собой обход в глубину, посещающий дочерние узлы в порядке слева направо, как показано на рис. 2.12. При таком обходе действия по вычислению трансляции в каждом узле должны включаться непосредственно перед тем, как будет покинут данный узел (т.е. после того, как будут корректно вычислены трансляции в дочерних узлах). В общем случае действия, выполняемые обходом, могут быть любыми, в том числе отсутствовать вообще.

```
procedure visit(node N) {
    for ( каждый дочерний узел C узла N
          в порядке слева направо ) {
              visit(C);
    }
    Вычислить семантические правила в узле N;
}
```

Рис. 2.11. Рис. 2.11. Обход дерева в глубину

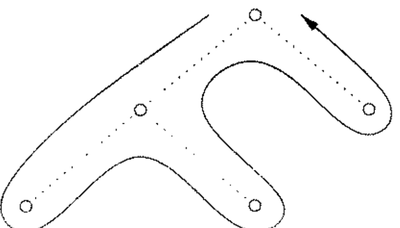


Рис. 2.12. Пример обхода дерева в глубину

Синтаксически управляемое определение не задает конкретный порядок вычисления атрибутов в дереве разбора. Пригоден любой порядок, который вычисляет атрибут a после всех атрибутов, от которых зависит a. Синтезированные атрибуты могут быть вычислены при любом *восходящем* (bottom-up) обходе дерева, т.е. обходе, который вычисляет атрибуты узла после вычисления атрибутов дочерних узлов. В общем случае при наличии и синтезированных, и наследуемых атрибутов вопрос порядка вычисления становится достаточно сложным; см. раздел 5.2.

Обходы в прямом и обратном порядке

Обходы дерева в прямом и обратном порядке — это два важных частных случая обхода в глубину, при которых мы посещаем дочерние узлы каждого узла слева направо.

Зачастую дерево обходится для выполнения некоторых определенных действий в каждом узле. Если действие выполняется, когда мы впервые попадаем в узел, то такой обход можно назвать *обходом в прямом порядке* (preorder). Аналогично, если действие выполняется непосредственно перед тем, как узел покидается, такой обход называется *обходом в обратном порядке* (postorder). Процедура $visit(N)$ на рис. 2.11 представляет собой пример обхода в обратном порядке.

Обходы в прямом и обратном порядке определяют соответствующие упорядочения узлов, основанные на том, когда выполняются действия в узлах. *Прямой порядок* (под)дерева с корнем N представляет собой N, за которым следуют прямые порядки поддеревьев всех его дочерних узлов (если таковые имеются) слева направо. *Обратный порядок* (под)дерева с корнем N представляет собой обратные порядки поддеревьев всех его дочерних узлов (если таковые имеются) слева направо, за которыми следует узел N.

2.3.5 Схемы трансляции

Синтаксически управляемое определение на рис. 2.10 строит трансляцию путем присоединения в качестве атрибутов строк к узлам дерева разбора. Теперь мы рассмотрим альтернативный подход, который не требует работы со строками; он инкрементно создает ту же самую трансляцию путем выполнения программных фрагментов.

Схема синтаксически управляемой трансляции представляет собой запись для определения конкретной трансляции путем присоединения программных фрагментов к продукциям грамматики. Схема трансляции похожа на синтаксически управляемое определение, с тем отличием, что явно определен порядок вычисления семантических правил.

Программные фрагменты, вставленные в тела продукций, называются *семантическими действиями*. Позиция выполняемого действия указывается фигурными скобками в теле продукции, например

$$rest \;\rightarrow\; +\; term \;\; \{\text{print}('+')\} \;\; rest_1$$

Мы встретимся с такими правилами при рассмотрении альтернативных видов грамматик для выражений, где нетерминал *rest* представляет "все, кроме первого члена выражения". Такой вид грамматики будет рассматриваться в разделе 2.4.5.

Здесь нижний индекс у $rest_1$ так же, как и ранее, присутствует для того, чтобы отличить нетерминал *rest* в теле продукции от экземпляра *rest* в ее заголовке.

При изображении дерева разбора для схемы трансляции мы указываем действие путем добавления дополнительного дочернего узла и проведения от него пунктирной линии к узлу, соответствующему заголовку продукции. Например, часть дерева разбора для приведенных выше продукции и действия показана на рис. 2.13. Узел семантического действия не имеет дочерних узлов, так что действие выполняется при первом посещении узла.

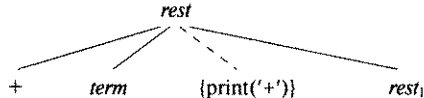


Рис. 2.13. Для семантического действия создается дополнительный лист

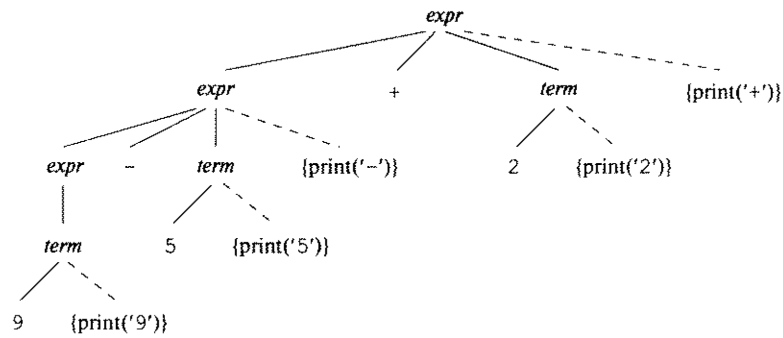


Рис. 2.14. Действия при трансляции 9-5+2 в 95-2+

$$
\begin{array}{lcll}
expr & \rightarrow & expr_1 + term & \{\text{print}('+')\} \\
expr & \rightarrow & expr_1 - term & \{\text{print}('-')\} \\
expr & \rightarrow & term & \\
term & \rightarrow & 0 & \{\text{print}('0')\} \\
term & \rightarrow & 1 & \{\text{print}('1')\} \\
 & & \dots & \\
term & \rightarrow & 9 & \{\text{print}('9')\}
\end{array}
$$

Рис. 2.15. Действия при трансляции в постфиксную запись

Пример 2.12. Дерево разбора на рис. 2.14 содержит инструкции печати в дополнительных листьях, которые присоединены пунктирными линиями ко внутренним узлам дерева разбора. Схема трансляции показана на рис. 2.15. Лежащая в ее основе грамматика генерирует выражения, состоящие из цифр, разделенных знаками "плюс" и "минус". Действия в телах продукций транслируют такие выражения в постфиксную запись при обходе дерева в глубину слева направо и выполнении каждой инструкции печати при посещении соответствующего листа.

Корень дерева на рис. 2.14 представляет первую продукцию на рис. 2.15. При обратном порядке обхода сначала выполняются все действия в крайнем слева поддереве корня для левого операнда, который, как и корень, помечен как *expr*. Затем посещается лист +, в котором не выполняются никакие действия, после чего выполняются действия в поддереве правого операнда *term* и, наконец, выполняется семантическое действие $\{\text{print}\,('+')\}$ в дополнительном узле.

Поскольку продукции для *term* в правой части имеют только одну цифру, эта цифра выводится действиями данных продукций. Не требуется никакой вывод в случае продукции $expr \to term$, а в действиях первых двух продукций выводится только соответствующий оператор. При обходе в обратном порядке выполняющиеся действия на рис. 2.14 выводят строку `95-2+`. □

Обратите внимание, что хотя схемы на рис. 2.10 и 2.15 дают одну и ту же трансляцию, их построение отличается. На рис. 2.10 узлам дерева разбора назначаются строковые атрибуты, в то время как схема на рис. 2.15 выводит трансляцию инкрементно, посредством семантических действий. Семантические действия в дереве разбора на рис. 2.14 транслируют инфиксное выражение `9-5+2` в `95-2+` путем вывода каждого символа ровно один раз, без привлечения дополнительной памяти для трансляции подвыражений. При инкрементном выводе описанным способом становится важен порядок, в котором выполняется вывод символов.

Реализация схемы трансляции должна гарантировать, что семантические действия выполняются в том же порядке, в котором они встречаются при обходе дерева разбора в обратном порядке. Реализация не обязана реально строить дерево разбора (зачастую оно и не строится), но должна гарантировать такое выполнение семантических действий, как если бы дерево разбора было построено, а затем выполнены действия при обходе этого дерева разбора в обратном порядке.

2.3.6 Упражнения к разделу 2.3

Упражнение 2.3.1. Постройте схему синтаксически управляемой трансляции, которая транслирует арифметические выражения из инфиксной записи в префиксную, в которой оператор находится перед операндами; например, $-xy$ представляет собой префиксную запись для $x - y$. Изобразите аннотированное дерево разбора для входных строк `9-5+2` и `9-5*2`.

Упражнение 2.3.2. Постройте схему синтаксически управляемой трансляции, которая транслирует арифметические выражения из постфиксной записи в инфиксную. Изобразите аннотированное дерево разбора для входных строк `95-2*` и `952*-`.

Упражнение 2.3.3. Постройте схему синтаксически управляемой трансляции, которая транслирует целые числа в числа, записанные римскими цифрами.

Упражнение 2.3.4. Постройте схему синтаксически управляемой трансляции, которая транслирует числа, записанные римскими цифрами, в обычную десятичную запись.

Упражнение 2.3.5. Постройте схему синтаксически управляемой трансляции, которая транслирует постфиксные арифметические выражения в эквивалентные префиксные арифметические выражения.

2.4 Разбор

Разбор (parsing) представляет собой процесс, определяющий, может ли некоторая строка терминалов быть сгенерирована данной грамматикой. При рассмотрении этой задачи полезно представлять ее как построение дерева разбора, хотя в действительности оно может и не создаваться компилятором. Однако анализатор должен быть способен построить такое дерево в принципе, иначе невозможно гарантировать корректность трансляции.

В этом разделе представлен метод разбора, называющийся "рекурсивным спуском", который может использоваться как для синтаксического анализа, так и для реализации синтаксически управляемых трансляторов. В следующем разделе приведена законченная программа на языке программирования Java, реализующая схему трансляции на рис. 2.15. Альтернативный способ состоит в использовании программного инструментария для генерации транслятора непосредственно на основе схемы трансляции. В разделе 4.9 описывается такой инструмент — Yacc; он может реализовать схему трансляции на рис. 2.15 без какой-либо модификации.

Для любой контекстно-свободной грамматики существует анализатор, который требует для разбора строки из n терминалов время, не превышающее $O\left(n^3\right)$. Однако, в целом, кубическое время слишком велико; к счастью, для реальных языков программирования можно разработать грамматику, обрабатываемую существенно быстрее. Для разбора почти всех встречающихся на практике языков достаточно линейного алгоритма. Анализаторы языков программирования почти всегда делают один проход входного потока слева направо, заглядывая вперед на один терминал, и по ходу просмотра строят части дерева разбора.

Большинство методов разбора делится на два класса — нисходящие (*сверху вниз*, top-down) и восходящие (*снизу вверх*, bottom-up). Эти термины связаны

с порядком, в котором строятся узлы дерева разбора. В нисходящих анализаторах построение начинается от корня по направлению к листьям, в то время как при восходящем методе — от листьев по направлению к корню. Популярность нисходящих анализаторов обусловлена тем фактом, что построить эффективный анализатор вручную проще с использованием нисходящих методов. Однако восходящий разбор может работать с бо́льшим классом грамматик и схем трансляции, так что программный инструментарий для генерации анализаторов непосредственно на основе грамматик часто использует восходящие методы.

2.4.1 Нисходящий анализ

Знакомство с нисходящим анализом начнем с рассмотрения грамматики, которая хорошо подходит для методов разбора этого типа. Позже в этом разделе будет рассмотрено построение нисходящих анализаторов в общем случае. Приведенная на рис. 2.16 грамматика генерирует подмножество инструкций C или Java. Мы используем выделенные полужирным шрифтом терминалы **if** и **for** для ключевых слов `if` и `for` соответственно, чтобы подчеркнуть, что эти последовательности символов рассматриваются как единицы, т.е. отдельные терминальные символы. Далее, терминал **expr** представляет выражения; более полная грамматика использовала бы нетерминал *expr* и содержала бы продукции для него. Аналогично **other** — терминал, представляющий другие конструкции языка.

$$
\begin{array}{rcl}
\mathit{stmt} & \rightarrow & \textbf{expr}\ ; \\
& | & \textbf{if}\ (\ \textbf{expr}\ )\ \mathit{stmt} \\
& | & \textbf{for}\ (\ \mathit{optexpr}\ ;\ \mathit{optexpr}\ ;\ \mathit{optexpr}\ )\ \mathit{stmt} \\
& | & \textbf{other} \\
\\
\mathit{optexpr} & \rightarrow & \epsilon \\
& | & \textbf{expr}
\end{array}
$$

Рис. 2.16. Грамматика для некоторых выражений C и Java

Построение дерева разбора (наподобие показанного на рис. 2.17) сверху вниз начинается с корня, помеченного стартовым нетерминалом *stmt*, и осуществляется многократным выполнением следующих двух шагов.

1. В узле N, помеченном нетерминалом A, выбираем одну из продукций для A и строим дочерние узлы N для символов из правой части продукции.

2. Находим следующий узел, в котором должно быть построено поддерево; обычно это крайний слева неразвернутый нетерминал дерева.

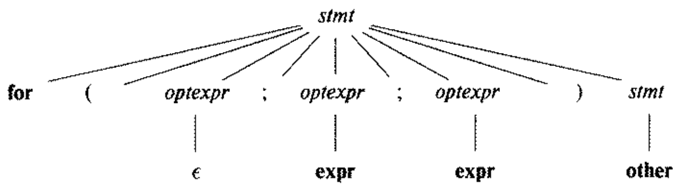


Рис. 2.17. Дерево разбора, соответствующее грамматике на рис. 2.16

Для некоторых грамматик описанные выше шаги могут быть реализованы в процессе единственного прохода слева направо по входной строке. Текущий сканируемый терминал входной строки часто называют *сканируемым символом* или "предсимволом" (lookahead symbol).[5] Изначально предсимволом является первый, т.е. крайний слева, терминал входной строки. На рис. 2.18 проиллюстрировано построение дерева разбора на рис. 2.17 для входной строки

for (; **expr** ; **expr**) **other**

Изначально предсимволом является терминал **for**, и известная часть дерева разбора состоит из корня, помеченного стартовым нетерминалом *stmt* на рис. 2.18, *а*. Цель заключается в построении оставшегося дерева разбора таким способом, чтобы строка, сгенерированная деревом разбора, соответствовала входной строке.

Для получения соответствия нетерминал *stmt* на рис. 2.18, *а* должен порождать строку, которая начинается с предсимвола **for**. В грамматике на рис. 2.16 имеется только одна продукция для *stmt*, которая может породить такую строку, так что мы выбираем ее и строим дочерние узлы корня, помеченные символами из тела этой продукции. Соответствующий рост дерева разбора показан на рис. 2.18, *б*.

На каждом из трех "снимков" дерева на рис. 2.18 стрелки указывают сканируемый символ входной строки и рассматриваемый узел дерева разбора. Как только у узла дерева создаются дочерние узлы, следует рассмотреть крайний слева узел. На рис. 2.18, *б* только что были созданы дочерние по отношению к корню узлы и рассматривается крайний слева узел, помеченный как **for**.

Если рассматриваемый узел дерева разбора представляет терминал и этот терминал соответствует сканируемому символу, мы перемещаемся как по дереву разбора, так и по входной строке. На рис. 2.18, *в* стрелка в дереве разбора перемещается к следующему дочернему узлу корня, а во входном потоке — к следующему терминалу, который в данном случае представляет собой (. Дальнейшее перемещение стрелки в дереве разбора приведет ее в дочерний узел, помеченный нетерминалом *optexpr*, а во входной строке — к терминалу ; .

[5]Дословно — "предвиденный символ". — *Прим. пер.*

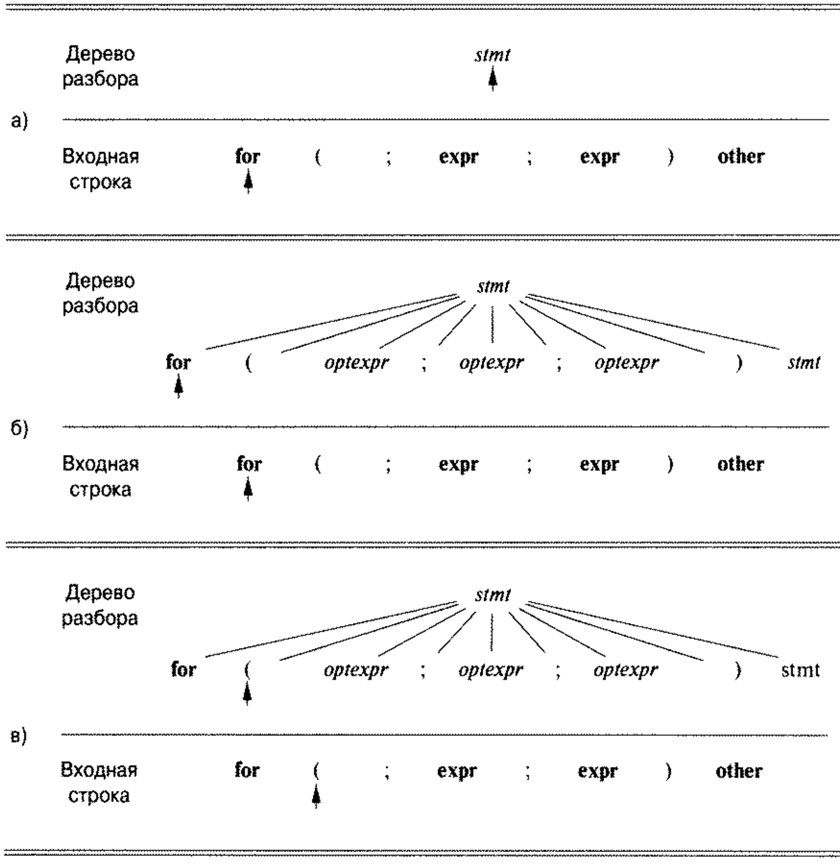


Рис. 2.18. Нисходящий разбор при сканировании слева направо

В нетерминальном узле, помеченном *optexpr*, мы повторяем процесс выбора продукции для нетерминала. Продукция, в теле которой находится ϵ ("ϵ-продукция") требует специального рассмотрения. В настоящее время мы используем ее по умолчанию, если ни одна другая продукция не может быть использована; мы вернемся к этому вопросу в разделе 2.4.3. При нетерминале *optexpr* и сканируемом символе *;* используется ϵ-продукция, поскольку *;* не соответствует единственной другой продукции для *optexpr*, в теле которой имеется терминал **expr**.

В общем случае выбор продукции для нетерминала может включать метод проб и ошибок, т.е. следует испытывать продукции, возвращаться и пробовать другие, если испытанная оказалась неподходящей. Продукция является неподходящей, если после ее использования не получается завершить дерево, соответствующее входной строке. Однако возврат оказывается излишним в важном частном случае предиктивного анализа, который будет рассматриваться в следующем разделе.

2.4.2 Предиктивный анализ

Анализ методом рекурсивного спуска (recursive-descent parsing) представляет собой способ нисходящего синтаксического анализа, при котором для обработки входной строки используется множество рекурсивных процедур (с каждым нетерминалом грамматики связана своя процедура). Здесь будет рассмотрен простой вид анализа методом рекурсивного спуска, именуемый *предиктивным* (или *предсказывающим*) *анализом* (predictive parsing), при котором сканируемый символ однозначно определяет поток управления в теле процедуры для каждого нетерминала. Последовательность вызовов процедур при обработке входной строки неявно определяет его дерево разбора и при необходимости может использоваться для его явного построения.

Предиктивный анализатор на рис. 2.19 состоит из процедур для нетерминалов *stmt* и *optexpr* грамматики на рис. 2.16 и дополнительной процедуры *match*, использующейся для упрощения кода для *stmt* и *optexpr*. Процедура *match* (t) сравнивает свой аргумент t со сканируемым символом и переходит к следующему символу в случае соответствия. Она изменяет значение глобальной переменной *lookahead*, которая хранит сканируемый входной терминал.

Анализ начинается с вызова процедуры для стартового нетерминала *stmt*. Для той же входной строки, что и на рис. 2.18, переменная *lookahead* изначально представляет собой первый терминал **for**. Процедура *stmt* выполняет код, соответствующий продукции

$$stmt \rightarrow \textbf{for}\ (\ optexpr\ ;\ optexpr\ ;\ optexpr\ )\ stmt$$

В коде для тела продукции — т.е. для **for** в процедуре *stmt* — каждый терминал проверяется на соответствие сканируемому символу, а каждый нетерминал приводит к вызову соответствующей процедуры путем следующей последовательности вызовов:

match(**for**); *match*('(');
optexpr(); *match*(';'); *optexpr*(); *match*(';'); *optexpr*();
match(')'); *stmt*();

Предиктивный анализ основан на информации о первых символах, которые могут быть сгенерированы телом продукции. Говоря более строго, пусть α — строка символов грамматики (терминалов и/или нетерминалов). Определим FIRST (α) как множество терминалов, которые могут появиться в качестве первого символа одной или нескольких строк, сгенерированных из α. Если α представляет собой ϵ или может порождать ϵ, то ϵ также входит в FIRST (α).

Детальное вычисление FIRST (α) приведено в разделе 4.4.2. Здесь же мы воспользуемся рассуждениями для вывода символов в FIRST (α) для частного случая.

```
void stmt() {
    switch ( lookahead ) {
    case expr:
        match(expr); match(';'); break;
    case if:
        match(if); match('('); match(expr); match(')'); stmt();
        break;
    case for:
        match(for); match('(');
        optexpr(); match(';'); optexpr(); match(';'); optexpr();
        match(')'); stmt(); break;
    case other;
        match(other); break;
    default:
        report("syntax error");
    }
}

void optexpr() {
    if ( lookahead == expr ) match(expr);
}

void match(terminal t) {
    if ( lookahead == t ) lookahead = nextTerminal;
    else report("syntax error");
}
```

Рис. 2.19. Псевдокод предиктивного анализатора

Обычно α начинается либо с терминала, который, таким образом, является единственным символом в FIRST (α), либо с нетерминала, тела продукций которого начинаются с терминалов (и в данном случае в FIRST (α) входят только эти терминалы).

Например, что касается грамматики на рис. 2.16, то вот примеры корректных вычислений FIRST (α):

$$\begin{array}{rcl} \text{FIRST}(\mathit{stmt}) & = & \{\textbf{expr}, \textbf{if}, \textbf{for}, \textbf{other}\} \\ \text{FIRST}(\textbf{expr}\ ;) & = & \{\textbf{expr}\} \end{array}$$

Множества FIRST должны рассматриваться, если существуют две продукции $A \to \alpha$ и $A \to \beta$. Игнорируя пока что ϵ-продукции, предиктивный анализатор

требует, чтобы множества FIRST (α) и FIRST (β) были непересекающимися. Тогда текущий сканируемый символ может использоваться для принятия решения, какую из продукций следует применить. Если сканируемый символ принадлежит множеству FIRST (α), используется продукция α; в противном случае, если сканируемый символ принадлежит множеству FIRST (β), применяется продукция β.

2.4.3 Использование пустых продукций

Наш предиктивный анализатор использует ϵ-продукцию в качестве продукции по умолчанию, когда не могут быть использованы никакие другие продукции. Для входной строки, приведенной на рис. 2.18, после соответствия терминалов **for** и (сканируемым символом становится ; . В этот момент вызывается процедура *optexpr*, а в ее теле выполняется код

if (*lookahead* == **expr**) *match*(**expr**);

Нетерминал *optexpr* имеет две продукции, с телами **expr** и ϵ. Сканируемый символ ; не соответствует терминалу **expr**, так что продукция с телом **expr** неприменима. Реально происходит возврат из процедуры без изменения текущего сканируемого символа или выполняются какие-то иные действия. Случай, когда процедура ничего не делает, соответствует применению ϵ-продукции.

Рассмотрим более общий случай продукции на рис. 2.16, где *optexpr* генерирует нетерминальное выражение вместо терминала **expr**:

$$\begin{array}{rcl} optexpr & \to & expr \\ & | & \epsilon \end{array}$$

Таким образом, *optexpr* генерирует либо выражение с использованием нетерминала *expr*, либо ϵ. При анализе *optexpr*, если сканируемый символ не принадлежит FIRST $(expr)$, используется ϵ-продукция.

Дополнительную информацию о том, когда применяются ϵ-продукции, можно почерпнуть из раздела 4.4.3, посвященного LL(1)-грамматикам.

2.4.4 Разработка предиктивного анализатора

Можно обобщить неформально описанные в разделе 2.4.2 методы для применения к любой грамматике, которая обладает непересекающимися множествами FIRST для тел продукций, принадлежащих нетерминалам. Мы также увидим, что если имеется схема трансляции — т.е. грамматика с внедренными действиями, — то эти действия можно выполнять как часть процедур, разработанных для анализатора.

Вспомним, что *предиктивный анализатор* представляет собой программу, состоящую из процедур для каждого нетерминала. Каждая такая процедура для нетерминала A решает две задачи.

1. Принимает решение, какая A-продукция будет использоваться, исходя из текущего сканируемого символа. Если сканируемый символ принадлежит множеству FIRST (α), применяется продукция с телом α (где α не является пустой строкой ϵ). В случае конфликта двух непустых тел для некоторого сканируемого символа этот метод анализа для рассматриваемой грамматики неприменим. Кроме того, ϵ-продукция для A, если таковая существует, используется в случае, когда сканируемый символ отсутствует в множестве FIRST для всех остальных тел продукций для A.

2. Процедура имитирует тело выбранной продукции, т.е. символы тела "выполняются" по очереди, слева направо. Нетерминал "выполняется" путем вызова процедуры, соответствующей этому нетерминалу, а терминал, соответствующий текущему сканируемому символу, — путем чтения следующего входного сканируемого символа. Если в какой-то момент терминал продукции не соответствует сканируемому символу, сообщается о наличии синтаксической ошибки.

На рис. 2.19 показан результат применения этих правил к грамматике на рис. 2.16.

Так же, как схема трансляции получается в результате расширения грамматики, синтаксически управляемый транслятор может быть получен расширением предиктивного анализатора. Алгоритм для решения этой задачи приведен в разделе 5.4. В настоящий момент достаточно следующего ограниченного построения.

1. Построить предиктивный анализатор, игнорируя действия в продукциях.

2. Скопировать действия из схемы трансляции в анализатор. Если действие в продукции p находится после символа грамматики X, то оно копируется после кода, реализующего X. В противном случае, если это действие располагается в начале продукции, оно копируется непосредственно перед кодом, реализующим продукцию.

Такой транслятор будет построен в разделе 2.5.

2.4.5 Левая рекурсия

Анализатор, работающий методом рекурсивного спуска, может оказаться в состоянии зацикливания. Такая проблема возникает при "леворекурсивных" продукциях наподобие

$$expr \;\rightarrow\; expr + term$$

В них левый символ тела продукции идентичен нетерминалу в заголовке продукции. Предположим, что процедура для *expr* принимает решение о применении

этой продукции. Поскольку тело продукции начинается с *expr*, рекурсивно вызывается та же самая процедура. Поскольку сканируемый символ изменяется только тогда, когда он соответствует терминалу в теле продукции, между рекурсивными вызовами *expr* не происходит никаких изменений. В результате второй вызов *expr* действует точно так же, как и первый, что означает выполнение третьего и прочих вызовов *expr* до бесконечности.

Левую рекурсию можно устранить, переписав некорректную продукцию. Рассмотрим нетерминал A с двумя продукциями:

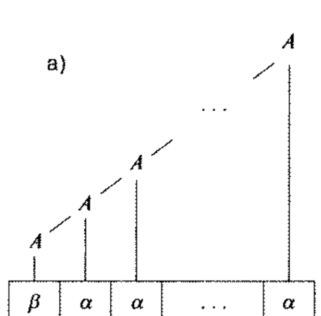
$$A \rightarrow A\alpha \mid \beta$$

Здесь α и β – последовательности терминалов и нетерминалов, которые не начинаются с A. Например, в

$$expr \rightarrow expr + term \mid term$$

нетерминал A представляет собой *expr*, строка $\alpha = +term$, а строка $\beta = term$.

Нетерминал A и его продукция называются *леворекурсивными*, потому что продукция $A \rightarrow A\alpha$ содержит A в качестве крайнего слева символа в теле продукции.[6] Повторное применение данной продукции приводит к последовательности α справа от A, как на рис. 2.20, *а*. Когда A, наконец, заменяется на β, за ним следует последовательность из нуля или большего количества α.



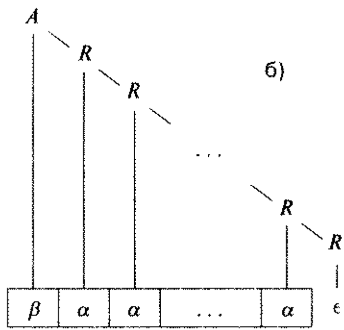


Рис. 2.20. Лево- и праворекурсивные способы генерации строки

Тот же результат может быть достигнут (рис. 2.20, *б*) путем переписывания продукции для A с использованием нового нетерминала R:

$$\begin{aligned} A &\rightarrow \beta R \\ R &\rightarrow \alpha R \mid \epsilon \end{aligned}$$

[6] В общем случае в леворекурсивной грамматике вместо продукции $A \rightarrow A\alpha$ нетерминал A может порождать $A\alpha$ через промежуточную продукцию.

Нетерминал R и его продукция $R \to \alpha R$ – праворекурсивные, поскольку продукция для R содержит в теле R в качестве крайнего справа символа. Праворекурсивные продукции приводят к деревьям, которые растут вниз вправо, как на рис. 2.20, *б*. Рост деревьев вниз вправо затрудняет трансляцию выражений, содержащих левоассоциативные операторы, такие как "минус". Однако в разделе 2.5.2 вы увидите, что корректная трансляция выражения в постфиксную запись все еще может быть получена путем аккуратного проектирования схемы трансляции.

В разделе 4.3.3 мы рассмотрим более общие виды левой рекурсии и покажем, как можно устранить левую рекурсию из грамматики.

2.4.6 Упражнения к разделу 2.4

Упражнение 2.4.1. Постройте анализатор, работающий методом рекурсивного спуска, для следующих грамматик.

а) $S \;\to\; + S\, S \mid - S\, S \mid$ **a**

б) $S \;\to\; S\ (\ S\ )\ S \mid \epsilon$

в) $S \;\to\; 0\, S\, 1 \mid 0\, 1$

2.5 Транслятор простых выражений

С помощью методов, описанных в трех последних разделах, мы построим синтаксически управляемый транслятор в виде рабочей программы на Java, которая будет транслировать арифметические выражения в постфиксную запись. Чтобы программа не была слишком больших размеров, ограничимся выражениями, состоящими из цифр, разделенных бинарными плюсами и минусами. В разделе 2.6 эта программа будет расширена таким образом, чтобы транслировать выражения, включающие числа и другие операторы. Поскольку выражения встречаются в качестве конструкций практически во всех языках, стоит детально изучить их трансляцию.

Синтаксически управляемая схема трансляции часто служит в качестве спецификации транслятора. Схема на рис. 2.21 (повторяющая схему на рис. 2.15) определяет выполняемую трансляцию.

Зачастую грамматика, лежащая в основе данной схемы, требует модификации перед тем, как сможет быть разобрана предиктивным анализатором. В частности, грамматика, лежащая в основе схемы, представленной на рис. 2.21, леворекурсивна, а, как вы видели в предыдущем разделе, предиктивный анализатор не может обработать леворекурсивную грамматику.

Мы оказываемся на распутье: с одной стороны, нам нужна грамматика, которая упрощает трансляцию, а с другой стороны, нам нужна совсем другая грамматика, которая упрощает синтаксический анализ. Решение заключается в том,

$$
\begin{array}{llll}
expr & \to & expr + term & \{ \text{print}('+') \} \\
 & | & expr - term & \{ \text{print}('-') \} \\
 & | & term & \\
 & & & \\
term & \to & 0 & \{ \text{print}('0') \} \\
 & | & 1 & \{ \text{print}('1') \} \\
 & & \dots & \\
 & | & 9 & \{ \text{print}('9') \}
\end{array}
$$

Рис. 2.21. Действия для трансляции в постфиксную запись

чтобы начать с грамматики с простой трансляцией и осторожно и аккуратно преобразовать ее в упрощающую синтаксический анализ. Путем устранения левой рекурсии из схемы на рис. 2.21 мы можем получить грамматику, подходящую для использования в предиктивном трансляторе, работающем методом рекурсивного спуска.

2.5.1 Абстрактный и конкретный синтаксис

Хорошим отправным пунктом для проектирования транслятора служит структура данных, которая называется абстрактным синтаксическим деревом или деревом абстрактного синтаксиса (abstract syntax tree). В *абстрактном синтаксическом дереве* для выражения каждый внутренний узел представляет оператор; его дочерние узлы представляют операнды этого оператора. В общем случае любая программная конструкция может быть обработана путем создания оператора для данной конструкции и рассмотрения семантически значащих компонентов конструкции в качестве операндов этого оператора.

В абстрактном синтаксическом дереве для выражения `9-5+2` на рис. 2.22 корень представляет оператор `+`. Поддеревья корня представляют подвыражения `9-5` и `2`. Группирование `9-5` в качестве операнда отражает порядок вычисления операторов с одинаковым уровнем приоритетов слева направо. Поскольку `-` и `+` имеют один и тот же приоритет, `9-5+2` эквивалентно `(9-5)+2`.

Абстрактные синтаксические деревья, или просто *синтаксические деревья*, похожи на деревья разбора, но в синтаксическом дереве внутренние узлы представляют программные конструкции, в то время как в дереве разбора внутренние узлы представляют нетерминалы. Многие нетерминалы грамматики представляют программные конструкции, но некоторые из них являются "вспомогательными" того или иного вида, такими, как представляющие множители, слагаемые или другие разновидности выражений. В синтаксическом дереве эти вспомогательные узлы обычно не нужны и потому опускаются. Чтобы подчеркнуть различия, дерево

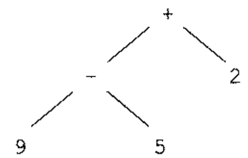


Рис. 2.22. Синтаксическое дерево для выражения 9–5+2

разбора иногда называют *конкретным синтаксическим деревом*, а лежащую в его основе грамматику — *конкретным синтаксисом* языка программирования.

В синтаксическом дереве на рис. 2.22 каждый внутренний узел связан с оператором без "вспомогательных" узлов для *одинарных продукций* (продукция, тело которой содержит только один нетерминал, и ничего более) наподобие $expr \rightarrow term$ или ϵ-продукций наподобие $rest \rightarrow \epsilon$.

Желательно, чтобы схема трансляции была основана на грамматике, деревья разбора которой, насколько это можно, близки к синтаксическим деревьям. Группировка подвыражений грамматикой на рис. 2.21 аналогична их группировке в синтаксических деревьях. Например, подвыражения оператора сложения представлены в теле продукции $expr + term$ членами $expr$ и $term$.

2.5.2 Адаптация схемы трансляции

Метод устранения левой рекурсии, набросок которой приведен на рис. 2.20, может быть применен и к продукциям, содержащим семантические действия. Сначала распространим этот метод на множественные продукции. В нашем примере A представляет собой $expr$ и для $expr$ у нас имеются две леворекурсивные продукции и одна, таковой не являющаяся. Метод устранения левой рекурсии трансформирует продукции $A \rightarrow A\alpha \mid A\beta \mid \gamma$ в

$$\begin{array}{lcl} A & \rightarrow & \gamma R \\ R & \rightarrow & \alpha R \mid \beta R \mid \epsilon \end{array}$$

Затем требуется преобразовать продукции с внедренными действиями, а не только с терминалами и нетерминалами. Семантические действия, внедренные в продукции, просто переносятся при преобразовании так же, как если бы они были терминалами.

Пример 2.13. Рассмотрим схему трансляции на рис. 2.21. Пусть

$$\begin{array}{lcl} A & = & expr \\ \alpha & = & + \; term \; \{ \; \mathrm{print}('+') \; \} \\ \beta & = & - \; term \; \{ \; \mathrm{print}('-') \; \} \\ \gamma & = & term \end{array}$$

В этом случае преобразование для устранения левой рекурсии приводит к схеме трансляции, показанной на рис. 2.23. Продукции для *expr* из рис. 2.21 преобразованы в одну продукцию для *expr* и продукции для нового нетерминала, который играет роль R. Продукции для *term* те же, что и на рис. 2.21. На рис. 2.24 показано, как выражение 9-5+2 транслируется с использованием грамматики, приведенной на рис. 2.23. □

$$\begin{array}{lcl} expr & \rightarrow & term \; rest \\ rest & \rightarrow & + \; term \;\; \{ \; \mathrm{print}('+') \; \} \;\; rest \\ & | & - \; term \;\; \{ \; \mathrm{print}('-') \; \} \;\; rest \\ & | & \epsilon \\ term & \rightarrow & 0 \;\; \{ \; \mathrm{print}('0') \; \} \\ & | & 1 \;\; \{ \; \mathrm{print}('1') \; \} \\ & & \ldots \\ & | & 9 \;\; \{ \; \mathrm{print}('9') \; \} \end{array}$$

Рис. 2.23. Схема трансляции после устранения левой рекурсии

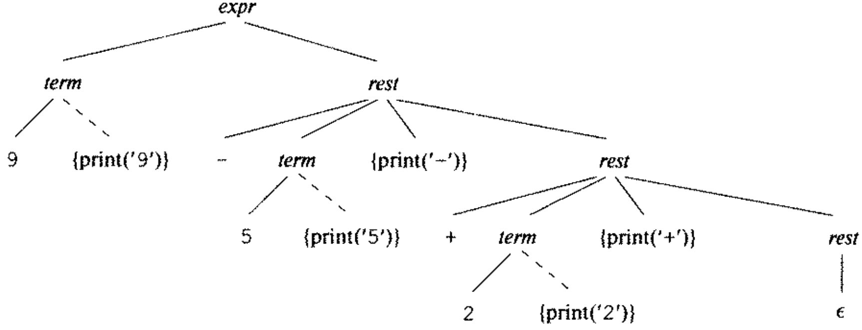


Рис. 2.24. Трансляция 9-5+2 в 95-2+

Устранение левой рекурсии должно выполняться очень аккуратно, чтобы гарантировать сохранение порядка семантических действий. Например, преобразованная схема на рис. 2.23 содержит действия { print('+') } и { print('-') } в средине

тела продукции, между нетерминалами *term* и *rest*. Если действие будет перемещено в конец тела продукции, после *rest*, трансляция станет некорректной. Читатель может самостоятельно убедиться, что в этом случае выражение `9-5+2` будет транслировано в `952+-`, постфиксную запись для `9-(5+2)`, вместо корректной постфиксной записи `95-2+` выражения `(9-5)+2`.

2.5.3 Процедуры для нетерминалов

Функции *expr*, *rest* и *term* на рис. 2.25 реализуют схему синтаксически управляемой трансляции из рис. 2.23. Эти функции имитируют тела продукций соответствующих нетерминалов. Функция *expr* реализует продукцию *expr* → *term rest* путем вызова *term* () с последующим вызовом *rest* ().

```
void expr() {
      term(); rest();
}

void rest() {
      if ( lookahead == '+' ) {
            match('+');  term();  print('+');  rest();
      }
      else if ( lookahead == '-' ) {
            match('-');  term();  print('-');  rest();
      }
      else { } /* Не делать ничего */ ;
}

void term() {
      if ( lookahead - цифра ) {
            t = lookahead;  match(lookahead);  print(t);
      }
      else report("syntax error");
}
```

Рис. 2.25. Псевдокод для нетерминалов expr, rest и term

Функция *rest* реализует три продукции для нетерминала *rest* на рис. 2.23. Она применяет первую продукцию, если сканируемый символ — знак "плюс", вторую — если это знак "минус", и продукцию *rest* → ϵ во всех прочих случаях. Первые две продукции для *rest* реализуются первыми двумя ветвями инструкции **if** в процедуре *rest*. Если сканируемый символ — +, вызывается функция соответствия *match* ('+'), перемещающаяся по входной строке. После вызова *term* ()

реализуется семантическое действие путем печати символа "плюс". Вторая продукция совершенно аналогична, только знак + заменяется знаком −. Поскольку третья продукция для *rest* в правой части содержит только ϵ, последняя конструкция else в функции *rest* не выполняет никаких действий.

Десять продукций для *term* генерируют десять цифр. Поскольку каждая из этих продукций генерирует цифру и печатает ее, один и тот же код на рис. 2.25 применим для всех них. При успешной проверке переменная *t* используется для хранения цифры, представленной переменной *lookahead*, поскольку последняя будет перезаписана вызовом *match*. Вспомните, что эта функция изменяет текущий сканируемый символ, так что цифра должна быть сохранена для последующей печати.[7]

2.5.4 Упрощение транслятора

Перед тем как показать вам полный текст программы, выполним два упрощающих преобразования кода на рис. 2.25. Эти упрощения приведут к переносу процедуры *rest* в процедуру *expr*. Такое упрощение позволяет уменьшить количество необходимых процедур при наличии многочисленных уровней приоритетов.

Во-первых, некоторые рекурсивные вызовы можно заменить итерациями. Когда последней выполняемой инструкцией в теле процедуры является рекурсивный вызов той же самой процедуры, говорят, что такой вызов *оконечно рекурсивен* (tail recursive). Например, в функции *rest* вызовы *rest* () для сканируемых символов + и − являются оконечно рекурсивными, поскольку рекурсивный вызов *rest* является последней инструкцией, выполняемой текущим вызовом *rest*.

В случае процедуры без параметров оконечно рекурсивный вызов можно заменить простым безусловным переходом в начало процедуры. Код функции *rest* может быть переписан так, как показано в псевдокоде на рис. 2.26. Пока сканируемый символ представляет собой знак "плюс" или "минус", процедура *rest* вызывает функцию *match* для данного символа, затем вызывает функцию *term* для цифры из входной строки и циклически продолжает этот процесс. В противном случае цикл **while** прекращает работу и выполняется выход из функции *rest*.

Во-вторых, завершенная программа на языке программирования Java должна включать еще одно изменение. Поскольку оконечно рекурсивный вызов *rest* на рис. 2.25 заменен итерациями, единственный остающийся вызов *rest* остается внутри процедуры *expr*. Таким образом, эти две процедуры могут быть объединены в одну путем замены вызова *rest* () телом процедуры *rest*.

[7]В качестве небольшой оптимизации можно выполнить печать перед вызовом *match*, чтобы избежать необходимости хранения цифры. В общем же случае изменение порядка действий и грамматических символов рискованно, поскольку при этом может измениться сама трансляция.

```
void rest() {
    while( true ) {
        if( lookahead == '+' ) {
            match('+'); term(); print('+'); continue;
        }
        else if ( lookahead == '-' ) {
            match('-'); term(); print('-'); continue;
        }
        break ;
    }
}
```

Рис. 2.26. Устранение оконечной рекурсии в процедуре rest из рис. 2.25

2.5.5 Завершенная программа

Завершенная программа на языке программирования Java показана на рис. 2.27. Первая строка на рис. 2.27, начинающаяся со слова `import`, предоставляет доступ к пакету `java.io` для системного ввода и вывода. Остальной код состоит из двух классов — `Parser` и `Postfix`. Класс `Parser` содержит переменную `lookahead` и функции `Parser`, `expr`, `term` и `match`.

Выполнение программы начинается с функции `main`, определенной в классе `Postfix`. Функция `main` создает экземпляр `parse` класса `Parser` и вызывает его функцию `expr` для синтаксического анализа выражения.

Функция `Parser`, имеющая то же имя, что и имя класса, является *конструктором*; она вызывается автоматически при создании объекта класса. Обратите внимание, что ее определение в начале класса `Parser` инициализирует переменную `lookahead` путем чтения токена. Токены, состоящие из отдельных символов, предоставляются системной подпрограммой `read`, которая считывает очередной символ из входной строки. Обратите внимание, что переменная `lookahead` объявлена как целое число, а не символ. Это сделано для того, чтобы упредить возможное появление в дальнейшем дополнительных токенов, которые будут отличны от отдельных символов.

Функция `expr` представляет собой результат применения упрощений, рассмотренных в разделе 2.5.4; она реализует нетерминалы *expr* и *rest* из рис. 2.23. Код функции `expr` на рис. 2.27 вызывает `term`, за которым следует цикл **while**, который бесконечно проверяет переменную `lookahead` на равенство `'+'` или `'-'`. Выход из цикла осуществляется по достижении инструкции `return`. Внутри цикла для вывода символа используются средства класса `System`.

Функция `term` использует подпрограмму `isDigit` из класса `Character` языка программирования Java для проверки того, что сканируемый символ явля-

```
import java.io.*; class Parser {
    static int lookahead;

    public Parser() throws IOException {
        lookahead = System.in.read();
    }

    void expr() throws IOException {
        term();
        while(true) {
            if( lookahead == '+' ) {
                match('+'); term(); System.out.write('+');
            }
            else if( lookahead == '-' ) {
                match('-'); term(); System.out.write('-');
            }
            else return;
        }
    }

    void term() throws IOException {
        if( Character.isDigit((char)lookahead) ) {
            System.out.write((char)lookahead); match(lookahead);
        }
        else throw new Error("syntax error");
    }

    void match(int t) throws IOException {
        if( lookahead == t ) lookahead = System.in.read();
        else throw new Error("syntax error");
    }
}

public class Postfix {
    public static void main(String[] args) throws IOException {
        Parser parse = new Parser();
        parse.expr(); System.out.write('\n');
    }
}
```

Рис. 2.27. Программа на языке программирования Java для трансляции инфиксных выражений в постфиксную запись

Несколько выдающихся возможностей Java

Тем, кто не знаком с языком программирования Java, при чтении кода на рис. 2.27 могут помочь следующие примечания.

- Класс в Java состоит из последовательности определений переменных и функций.
- Скобки вокруг списка параметров необходимы даже при отсутствии параметров; следовательно, надо писать `expr()` и `term()`. Эти функции в действительности являются процедурами, поскольку они не возвращают никаких значений, на что указывает ключевое слово `void` перед именем функции.
- Функции обмениваются информацией посредством либо передачи параметров по значению, либо обращения к общим данным. Например, функции `expr()` и `term()` получают информацию о сканируемом символе при помощи переменной класса `lookahead`, к которой они могут обратиться, являясь с ней членами одного класса `Parser`.
- Как и в C, в Java для присваивания используется оператор =, для проверки на равенство — ==, а для проверки на неравенство — !=.
- Конструкция `throws IOException` в определении `term()` объявляет, что может быть сгенерировано исключение `IOException`. Такие исключения генерируются в ситуации, когда функция `match` при помощи подпрограммы `read` не может считать очередной символ из входного потока. Любая функция, которая вызывает `match`, также должна объявить, что в процессе ее выполнения может быть сгенерировано исключение `IOException`.

ется цифрой. Подпрограмма `isDigit` применяется к символу, однако переменная `lookahead`, как уже говорилось, имеет тип `int` для обеспечения возможного будущего расширения грамматики. Конструкция `(char)lookahead` *приводит* `lookahead` к типу `char`. Небольшое отклонение от рис. 2.25 состоит в том, что семантическое действие по выводу сканируемого символа выполняется до вызова `match`.

Функция `match` проверяет терминалы; она считывает очередной входной терминал, если текущий сканируемый символ соответствует переданному, или в противном случае сообщает об ошибке при помощи

```
throw new Error("syntax error");
```

Этот код создает новое исключение класса `Error` со строкой `"syntax error"` в качестве сообщения об ошибке. Java не требует объявления исключений `Error` в конструкциях `throws`, поскольку они предназначены только для аварийных ситуаций, которые никогда не должны происходить.[8]

2.6 Лексический анализ

Лексический анализатор считывает символы из входной строки и группирует их в "объекты токенов". Вместе с терминальными символами, которые используются при синтаксическом анализе, объекты токенов несут дополнительную информацию в форме значений атрибутов. Пока что нам не требовалось отличать термины "токен" и "терминал", поскольку синтаксический анализатор игнорирует значения атрибутов в токенах. В данном разделе токен представляет собой терминал с дополнительной информацией.

Последовательность входных символов, составляющая отдельный токен, называется *лексемой* (lexeme). Таким образом, можно сказать, что лексический анализатор отделяет синтаксический анализатор от представления токенов в виде лексем.

Лексический анализатор в данном разделе допускает появление в выражениях чисел, идентификаторов и пробельных символов (whitespace), в число которых входят пробелы, символы табуляции и новой строки. Можно воспользоваться расширением транслятора выражений из предыдущего раздела. Поскольку грамматика выражений на рис. 2.21 должна быть расширена, чтобы позволять использование чисел и идентификаторов, сто́ит воспользоваться этой возможностью и добавить в нее умножение и деление. Такая расширенная схема трансляции приведена на рис. 2.28.

На рис. 2.28 предполагается, что терминал **num** имеет атрибут **num**.*value*, который содержит целое значение, соответствующее этому терминалу. Терминал **id** имеет строковый атрибут, записываемый как **id**.*lexeme*; предполагается, что эта строка представляет собой фактическую лексему, соответствующую данному экземпляру токена **id**.

---

[8]Обработка ошибок может быть осуществлена с использованием возможностей по обработке исключений в Java. Один из подходов состоит в определении нового исключения, скажем, `SyntaxError`, который расширяет системный класс `Exception`. Тогда в функциях `term` и `match` при обнаружении ошибки следует генерировать исключения `SyntaxError` вместо `Error`. Обработка исключений в `main` должна осуществляться путем помещения вызова `parse.expr()` в блок `try`, который перехватывает исключение `SyntaxError`, выводит сообщение и завершает работу программы. Для этого мы должны добавить в программу на рис. 2.27 класс `SyntaxError`. Для завершения модификации в дополнение к спецификации исключений `IOException` функции `match` и `term` должны объявить, что они могут генерировать исключение `SyntaxError`. То же должно быть сделано и в функции `expr`, которая их вызывает.

expr	→	*expr* + *term*	{ print('+') }
	|	*expr* − *term*	{ print('−') }
	|	*term*	
term	→	*term* * *factor*	{ print('*') }
	|	*term* / *factor*	{ print('/') }
	|	*factor*	
factor	→	(*expr*)	
	|	**num**	{ print(**num**.*value*) }
	|	**id**	{ print(**id**.*lexeme*) }

Рис. 2.28. Действия для трансляции в постфиксную запись

Фрагменты псевдокода использованы для иллюстрации действий лексического анализатора, который будет включен в код Java в конце этого раздела. Подход, описанный в данном разделе, применим для написания лексических анализаторов вручную. В разделе 3.5 будет описан инструмент под названием Lex, который генерирует лексический анализатор на основе предоставляемой ему спецификации. Таблицы символов или структуры данных для хранения информации об идентификаторах рассматриваются в разделе 2.7.

2.6.1 Удаление пробельных символов и комментариев

Транслятор выражений из раздела 2.5 просматривает каждый символ входного потока, так что посторонние символы, такие как пробелы, приводят к ошибке. Большинство языков программирования допускают произвольное количество пробельных символов между токенами. Аналогично в процессе синтаксического анализа игнорируются комментарии, так что они также могут рассматриваться как пробельные символы.

Если пробельные символы удаляются лексическим анализатором, синтаксический анализатор никогда не столкнется с ними. Альтернативным способом является изменение грамматики, при котором в нее включаются и пробельные символы, но он не так прост в реализации, как использование лексического анализатора для удаления пробельных символов.

Псевдокод на рис. 2.29 при чтении входного потока пропускает пробельные символы, такие как пробелы, символы табуляции и новой строки. Переменная *peek* содержит очередной считываемый символ. Номер строки и контекст в сообщениях об ошибках помогают точно локализовать место ошибки; в приведенном коде используется переменная *line*, подсчитывающая символы новой строки во входном потоке.

```
for ( ; ; peek = очередной считываемый символ ) {
    if ( peek — пробел или символ табуляции ) do ничего;
    else if ( peek — символ новой строки ) line = line+1;
    else break;
}
```

Рис. 2.29. Пропуск пробельных символов

2.6.2 Опережающее чтение

Лексическому анализатору может потребоваться прочесть некоторое количество символов за текущим, чтобы решить, какой именно токен следует вернуть синтаксическому анализатору. Например, лексический анализатор С, встретив символ >, должен прочесть следующий за ним символ. Если он окажется символом =, то > представляет собой часть последовательности >=, лексемы токена оператора "больше или равно"; если же это символ >, то первый символ представляет собой часть последовательности >>, лексемы токена оператора "битовый сдвиг вправо". В противном случае символ > сам по себе образует оператор "больше", и оказывается, что лексический анализатор прочитал один лишний символ из входного потока.

В общем случае подход к опережающему чтению входного потока состоит в поддержке входного буфера, из которого лексический анализатор может выполнять чтение и в который может возвращать прочитанные символы. Использовать входной буфер можно уже только потому, что он повышает эффективность анализатора, так как считывание блока символов обычно существенно более эффективно, чем посимвольное считывание. Указатель отслеживает проанализированную часть входа; возврат символов во входной поток осуществляется путем простого перемещения указателя. Методы буферизации ввода рассматриваются в разделе 3.2.

Обычно достаточно односимвольного опережающего чтения, так что простейшее решение состоит в использовании переменной, скажем, *peek*, для хранения очередного считанного символа. Лексический анализатор из этого раздела применяет опережающее на один символ чтение при сборке чисел из цифр или идентификаторов из символов; например, он считывает один символ после `1`, чтобы отличить `1` от `10`, и считывает один символ после `t`, чтобы отличить `t` от `true`.

Лексический анализатор использует опережающее чтение только тогда, когда это необходимо. Оператор наподобие `*` может быть идентифицирован без дополнительного чтения. В таких случаях в *peek* помещается пробел, который будет пропущен при вызове лексического анализатора для получения следующего токена. Инвариант в данном разделе заключается в том, что, когда лексический

анализатор возвращает токен, переменная *peek* содержит либо символ, следующий за лексемой текущего токена, либо пробел.

2.6.3 Константы

Везде, где в грамматике выражений встречается отдельная цифра, на ее месте естественно использовать произвольную целую константу. Целая константа может быть введена в грамматику путем либо создания для таких констант терминального символа, скажем, **num**, либо добавления в грамматику синтаксиса для целых констант. Работа по сборке последовательных цифр в целые числа и вычислению их значений обычно поручается лексическому анализатору, поскольку числа в процессе синтаксического анализа и трансляции можно рассматривать как отдельные элементы.

Когда во входном потоке встречается последовательность цифр, лексический анализатор передает синтаксическому анализатору токен, состоящий из терминала **num**, вместе с целочисленным атрибутом, вычисленным из цифр входного потока. Если записывать токены как кортежи в угловых скобках $\langle\rangle$, то входная строка `31+28+59` преобразуется в последовательность

$$\langle \mathbf{num}, 31\rangle \ \langle + \rangle \ \langle \mathbf{num}, 28\rangle \ \langle + \rangle \ \langle \mathbf{num}, 59\rangle$$

Терминальный символ + не имеет атрибутов, так что его кортеж представляет собой просто $\langle + \rangle$. Псевдокод на рис. 2.30 считывает цифры целого числа и накапливает его значение в переменной v.

```
if ( peek содержит цифру ) {
    v = 0;
    do {
        v = v * 10 + целочисленное значение цифры peek;
        peek = следующий входной символ;
    } while ( peek содержит цифру );
    return token ⟨num, v⟩;
}
```

Рис. 2.30. Группирование цифр в целые числа

2.6.4 Распознавание ключевых слов и идентификаторов

Большинство языков программирования используют фиксированные символьные строки, такие как `for`, `do` и `if`, в качестве знаков препинания или для идентификации конструкций. Такие символьные строки называются *ключевыми словами*.

Символьные строки могут также использоваться в качестве идентификаторов для имен переменных, массивов, функций и т.п. Грамматики обычно трактуют идентификаторы как терминалы для упрощения синтаксического анализатора, который может ожидать один и тот же терминал, скажем, **id**, всякий раз, когда во входном потоке оказывается идентификатор. Например, при входной строке

$$\texttt{count = count + increment;} \tag{2.6}$$

синтаксический анализатор работает с потоком терминалов **id** = **id** + **id**. Токен **id** имеет атрибут, хранящий лексему. Записывая токены в виде кортежей, мы получим для входного потока (2.6) токены

$$\langle \textbf{id}, \texttt{"count"}\rangle\ \langle=\rangle\ \langle \textbf{id}, \texttt{"count"}\rangle\ \langle+\rangle\ \langle \textbf{id}, \texttt{"increment"}\rangle\ \langle\texttt{;}\rangle$$

Ключевые слова обычно удовлетворяют правилам, по которым формируются идентификаторы, так что необходим механизм, который принимал бы решения, когда лексема образует ключевое слово, а когда — идентификатор. Задача решается существенно легче, если ключевые слова являются *зарезервированными*, т.е. если они не могут использоваться в качестве идентификаторов. Тогда символьная строка образует идентификатор только в том случае, если она не является ключевым словом.

Лексический анализатор из этого раздела использует таблицу для хранения символьных строк для решения двух задач.

- *Единственность представления.* Таблица строк может отделить остальную часть компилятора от представления строк, поскольку фазы компилятора могут работать со ссылками или указателями на строки таблицы. Работа со ссылками может также оказаться более эффективной, чем работа непосредственно со строками.

- *Зарезервированные слова.* Зарезервированные слова могут быть реализованы путем инициализации таблицы зарезервированными строками и их токенами. Когда лексический анализатор считывает строку или лексему, которая может образовывать идентификатор, он сначала проверяет, нет ли такой лексемы в таблице строк. Если она имеется в таблице, то лексический анализатор возвращает соответствующий токен из таблицы; если нет — возвращается токен с терминалом **id**.

В Java таблица строк может быть реализована как хеш-таблица с использованием класса *Hashtable*. Объявление

Hashtable words = **new** *Hashtable*();

```
if ( peek содержит букву ) {
    Собираем буквы или цифры в буфер b;
    s = строка, образованная символами в b;
    w = токен, возвращенный вызовом words.get(s);
    if ( w не равно null ) return w;
    else {
        Внести пару "ключ–значение" pair (s, ⟨id, s⟩) в words
        return token ⟨id, s⟩;
    }
}
```

Рис. 2.31. Различение ключевых слов и идентификаторов

делает *words* хеш-таблицей по умолчанию, которая отображает ключи на значения. Мы используем ее для отображения лексем на токены. Псевдокод на рис. 2.31 использует операцию *get* для поиска зарезервированных слов.

Приведенный псевдокод собирает из букв и цифр входного потока строку s, начинающуюся с буквы. Предполагается, что строка s должна быть максимально возможной длины, т.е. лексический анализатор продолжает чтение входного потока до тех пор, пока из него считываются буквы и цифры. Когда из входного потока считывается некоторый другой символ, например пробельный, лексема копируется в буфер b. Если в таблице имеется запись для s, то возвращается токен, полученный вызовом *words.get*. В этом случае строка s может представлять собой либо ключевое слово из тех, которые были изначально помещены в таблицу *words*, либо идентификатор, ранее внесенный в таблицу. В противном случае токен **id** с атрибутом s вносится в таблицу и возвращается лексическим анализатором.

2.6.5 Лексический анализатор

Фрагменты псевдокодов из всего раздела можно объединить в одну функцию *scan*, которая возвращает объекты токенов, следующим образом:

```
Token scan() {
    Пропуск пробельных символов (раздел 2.6.1);
    Обработка чисел (раздел 2.6.3);
    Обработка зарезервированных слов и идентификаторов (раздел 2.6.4);
    /* Если мы достигли этого места, рассматриваем считанный символ
       peek как токен */
    Token t = new Token(peek);
    peek = пробел /* Инициализация (раздел 2.6.2) */ ;
    return t;
}
```

В оставшейся части этого раздела функция *scan* реализована как часть Java-пакета для лексического анализа. Этот пакет под названием `lexer` содержит классы для токенов и класс `Lexer`, содержащий функцию *scan*.

Классы для токенов и их поля проиллюстрированы на рис. 2.32; методы классов на рисунке не показаны. Класс `Token` имеет поле `tag`, которое используется для принятия решений в процессе анализа. Подкласс `Num` добавляет поле `value` для целого значения. Подкласс `Word` добавляет поле `lexeme`, которое используется для зарезервированных слов и идентификаторов.

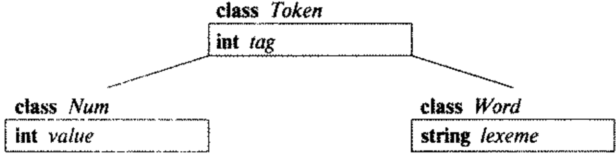


Рис. 2.32. Класс Token и подклассы Num и Word

Каждый класс находится в собственном файле. Файл для класса `Token` имеет следующий вид:

```
1) package lexer;                       // Файл Token.java
2) public class Token {
3)     public final int tag;
4)     public Token(int t) { tag = t; }
5) }
```

Строка 1 идентифицирует пакет `lexer`. Поле `tag` в строке 3 объявлено как `final`, так что, будучи установленным, оно не может быть изменено. Конструктор `Token` в строке 4 используется для создания объекта токена. Так,

```
new Token('+')
```

создает новый объект класса `Token` и устанавливает его поле `filed` равным целочисленному представлению `'+'`. (Для краткости мы опускаем обычный метод `toString`, который должен вернуть строку для вывода на печать.)

Там, где в псевдокоде встречаются терминалы наподобие **num** и **id**, код на языке программирования Java использует целые константы. Класс `Tag` реализует эти константы следующим образом:

```
1) package lexer;                       // Файл Tag.java
2) public class Tag {
3)     public final static int
4)         NUM = 256, ID = 257, TRUE = 258, FALSE = 259;
5) }
```

В дополнение к целочисленным полям NUM и ID этот класс определяет два дополнительных поля TRUE и FALSE для будущего использования; они будут использованы для иллюстрации работы с ключевыми словами.[9]

Поля в классе Tag объявлены как public, т.е. они могут использоваться и вне пакета. Они имеют спецификатор static, так что существует только один экземпляр каждого из этих полей. И наконец, они объявлены как final, т.е. они могут быть установлены только один раз. В сущности, эти поля являются константами. В C аналогичный эффект достигается при помощи определения макроса NUM как символьной константы, например

```
#define NUM 256
```

Код на языке программирования Java обращается к Tag.NUM и Tag.ID там, где псевдокод использует терминалы **num** и **id**. Единственное требование заключается в том, что Tag.NUM и Tag.ID должны быть инициализированы различными значениями, отличающимися друг от друга и от констант, представляющих односимвольные токены, такие как '+' и '*'.

```
1) package lexer;                    // Файл Num.java
2) public class Num extends Token {
3)    public final int value;
4)    public Num(int v) { super(Tag.NUM); value = v; }
5) }
```

```
1) package lexer;                    // Файл Word.java
2) public class Word extends Token {
3)    public final String lexeme;
4)    public Word(int t, String s) {
5)       super(t); lexeme = new String(s);
6)    }
7) }
```

Рис. 2.33. Подклассы Num и Word класса Token

На рис. 2.33 показаны классы Num и Word. Класс Num расширяет класс Token путем объявления целочисленного поля value в строке 3. Конструктор Num в строке 4 делает вызов super(Tag.NUM), который устанавливает поле tag в суперклассе Token равным Tag.NUM.

Класс Word используется и для зарезервированных слов, и для идентификаторов, так что конструктор Word в строке 4 получает два параметра: лексему

[9] Обычно ASCII-символы преобразуются в целые числа от 0 до 255. Поэтому для терминалов мы используем целые числа, превышающие 255.

и соответствующее целочисленное значение для tag. Объект для зарезервированного слова true может быть создан путем выполнения кода

```
new Word(Tag.true, "true")
```

Он создает новый объект с полем tag, установленным равным Tag.*true*, и полем lexeme, равным строке "true".

```
 1) package lexer;                    // Файл Lexer.java
 2) import java.io.*; import java.util.*;
 3) public class Lexer {
 4)    public int line = 1;
 5)    private char peek = ' ';
 6)    private Hashtable words = new Hashtable();
 7)    void reserve(Word t) { words.put(t.lexeme, t); }
 8)    public Lexer() {
 9)       reserve( new Word(Tag.TRUE,  "true")  );
10)       reserve( new Word(Tag.FALSE, "false") );
11)    }
12)    public Token scan() throws IOException {
13)       for( ; ; peek = (char)System.in.read() ) {
14)          if( peek == ' ' || peek == '\t' ) continue;
15)          else if( peek == '\n' ) line = line + 1;
16)          else break;
17)       }
          /* Продолжение на рис. 2.35 */
```

Рис. 2.34. Код лексического анализатора (часть 1)

Класс Lexer, предназначенный для лексического анализа, показан на рис. 2.34 и 2.35. Целочисленная переменная line в строке 4 подсчитывает количество входных строк, а переменная peek в строке 5 хранит очередной входной символ.

Зарезервированные слова обрабатываются в строках 6–11. Таблица words объявлена в строке 6. Вспомогательная функция reserve в строке 7 помещает пару "строка — слово" в таблицу. Конструктор Lexer в строках 9 и 10 инициализирует таблицу. Он использует конструктор Word для создания объектов слов, которые передаются вспомогательной функции reserve. Таким образом, таблица оказывается инициализированной зарезервированными словами "true" и "false" до первого вызова функции scan.

Код функции scan на рис. 2.34 и 2.35 реализует фрагменты псевдокодов, приведенные ранее в этом разделе. Цикл **for** в строках 13 и 14 пропускает пробелы,

```
18)         if( Character.isDigit(peek) ) {
19)            int v = 0;
20)            do {
21)               v = 10*v + Character.digit(peek, 10);
22)               peek = (char)System.in.read();
23)            } while( Character.isDigit(peek) );
24)            return new Num(v);
25)         }
26)         if( Character.isLetter(peek) ) {
27)            StringBuffer b = new StringBuffer();
28)            do {
29)               b.append(peek);
30)               peek = (char)System.in.read();
31)            } while( Character.isLetterOrDigit(peek) );
32)            String s = b.toString();
33)            Word w = (Word)words.get(s);
34)            if( w != null ) return w;
35)            w = new Word(Tag.ID, s);
36)            words.put(s, w);
37)            return w;
38)         }
39)         Token t = new Token(peek);
40)         peek = ' ';
41)         return t;
42)      }
43) }
```

Рис. 2.35. Код лексического анализатора (часть 2)

символы табуляции и новой строки. Когда в переменной `peek` оказывается не пробельный символ, управление покидает цикл.

Код для чтения последовательностей символов располагается в строках 18–25. Функция `isDigit` – из встроенного класса Java `Character`. Она используется в строке 18 для того, чтобы проверить, не содержит ли переменная `peek` цифру. Если содержит, код в строках 19–24 накапливает целочисленное значение из последовательности цифр во входном потоке и возвращает новый объект `Num`.

Строки 26–38 анализируют зарезервированные слова и идентификаторы. Ключевые слова **true** и **false** зарезервированы в строках 9 и 10. Таким образом, строка 35 достигается, только если строковая переменная `s` не представляет собой зарезервированное слово, так что она должна быть лексемой идентификатора.

Строка 35 возвращает новый объект типа `Word`, поле `lexeme` которого устанавливается равным `s`, а `tag` — равным `Tag.ID`. Наконец, строки 39–41 возвращают текущий символ как токен и сохраняют в переменной `peek` символ пробела, который будет пропущен при следующем вызове функции `scan`.

2.6.6 Упражнения к разделу 2.6

Упражнение 2.6.1. Расширьте лексический анализатор из раздела 2.6.5 так, чтобы он мог удалять комментарии, определенные следующим образом.

а) Комментарий начинается с символов `//` и включает все символы до конца данной строки.

б) Комментарий начинается с последовательности символов `/*` и включает все символы до последовательности `*/`.

Упражнение 2.6.2. Расширьте лексический анализатор из раздела 2.6.5 так, чтобы он мог распознавать операторы отношений `<`, `<=`, `==`, `!=`, `>=`, `>`.

Упражнение 2.6.3. Расширьте лексический анализатор из раздела 2.6.5 так, чтобы он мог распознавать числа с плавающей точкой, такие как `2.`, `3.14` и `.5`.

2.7 Таблицы символов

Таблицы символов представляют собой структуры данных, которые используются компилятором для хранения информации о конструкциях исходной программы. Информация накапливается инкрементно в фазе анализа компилятора и используется фазой синтеза для генерации целевого кода. Записи в таблице символов содержат информацию об идентификаторах, такую как их символьные строки (или лексемы), тип, местоположение в памяти и прочую связанную с ними информацию. Обычно таблицы символов должны поддерживать множественные объявления одного и того же идентификатора в программе.

Из раздела 1.6.1 мы знаем, что область видимости объявления представляет собой часть программы, в которой может применяться данное объявление. Можно реализовать области видимости путем настройки отдельной таблицы символов для каждой области видимости. Программный блок с объявлениями[10] будет иметь собственную таблицу символов с записями для каждого объявления в блоке. Такой подход работает и для других конструкций с областями видимости; например, класс может иметь собственную таблицу с записями для каждого поля и метода.

В этом разделе содержится модуль для работы с таблицей символов, подходящий для применения в фрагментах транслятора на языке программирования Java

[10] В C, например, программные блоки являются либо функциями, либо отделенными фигурными скобками разделами функций, содержащими одно или несколько объявлений.

из этой главы. Этот модуль будет использован в приложении А, когда фрагменты транслятора будут объединены в одно целое. Пока же для простоты основной пример в данном разделе будет представлять собой урезанный язык только с теми ключевыми конструкциями, которые имеют отношение к таблицам символов, а именно — с блоками, объявлениями и идентификаторами. Все прочие инструкции и выражения опущены, так что мы можем сконцентрироваться исключительно на операциях с таблицей символов. Программа состоит из блоков с необязательными объявлениями и "инструкциями", представляющих собой отдельные идентификаторы. Каждая такая инструкция представляет использование идентификатора. Вот пример программы на таком языке:

```
{ int x; char y; { bool y; x; y; } x; y; }        (2.7)
```

Примеры блочной структуры в разделе 1.6.3 работают с определениями и использованиями имен; входная строка (2.7) состоит исключительно из определений и использований имен.

Задача, которую мы будем решать, заключается в выводе видоизмененной программы, из которой удалены объявления, и каждая "инструкция" содержит идентификатор, за которым следуют двоеточие и его тип.

Пример 2.14. Приведенная выше входная строка (2.7) должна привести к выводу

```
{ { x:int; y:bool; } x:int; y:char; }
```

Первые x и y — из внутреннего блока входной строки (2.7). Поскольку данное использование x обращается к объявлению x во внешнем блоке, за ним следует тип int из этого объявления. Использование y во внутреннем блоке соответствует объявлению y в этом же блоке, так что его тип — bool. Далее следуют использования x и y во внешнем блоке с типами, взятыми из объявлений во внешнем блоке, т.е. int и char соответственно. □

2.7.1 Таблица символов для области видимости

Термин "область видимости идентификатора x" фактически означает область видимости конкретного объявления x. Термин *область видимости* сам по себе означает часть программы, которая является областью видимости одного или нескольких объявлений.

Области видимости весьма важны, поскольку один и тот же идентификатор может быть объявлен в разных частях программы для разных целей. Распространенные имена наподобие i или x очень часто многократно используются в одной и той же программе. В качестве другого примера можно привести перекрытие метода суперкласса в подклассе.

Создание записей таблицы символов

Записи таблицы символов создаются и используются в процессе фазы анализа лексическим, синтаксическим и семантическим анализаторами. В этой главе записи в таблице символов создает синтаксический анализатор. В связи с его знаниями о синтаксической структуре программы синтаксический анализатор зачастую куда лучше отличает различные объявления идентификаторов, чем лексический анализатор.

В некоторых случаях лексический анализатор способен, встречая образующие лексемы последовательности символов, создавать записи в таблице символов. Однако гораздо чаще лексический анализатор может только вернуть синтаксическому анализатору токен, скажем, **id**, вместе с указателем на лексему. Однако решить, следует ли использовать ранее созданную запись в таблице символов или для данного идентификатора следует создать новую запись, может только синтаксический анализатор.

Если блоки могут быть вложенными, то несколько объявлений одного и того же идентификатора могут появляться в пределах одного и того же блока. Приведенный далее синтаксис дает вложенные блоки, если *stmts* может генерировать блок:

$$block \rightarrow \text{'\{'}\ decls\ stmts\ \text{'\}'}$$

(Фигурные скобки заключены в кавычки для того, чтобы отличать их от фигурных скобок семантических действий.) В грамматике, приведенной на рис. 2.38, *decls* генерирует необязательную последовательность объявлений, а *stmts* генерирует необязательную последовательность инструкций. Кроме того, инструкция может быть блоком, так что наш язык допускает наличие вложенных блоков, в которых возможно переобъявление идентификаторов.

Правило *последнего вложения* (most-closely nested) для блоков заключается в том, что идентификатор x находится в области видимости последнего по вложенности объявления x, т.е. объявление идентификатора x следует искать путем исследования блоков изнутри наружу, начиная с блока, в котором находится интересующий нас идентификатор x.

Пример 2.15. В приведенном далее псевдокоде нижние индексы используются для того, чтобы отличать разные объявления одного и того же идентификатора:

Оптимизация таблиц символов для блоков

Реализация таблиц символов для блоков может использовать преимущества правила последнего вложения. Вложение гарантирует, что цепочка применяемых таблиц символов образует стек. На вершине стека находится таблица для текущего блока. Ниже в стеке располагаются таблицы охватывающих блоков. Таким образом, таблицы символов могут выделяться и освобождаться с использованием стековых методов.

Некоторые компиляторы поддерживают единую хеш-таблицу доступных записей, т.е. записей, которые не скрыты объявлениями во вложенном блоке. Такая хеш-таблица обеспечивает, по сути, константное время поиска ценой вставки и удаления записей при входе в блок и выходе из него. При выходе из блока B компилятор должен отменить все изменения, внесенные в хеш-таблицу объявлениями в блоке B. Это можно сделать при помощи вспомогательного стека для отслеживания изменений в хеш-таблице при обработке блока B.

1) $\{$ **int** x_1; **int** y_1;
2) $\{$ **int** w_2; **bool** y_2; **int** z_2;
3) $\cdots w_2 \cdots;\ \cdots x_1 \cdots;\ \cdots y_2 \cdots;\ \cdots z_2 \cdots;$
4) $\}$
5) $\cdots w_0 \cdots;\ \cdots x_1 \cdots;\ \cdots y_1 \cdots;$
6) $\}$

Индексы не являются частью идентификаторов; фактически это номера строк объявлений, относящихся к тому или иному идентификатору. Так, все встречающиеся в фрагменте идентификаторы x находятся в области видимости объявления в строке 1. Появление y в строке 3 относится к области видимости объявления y в строке 2, поскольку переменная y переобъявлена во внутреннем блоке. Однако появление y в строке 5 находится в области видимости объявления y в строке 1.

Появление w в строке 5, по-видимому, относится к области видимости объявления w вне этого программного фрагмента; индекс 0 этой переменной означает глобальное или внешнее по отношению к текущему блоку объявление.

Наконец, переменная z объявлена и используется внутри вложенного блока, но не может использоваться в строке 5, поскольку вложенное объявление применимо только к вложенному блоку. □

Правило последнего вложения для блоков может быть реализовано при помощи цепочек таблиц символов, когда таблица для вложенного блока указывает на таблицу для охватывающего блока.

Пример 2.16. На рис. 2.36 приведены таблицы символов для псевдокода в примере 2.15. Здесь B_1 — блок, начинающийся в строке 1, а B_2 — блок, начинающийся в строке 2. На вершине рисунка показана дополнительная таблица символов B_0 для всех глобальных объявлений или объявлений по умолчанию, предоставляемых языком программирования. Во время анализа строк 2–4 среда представлена ссылкой на самую нижнюю таблицу символов — для блока B_2. При переходе к строке 5 таблица символов для B_2 становится недоступной, и среда ссылается не на нее, а на таблицу символов для блока B_1, из которой можно достичь глобальной таблицы символов, но не таблицы символов для блока B_2. □

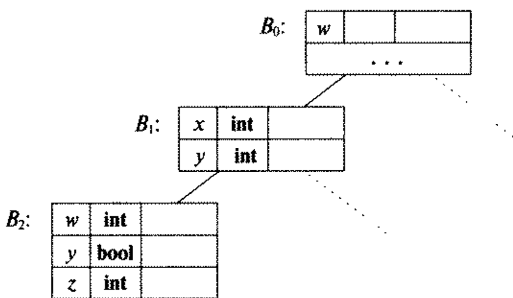


Рис. 2.36. Цепочки таблиц символов для примера 2.15

Реализация цепочки таблиц символов на языке программирования Java, показанная на рис. 2.37, определяет класс `Env` (от "environment" — "среда").[11] Класс `Env` поддерживает три операции.

- *Создание новой таблицы символов.* Конструктор `Env(p)` в строках 6–8 на рис. 2.37 создает объект `Env` с хеш-таблицей `table`. Объект вставляется в цепочку при помощи присваивания параметра `p` полю `prev`. Хотя, строго говоря, цепочку образуют объекты `Env`, удобнее говорить о цепочке таблиц.

- *Размещение* новой записи в текущей таблице. Хеш-таблица хранит пары "ключ — значение", где

  – *ключ* представляет собой строку или ссылку на строку; в качестве ключей можно использовать также ссылки на объекты токенов;

[11] "Среда" — еще один термин для коллекции таблиц символов, имеющих отношение к данной точке программы.

- *значение* представляет собой запись с типом Symbol; код в строках 9–11 не требует знания структуры записи, т.е. он не зависит от полей и методов класса Symbol.

• *Получение* записи для идентификатора путем поиска в цепочке таблиц, начиная с таблицы для текущего блока. Код для этой операции в строках 12–18 возвращает либо запись таблицы символов, либо значение null.

```
 1) package symbols;                    // Файл Env.java
 2) import java.util.*;
 3) public class Env {
 4)    private Hashtable table;
 5)    protected Env prev;

 6)    public Env(Env p) {
 7)       table = new Hashtable(); prev = p;
 8)    }

 9)    public void put(String s, Symbol sym) {
10)       table.put(s, sym);
11)    }

12)    public Symbol get(String s) {
13)       for( Env e = this; e != null; e = e.prev ) {
14)          Symbol found = (Symbol)(e.table.get(s));
15)          if( found != null ) return found;
16)       }
17)       return null;
18)    }
19) }
```

Рис. 2.37. Класс Env реализует цепочку таблиц символов

Объединение таблиц символов в цепочки дает в результате древовидную структуру, поскольку в охватывающий блок может быть вложено несколько блоков. Пунктирные линии на рис. 2.36 — это напоминание о том, что связанные в цепочки таблицы символов могут образовывать дерево.

2.7.2 Использование таблиц символов

В сущности, роль таблицы символов заключается в передаче информации от объявлений к использованиям. Семантические действия “помещают” информацию об идентификаторе x в таблицу символов при анализе объявления x. Позже

семантическое действие, связанное с продукцией наподобие *factor* → **id**, "получает" информацию об идентификаторе из таблицы символов. Поскольку трансляция выражения E_1 **op** E_2 для типичного оператора **op** зависит только от трансляций E_1 и E_2 и не зависит непосредственно от таблицы символов, можно добавить любое количество операторов без изменения основного потока информации от объявлений к использованиям через таблицу символов.

Пример 2.17. Схема трансляции на рис. 2.38 иллюстрирует применение класса `Env`. Схема трансляции концентрируется на областях видимости, объявлениях и использованиях и реализует трансляцию, описанную в примере 2.14. Как упоминалось ранее, для входной строки

```
{ int x; char y; { bool y; x; y; } x; y; }
```

схема трансляции удаляет объявления и дает строку

```
{ { x:int; y:bool; } x:int; y:char; }
```

Тела продукций на рис. 2.38 выровнены так, чтобы грамматические символы находились в одном столбце, а все действия — в другом. В результате компоненты тела продукции часто растягиваются на несколько строк.

Рассмотрим теперь семантические действия. Схема трансляции создает и уничтожает таблицы символов соответственно при входе в блок и выходе из него. Переменная *top* означает верхнюю таблицу, располагающуюся в заголовке цепочки таблиц. Первая продукция грамматики — *program* → *block*. Семантическое действие перед *block* инициализирует переменную *top* значением **null**, т.е. никаких записей пока нет.

Вторая продукция, *block* → '{' *decls stmts* '}', содержит действия при входе в блок и выходе из него. При входе в блок, до *decls*, семантическое действие сохраняет ссылку на текущую таблицу с использованием локальной переменной *saved*. Для каждого применения этой продукции используется своя локальная переменная *saved*, отличная от локальных переменных для других применений данной продукции. В синтаксическом анализаторе, работающем по методу рекурсивного спуска, переменная *saved* локальна для процедуры *block*. Локальные переменные в рекурсивных функциях рассматриваются в разделе 7.2. Код

top = **new** *Env*(*top*);

присваивает переменной *top* вновь созданную таблицу символов, которая соединена с предыдущим значением *top*, непосредственно перед входом в блок. Переменная *top* представляет собой объект класса *Env*; код конструктора *Env* приведен на рис. 2.37.

При выходе из блока, после '}', семантическое действие восстанавливает сохраненное при входе в блок значение переменной *top*. По существу, таблицы

program	→		{ *top* = **null**; }
		block	
block	→	'{'	{ *saved* = *top*; *top* = **new** *Env*(*top*); print("{ "); }
		decls stmts '}'	{ *top* = *saved*; print("} "); }
decls	→	*decls decl*	
	|	ϵ	
decl	→	**type id** ;	{ *s* = **new** *Symbol*; *s.type* = **type**.*lexeme* *top.put*(**id**.*lexeme*, *s*); }
stmts	→	*stmts stmt*	
	|	ϵ	
stmt	→	*block*	
	|	*factor* ;	{ print("; "); }

factor	→	**id**	{ *s* = *top.get*(**id**.*lexeme*);
			 print(**id**.*lexeme*);
			 print(":");
			 print(*s.type*); }

Рис. 2.38. Использование таблицы символов для трансляции языка с блоками

символов образуют стек; восстановление сохраненного значения *top* снимает со стека объявления в блоке.[12] Таким образом, объявления в блоке не видны вне этого блока.

Объявление, *decls* → **type id**, приводит к новой записи для объявленного идентификатора. Полагаем, что токены **type** и **id** имеют связанные с ними атрибуты, которые представляют собой соответственно тип и лексему объявляемого идентификатора. Нас не интересуют все поля объекта *s*, но мы полагаем, что у него имеется поле *type*, которое дает нам тип соответствующего идентификатора. Мы создаем новый объект *s* и определяем его свойство типа путем присваивания

[12]Вместо явного сохранения и восстановления таблиц можно добавить статические операции внесения в стек и снятия со стека *push* и *pop* к классу Env.

s.type = **type**.*lexeme*. Запись помещается в верхнюю таблицу символов при помощи вызова *top.put*(**id**.*lexeme*, *s*).

Семантическое действие в продукции *factor* → **id** использует таблицу символов для получения записи, соответствующей идентификатору. Операция *get* выполняет поиск первой записи в цепочке таблиц, начиная с таблицы *top*. Полученная запись содержит всю необходимую информацию об идентификаторе, такую, например, как его тип. □

2.8 Генерация промежуточного кода

Начальная стадия компилятора строит промежуточное представление исходной программы, на основании которого заключительная стадия генерирует целевую программу. В этом разделе мы рассмотрим промежуточное представление для выражений и инструкций и приведем учебные примеры получения таких представлений.

2.8.1 Два вида промежуточных представлений

Как говорилось в разделе 2.1 и, в частности, было показано на рис. 2.4, основными промежуточными представлениями являются:

- деревья, включая деревья разбора и (абстрактные) синтаксические деревья;
- линейные представления, в частности "трехадресный код".

С абстрактными синтаксическими деревьями, или просто синтаксическими деревьями, вы встречались в разделе 2.5.1, а в разделе 5.3.1 они будут изучаться более формализовано. В процессе синтаксического анализа для представления важных программных конструкций создаются узлы синтаксического дерева. В процессе анализа к ним добавляется различная информация в виде атрибутов, связанных с этими узлами. Выбор атрибутов зависит от выполняемой трансляции.

Трехадресный код, в свою очередь, представляет собой последовательность элементарных программных шагов, таких как, например, сложение двух величин. В отличие от деревьев, здесь нет иерархической структуры. Как мы увидим в главе 9, такое представление становится необходимым, если мы намерены выполнять сколько-нибудь существенную оптимизацию кода. В этом случае мы разбиваем длинную последовательность трехадресных инструкций, образующую программу, на "базовые блоки", которые представляют собой последовательности инструкций, всегда выполняющихся одна за другой, без ветвления.

В дополнение к созданию промежуточного представления на начальной стадии проверяется, что исходная программа следует синтаксическим и семантическим правилам исходного языка. Такая проверка называется *статической*; в об-

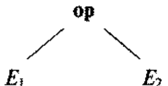
щем случае "статический" означает "выполняемый компилятором".[13] Статическая проверка гарантирует, что определенные виды программных ошибок, включая несоответствие типов, обнаруживаются во время компиляции.

Компилятор может строить синтаксическое дерево одновременно с генерацией трехадресного кода. Однако обычно компиляторы генерируют трехадресный код в то время, когда синтаксический анализатор "делает вид", что строит дерево, но без явного построения завершенного синтаксического дерева. Вместо этого компилятор хранит узлы и их атрибуты, необходимые для семантических проверок или иных целей, вместе со структурами данных, необходимых для синтаксического анализа. При этом части синтаксического дерева, требующиеся для построения трехадресного кода, оказываются доступными в тот момент, когда это необходимо, но, когда потребность в них отпадает, они уничтожаются. Детальнее об этом мы поговорим в главе 5.

2.8.2 Построение синтаксических деревьев

Сначала мы приведем схему трансляции для построения синтаксических деревьев, а позже, в разделе 2.8.4, покажем, как эта схема может быть модифицирована для получения трехадресного кода вместе с синтаксическим деревом (или вместо него).

Вспомним из раздела 2.5.1, что синтаксическое дерево



представляет выражение, образованное применением оператора **op** к подвыражениям, представленным узлами E_1 и E_2. Синтаксические деревья могут быть построены для любой конструкции, а не только для выражений. Каждая конструкция представлена узлом, дочерние узлы которого представляют семантически значащие компоненты конструкции. Например, семантически значащими компонентами цикла **while** языка программирования C

while (*expr*) *stmt*

[13]В противоположность этому "динамический" означает "во время выполнения программы". Многие языки выполняют также определенные динамические проверки. Например, объектно-ориентированные языки программирования наподобие Java иногда должны выполнять проверку типов в процессе выполнения программы, поскольку метод, применяемый к объекту, может зависеть от конкретного подкласса этого объекта.

являются выражение *expr* и инструкция *stmt*.[14] Узел синтаксического дерева для такой конструкции содержит оператор, который мы назовем **while**, и имеет два дочерних узла — синтаксические деревья для *expr* и *stmt*.

Схема трансляции, приведенная на рис. 2.39, строит синтаксические деревья для представительного, но очень ограниченного языка выражений и инструкций. Все нетерминалы в схеме трансляции имеют атрибут n, который является узлом синтаксического дерева. Узлы реализованы как объекты класса *Node*.

Класс *Node* имеет два непосредственных подкласса: *Expr* для выражений всех видов и *Stmt* для всех видов инструкций. Каждый тип инструкции имеет соответствующий подкласс класса *Stmt*; например, оператор **while** соответствует подклассу *While*. Узел синтаксического дерева для оператора **while** с дочерними узлами x и y создается при помощи псевдокода

$$\mathbf{new}\ \mathit{While}\,(x, y)$$

Этот псевдокод создает объект класса *While* путем вызова конструктора *While*, имеющего то же имя, что и имя класса. Параметры конструктора соответствуют операндам в абстрактном синтаксисе, так же как сам конструктор соответствует оператору.

При детальном рассмотрении кода в приложении А будут изучены методы этой иерархии классов; здесь же мы ограничимся только неформальным рассмотрением нескольких методов.

Мы по очереди рассмотрим все продукции и правила, показанные на рис. 2.39. Первыми будут рассмотрены продукции, определяющие различные типы инструкций, после чего последуют продукции, определяющие наши ограниченные типы выражений.

Синтаксические деревья для инструкций

Для каждого вида инструкций в абстрактном синтаксисе определяется свой оператор. Для конструкций, которые начинаются с ключевого слова, в качестве оператора будет использоваться само ключевое слово. Таким образом, существуют оператор **while** для цикла **while** и оператор **do** для цикла **do-while**. Условные конструкции могут обрабатываться путем определения двух операторов, **ifelse** и **if**, для инструкции **if** соответственно с частью **else** и без нее. В нашем простом учебном языке мы не используем **else**, так что у нас имеются только инструкции **if**. Добавление **else** представляет определенную сложность и будет рассмотрено в разделе 4.8.2.

Каждый оператор инструкции имеет соответствующий класс с тем же именем, но с прописной первой буквой; например, оператору **if** соответствует класс *If*. Кро-

[14]Правая скобка служит исключительно для отделения выражения от инструкции. Левая скобка фактически не имеет значения; она нужна только для красоты, поскольку без нее программа на С допускала бы наличие несбалансированных скобок.

program	→	*block*	{ return *block.n*; }
block	→	'{' *stmts* '}'	{ *block.n* = *stmts.n*; }
stmts	→	$stmts_1\ stmt$	{ *stmts.n* = **new** *Seq* $(stmts_1.n, stmt.n)$; }
	|	ϵ	{ *stmts.n* = **null**; }
stmt	→	*expr* ;	{ *stmt.n* = **new** *Eval* (*expr.n*); }
	|	**if** (*expr*) $stmt_1$	{ *stmt.n* = **new** *If* $(expr.n, stmt_1.n)$; }
	|	**while** (*expr*) $stmt_1$	{ *stmt.n* = **new** *While* $(expr.n, stmt_1.n)$; }
	|	**do** $stmt_1$ **while** (*expr*) ;	{ *stmt.n* = **new** *Do* $(stmt_1.n, expr.n)$; }
	|	*block*	{ *stmt.n* = *block.n*; }
expr	→	$rel = expr_1$	{ *expr.n* = **new** *Assign* $('=', rel.n, expr_1.n)$; }
	|	*rel*	{ *expr.n* = *rel.n*; }
rel	→	$rel_1 < add$	{ *rel.n* = **new** *Rel* $('<', rel_1.n, add.n)$; }
	|	$rel_1 <= add$	{ *rel.n* = **new** *Rel* $('\leq', rel_1.n, add.n)$; }
	|	*add*	{ *rel.n* = *add.n*; }
add	→	$add_1 + term$	{ *add.n* = **new** *Op* $('+', add_1.n, term.n)$; }
	|	*term*	{ *add.n* = *term.n*; }
term	→	$term_1 * factor$	{ *term.n* = **new** *Op* $('*', term_1.n, factor.n)$; }
	|	*factor*	{ *term.n* = *factor.n*; }
factor	→	(*expr*)	{ *factor.n* = *expr.n*; }
	|	**num**	{ *factor.n* = **new** *Num* (**num**.*value*); }

Рис. 2.39. Построение синтаксических деревьев для выражений и инструкций

ме того, мы определяем подкласс *Seq*, который представляет последовательность инструкций. Этот подкласс соответствует нетерминалу грамматики *stmts*. Каждый из упомянутых классов является подклассом *Stmt*, который, в свою очередь, является подклассом *Node*.

Схема трансляции на рис. 2.39 иллюстрирует построение узлов синтаксического дерева. Примером типичного правила может служить правило для инструкции **if**:

$$stmt \rightarrow \textbf{if}\ (\ expr\ )\ stmt_1 \quad \{\ stmt.n = \textbf{new}\ If(expr.n, stmt_1.n);\ \}$$

Значащими компонентами инструкции **if** являются *expr* и $stmt_1$. Семантическое действие определяет узел *stmt.n* как новый объект подкласса *If*. Код конструктора *If*, не показанный здесь, создает новый узел с меткой **if** и узлами *expr.n* и $stmt_1.n$ в качестве дочерних.

Выражения не начинаются с ключевого слова, так что для представления выражений, которые являются инструкциями, определен новый класс *Eval*, являющийся подклассом *Stmt*. Вот соответствующее правило:

$$stmt \rightarrow expr\ ; \quad \{\ stmt.n = \textbf{new}\ Eval(expr.n);\ \}$$

Представление блоков в синтаксических деревьях

Оставшаяся конструкция на рис. 2.39 — это блок, который состоит из последовательности инструкций. Рассмотрим правила, относящиеся к блоку:

$$\begin{array}{lll} stmt & \rightarrow block & \{\ stmt.n = block.n;\ \} \\ block & \rightarrow '\{'\ stmts\ '\}' & \{\ block.n = stmts.n;\ \} \end{array}$$

Первое правило гласит, что если инструкция представляет собой блок, то она имеет то же синтаксическое дерево, что и блок. Второе правило — синтаксическое дерево для нетерминала *block* представляет собой просто синтаксическое дерево для последовательности инструкций в блоке.

Для простоты язык, представленный на рис. 2.39, не включает объявления. В приложении А, где объявления включены в язык, можно увидеть, что синтаксическое дерево блока все равно остается синтаксическим деревом инструкций в блоке. Поскольку информация из объявлений вносится в таблицу символов, нет необходимости хранить ее в синтаксическом дереве. Таким образом, блоки, с объявлениями или без них, в промежуточном коде представляют собой всего лишь еще одну разновидность инструкции.

Последовательность инструкций представлена использованием листа **null** для пустой инструкции и оператором **seq** для последовательности инструкций:

$$stmts \rightarrow stmts_1\ stmt \quad \{\ stmts.n = \textbf{new}\ Seq(stmts_1.n, stmt.n);\ \}$$

Пример 2.18. На рис. 2.40 показана часть синтаксического дерева, представляющая блок или список инструкций. В списке содержатся две инструкции, первая из которых — конструкция **if**, а вторая — **while**. Здесь не показана часть дерева над списком, а все поддеревья представлены просто треугольниками: два дерева выражений для условий и два — для инструкций конструкций **if** и **while**. □

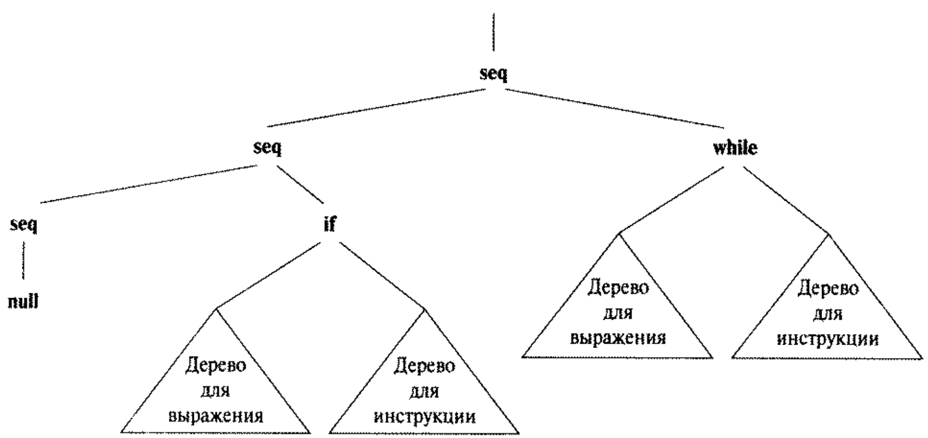


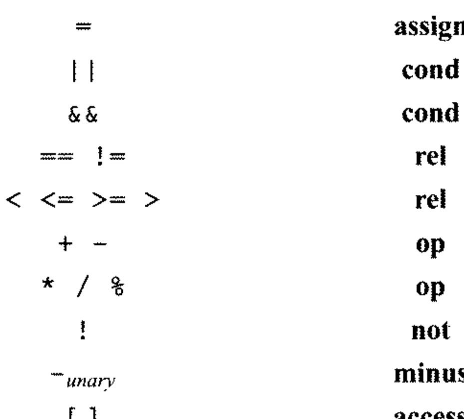
Рис. 2.40. Часть синтаксического дерева для списка инструкций, состоящего из инструкций **if** и **while**

Синтаксические деревья для выражений

Ранее мы обрабатывали более высокий приоритет умножения по сравнению со сложением при помощи трех нетерминалов *expr*, *term* и *factor*. Количество нетерминалов ровно на 1 больше количества уровней приоритетов в выражениях, как говорилось в разделе 2.2.6. На рис. 2.39, кроме обычных операторов сложения и умножения, имеются два оператора сравнения, < и <=, с одинаковым приоритетом, так что в грамматику требуется добавить еще один дополнительный нетерминал — *add*.

Абстрактный синтаксис позволяет нам сгруппировать "похожие" операторы для уменьшения количества вариантов и подклассов узлов при реализации выражений. В данной главе под "похожестью" подразумевается схожесть правил проверки типов и генерации кода для этих операторов. Например, обычно операторы + и * могут быть сгруппированы, поскольку они обрабатываются одним и тем же способом — у них одинаковые требования к типам операндов и каждый из них дает одну трехадресную команду, которая применяет один оператор к двум операндам. В общем случае группирование операторов в абстрактном синтаксисе основывается на требованиях более поздних фаз компилятора. На рис. 2.41 указано соответствие между конкретным и абстрактным синтаксисом для некоторых операторов Java.

В конкретном синтаксисе все операторы левоассоциативны (за исключением оператора присваивания =, который является правоассоциативным). Операторы в одной строке имеют одинаковые приоритеты; так, == и != имеют один и тот же приоритет. Строки таблицы приведены в порядке увеличения приоритета; например, оператор == имеет более высокий приоритет, чем операторы && и =. Нижний индекс *unary* в $-_{unary}$ использован исключительно для того, чтобы отличить ве-



Конкретный синтаксис	Абстрактный синтаксис
=	**assign**
||	**cond**
&&	**cond**
== !=	**rel**
< <= >= >	**rel**
+ -	**op**
* / %	**op**
!	**not**
$-_{unary}$	**minus**
[ ]	**access**

Рис. 2.41. Конкретный и абстрактный синтаксис некоторых операций Java

дущий унарный минус, как, например, в -2, от бинарного минуса, как, например, в 2-a. Оператор [ ] представляет обращение к массиву, как, например, в a[i].

Столбец "Абстрактный синтаксис" представляет группировку операторов. Оператор присваивания — единственный в своей группе. Группа **cond** содержит условные логические операторы && и ||. Группа **rel** включает операторы сравнения в строках, представителями которых являются == и <. Группа **op** содержит арифметические операторы наподобие + и *. Унарный минус, логическое отрицание и обращение к массиву образуют каждый свою собственную группу.

Отображение между конкретным и абстрактным синтаксисом на рис. 2.41 может быть реализовано путем написания схемы трансляции. Продукции для нетерминалов *expr*, *rel*, *add*, *term* и *factor* на рис. 2.39 определяют конкретный синтаксис для представительного подмножества операторов на рис. 2.41. Семантические действия этих продукций создают узлы синтаксического дерева. Например, правило

$$term \rightarrow term_1 * factor \quad \{\ term.n = \textbf{new}\ Op\,('*', term_1.n, factor.n);\ \}$$

создает узел класса *Op*, который реализует операторы, объединенные в группу **op** на рис. 2.41. Конструктор *Op* в дополнение к параметрам $term_1.n$ и $factor.n$ для подвыражений получает параметр $'*'$, который указывает фактический оператор.

2.8.3 Статические проверки

Статические проверки представляют собой проверки согласованности, выполняемые в процессе компиляции. Они не только гарантируют, что программа будет успешно скомпилирована, но и обладают потенциалом для раннего перехвата про-

граммных ошибок – до выполнения программы. Статические проверки включают следующее.

- *Проверки синтаксиса.* Эти проверки имеют большее отношение к синтаксису, чем к грамматике. Например, проверки выполнения таких ограничений, как то, что идентификатор должен быть объявлен в области видимости не более одного раза, или что инструкция **break** располагается в охватывающем цикле или инструкции **switch**, являются синтаксическими, хотя эти требования не кодируются и не обеспечиваются используемой грамматикой.

- *Проверки типов.* Правила типов языка гарантируют, что оператор или функция будет применена к корректному количеству операторов допустимого типа. При необходимости преобразования типов, например при сложении целого числа с числом с плавающей точкой, программа проверки типов может вставить соответствующий оператор в синтаксическое дерево. Преобразования типов будут рассмотрены позже.

L-значения и R-значения

Рассмотрим несколько простых статических проверок, которые могут быть выполнены в процессе построения синтаксического дерева для исходной программы. В общем случае сложные статические проверки могут потребовать, чтобы сначала было построено промежуточное представление, и затем они будут его анализировать.

Имеется существенная разница между смыслом идентификаторов слева и справа от оператора присваивания. В каждом из присваиваний

```
i = 5;
i = i + 1;
```

правая сторона определяет целое значение, в то время как левая – место в памяти, где это значение будет храниться. Термины *l-значение* и *r-значение* (*l-value* и *r-value*) определяют значения, которые могут использоваться в левой и правой частях инструкции присвоения соответственно. Таким образом, *r*-значения – это то, что обычно подразумевается под словом "значение", в то время как *l*-значения означают ячейки памяти.

Статическая проверка должна убедиться, что левая сторона присваивания имеет *l*-значение. Идентификатор наподобие `i` является *l*-значением, так же как и элемент массива наподобие `a[2]`. Однако константа 2 не может находиться слева от оператора присваивания, поскольку она имеет *r*-значение, но не *l*-значение.

Проверка типов

Проверка типов гарантирует, что тип конструкции соответствует ожидаемому в данном контексте. Например, в инструкции **if**

if (*expr*) *stmt*

ожидается, что выражение *expr* имеет тип **boolean**.

Правила проверки типов следуют структуре "оператор/операнд" абстрактного синтаксиса. Предположим, что оператор **rel** представляет операторы отношений, такие как <=. Правило типов для группы операторов **rel** заключается в том, что два операнда такого оператора должны иметь один и тот же тип, а результат должен иметь булев тип **boolean**. Используем атрибут *type* для типа выражения, и пусть E состоит из применения **rel** к операндам E_1 и E_2. Тип E может быть проверен после построения узла для этого выражения путем выполнения кода наподобие следующего:

if (E_1.*type* == E_2.*type*) E.*type* = boolean;
else error;

Идея соответствия фактического типа ожидаемому применима и в следующих ситуациях.

- *Приведение* (coercion). Осуществляется, когда тип операнда автоматически преобразуется в тип, требуемый оператором. В выражении наподобие `2*3.14` обычное преобразование состоит в превращении 2 в эквивалентное число с плавающей точкой `2.0`, после чего выполняется умножение двух чисел с плавающей точкой. Определение языка указывает, какие именно приведения допустимы. Например, фактическое правило для **rel**, рассматривавшееся выше, может заключаться в том, чтобы E_1.*type* и E_2.*type* могли быть приведены к одному и тому же типу. В этом случае сравнение, скажем, целого числа и числа с плавающей точкой оказывается вполне законным и корректным.

- *Перегрузка*. Оператор + в Java представляет сложение при применении к целым числам; в применении к строкам он означает их конкатенацию. Символ называется *перегруженным* (overloaded), если он имеет различные значения в зависимости от контекста. Таким образом, оператор + в Java перегружен. Значение перегруженного оператора определяется при рассмотрении известных типов его операндов и результатов. Например, мы знаем, что + в `z=x+y` представляет собой конкатенацию, если все переменные `x`, `y` и `z` имеют строковый тип. Однако если известно, что одна из переменных имеет целочисленный тип, то это означает наличие ошибки типов, и данное применение оператора + не имеет смысла.

2.8.4 Трехадресный код

После построения синтаксического дерева дальнейший анализ и синтез может быть выполнен путем вычисления атрибутов и выполнения фрагментов кода

в узлах дерева. Мы проиллюстрируем эти возможности путем обхода синтаксического дерева для генерации трехадресного кода. В частности, мы покажем, как написать функции, которые обрабатывают синтаксическое дерево и в качестве побочного результата генерируют необходимый трехадресный код.

Трехадресные команды

Трехадресный код представляет собой последовательность команд вида

$$x = y \ \mathbf{op} \ z$$

Здесь x, y и z являются именами, константами или генерируемыми компилятором временными переменными; **op** представляет оператор.

С массивами можно работать с использованием команд следующих двух видов:

$$x \ [\ y \ ] = z$$
$$x = y \ [\ z \ ]$$

Первая из них помещает значение z в ячейку памяти $x[y]$, а вторая помещает значение из ячейки памяти $y[z]$ в переменную x.

Трехадресные команды выполняются в числовой последовательности, если только иное не требуется командой условного или безусловного перехода. Для управления потоком управления используются следующие команды.

`ifFalse` x `goto L`	Если x ложно, следующей выполняется команда с меткой L
`ifTrue` x `goto L`	Если x истинно, следующей выполняется команда с меткой L
`goto L`	Следующей выполняется команда с меткой L

Метка L может быть назначена любой команде при помощи префикса `L:`. Одна команда может иметь несколько меток.

Наконец, следующая команда копирует значение y в переменную x:

$$x = y$$

Трансляция инструкций

Инструкции транслируются в трехадресный код с применением команд перехода для управления потоком выполнения. Схема на рис. 2.42 иллюстрирует трансляцию **if** (*expr*) **then** $stmt_1$. Команда условного перехода

`ifFalse` x `goto` *after*

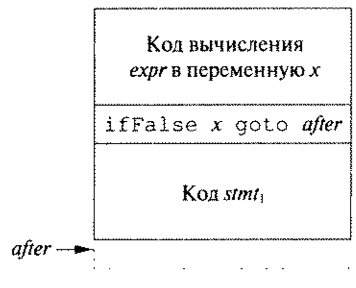


Рис. 2.42. Схема кода для конструкции if

осуществляет "прыжок" через код для вычисления $stmt_1$, если вычисленное значение *expr* ложно. Другие конструкции транслируются аналогично с использованием соответствующих переходов для обхода кода их компонентов.

Для конкретности на рис. 2.43 приведен псевдокод класса *If*. Класс *If* является подклассом класса *Stmt*, как и классы для других конструкций. Каждый подкласс *Stmt* имеет конструктор, в нашем случае — *If*, и функцию *gen*, которая вызывается для генерации трехадресного кода для данного вида инструкций.

```
class If extends Stmt {
    Expr E; Stmt S;
    public If(Expr x, Stmt y) { E = x; S = y; after = newlabel(); }
    public void gen() {
        Expr n = E.rvalue();
        emit( "ifFalse " + n.toString() + " goto " + after);
        S.gen();
        emit(after + ":");
    }
}
```

Рис. 2.43. Функция gen класса If генерирует трехадресный код

Конструктор *If* на рис. 2.43 создает узел синтаксического дерева для инструкции **if**. Он вызывается с двумя параметрами, узлом выражения x и узлом инструкции y, которые сохраняются как атрибуты E и S. Конструктор также выполняет присваивание атрибуту *after* новой уникальной метки путем вызова функции *newlabel* (). Метка будет использоваться согласно схеме на рис. 2.42.

После того, как синтаксическое дерево для исходной программы полностью построено, в его корне вызывается функция *gen*. Поскольку в нашем простом языке программа представляет собой блок, корень синтаксического дерева представ-

ляет последовательность инструкций в блоке. Все классы инструкций содержат функцию *gen*.

Псевдокод функции *gen* класса *If* на рис. 2.43 вполне представителен. Он вызывает $E.rvalue\,()$ для трансляции выражения E (выражение со значением типа **boolean**, являющееся частью конструкции **if**) и сохраняет возвращенный E узел (трансляция выражений будет рассмотрена в следующем подразделе). Затем функция *gen* генерирует условный переход и вызывает $S.gen\,()$ для трансляции подынструкции S.

Трансляция выражений

Теперь проиллюстрируем трансляцию выражений, рассматривая выражения, содержащие бинарные операторы **op**, обращение к массивам и присваивания в дополнение к константам и идентификаторам. Для простоты при рассмотрении обращений к массивам $y\,[z]$ будет требоваться, чтобы y было идентификатором.[15] Детальное обсуждение генерации промежуточного кода для выражений можно найти в разделе 6.4.

Мы воспользуемся простым подходом генерации трехадресных команд для каждого узла оператора в синтаксическом дереве выражения. Для констант и идентификаторов не генерируется никакой код, поскольку они входят в команды в качестве адресов. Если узел x класса *Expr* содержит оператор **op**, то генерируется команда для вычисления значения в узле x в генерируемое компилятором "временное" имя, скажем, t. Таким образом, выражение `i-j+k` транслируется в команды

```
t1 = i - j
t2 = t1 + k
```

При обращении к массивам и присваиваниях требуется различать l-значения и r-значения. Например, выражение `2*a[i]` может быть транслировано путем вычисления r-значения `a[i]` во временную переменную

```
t1 = a [ i ]
t2 = 2 * t1
```

Однако нельзя просто использовать временную переменную вместо `a[i]`, если `a[i]` появляется в левой части присваивания.

Этот простой подход использует две функции, *lvalue* и *rvalue*, представленные соответственно на рис. 2.44 и 2.45. При применении ко внутреннему узлу x функция *rvalue* генерирует команды для вычисления x во временную переменную и возвращает новый узел, представляющий эту переменную. При применении ко внутреннему узлу функции *lvalue* она также генерирует команды для вычисления поддеревьев ниже x и возвращает узел, представляющий "адрес" x.

[15]Наш простой язык поддерживает выражения типа `a[a[n]]`, но не `a[m][n]`. Заметим, что `a[a[n]]` имеет вид `a[`E`]`, где E представляет собой `a[n]`.

Expr lvalue(x : *Expr*) {
  if (x является узлом *Id*) **return** x;
  else if (x является узлом *Access* (y, z), а y – узел *Id*) {
    return new *Access* $(y, rvalue(z))$;
  }
  else error;
}

Рис. 2.44. Псевдокод функции *lvalue*

Рассмотрим сначала функцию *lvalue*, у которой меньше вариантов действий. При применении к узлу x функция *lvalue* просто возвращает x, если это узел идентификатора (т.е. если x принадлежит классу *Id*). В рассматриваемом простом языке единственный другой случай, когда выражение имеет l-значение, – это когда x представляет обращение к массиву, такое как `a[i]`. В этом случае x будет иметь вид $Acess\,(y, z)$, где класс *Access* является подклассом *Expr*, y представляет имя массива, к которому выполняется обращение, а z – смещение (индекс) выбранного элемента массива. В псевдокоде на рис. 2.44 функция *lvalue* вызывает $rvalue\,(z)$ для генерации необходимых команд для вычисления r-значения z. Затем она строит и возвращает новый узел *Access* с дочерними узлами для имени массива y и r-значения z.

Пример 2.19. Если узел x представляет обращение к массиву `a[2*k]`, вызов *lvalue* (x) генерирует команду

```
t = 2 * k
```

и возвращает новый узел x', представляющий l-значение `a[t]`, где `t` – новое временное имя.

Говоря более подробно, по достижении фрагмента кода

return new *Access* $(y, rvalue(z))$;

y представляет собой узел для `a`, а z – узел для выражения `2*k`. Вызов $rvalue\,(z)$ генерирует код для выражения `2*k` (т.е. трехадресную команду `t = 2 * k`) и возвращает новый узел z', представляющий временное имя `t`. Узел z' становится значением второго поля во вновь созданном узле x' типа *Access*. □

Функция *rvalue* на рис. 2.45 генерирует команды и возвращает новый узел, за исключением случая, когда узел x представляет идентификатор или константу (в этом случае функция *rvalue* возвращает сам узел x). Во всех остальных случаях функция возвращает узел *Id* для новой временной переменной t. Эти случаи перечислены ниже.

Expr rvalue(x : *Expr*) {
    if (x – узел *Id* или *Constant*) **return** x;
    else if (x – узел *Op* (**op**, y, z) или *Rel* (**op**, y, z)) {
        t = новая временная переменная;
        Генерация строки для t = *rvalue*(y) **op** *rvalue*(z);
        return Новый узел t;
    }
    else if (x узел *Access* (y, z)) {
        t = новая временная переменная;
        Вызов *lvalue*(x) возвращающий *Access* (y, z');
        Генерация строки для t = *Access* (y, z');
        return Новый узел t;
    }
    else if (x – узел *Assign* (y, z)) {
        z' = *rvalue*(z);
        Генерация строки для *lvalue*(y) = z';
        return z';
    }
}

Рис. 2.45. Псевдокод функции *rvalue*

- Если узел x представляет y **op** z, код функции сначала вычисляет $y' = rvalue(y)$ и $z' = rvalue(z)$. Он создает новую временную переменную t, генерирует команду $t = y'$ **op** z (точнее, команду, формируемую из строковых представлений t, y', **op** и z') и возвращает узел для идентификатора t.
- Если узел x представляет обращение к массиву $y[z]$, можно воспользоваться функцией *lvalue*. Вызов *lvalue* (x) возвращает обращение $y[z']$, где z' представляет идентификатор, хранящий смещение для обращения к массиву. Код создает новую временную переменную t, генерирует команду для $t = y[z']$ и возвращает узел для t.
- Если узел x представляет присваивание $y = z$, то код сначала вычисляет $z' = rvalue(z)$, затем генерирует команду для $lvalue(y) = z'$ и возвращает узел z'.

Пример 2.20. При применении синтаксического дерева для

```
a[i] = 2*a[j-k]
```

функция *rvalue* генерирует

```
t3 = j - k
t2 = a [ t3 ]
t1 = 2 * t2
a [ i ] = t1
```

Иначе говоря, корень представляет собой узел *Assign* с первым аргументом `a[i]` и вторым аргументом `2*a[j-k]`. Таким образом, оказывается применим третий случай функции *rvalue*, и она рекурсивно вычисляет `2*a[j-k]`. Корнем этого поддерева является узел *Op* для `*`, что приводит к созданию новой временной переменной `t1` перед тем, как будут вычислены левый операнд 2 и правый операнд. Константа 2 не генерирует трехадресный код, а ее r-значение, возвращаемое узлом *Constant*, равно 2.

Правый операнд `a[j-k]` представляет собой узел *Access*, который приводит к созданию новой временной переменной `t2` перед тем, как для этого узла будет вызвана функция *lvalue*. Для выражения `j-k` рекурсивно вызывается функция *rvalue*. В качестве побочного эффекта этого вызова после создания новой переменной `t3` генерируется трехадресная команда `t3 = j - k`. Затем, после возврата в вызов *lvalue* для `a[j-k]` временной переменной `t2` присваивается r-значение всего выражения присваивания, т.е. `t2 = a [ t3 ]`.

После этого происходит возврат в функцию *rvalue* для узла *Op* для умножения `2*a[j-k]`, где ранее была создана временная переменная `t1`. При вычислении этого выражения умножения в качестве побочного действия генерируется трехадресная команда `t1 = 2 * t2`. Наконец, функция *rvalue* для всего выражения вызывает функцию *lvalue* для левой части `a[i]` и генерирует трехадресную команду `a [ i ] = t1`, в которой правая часть оператора присваивания присваивается его левой части. □

Улучшение кода для выражений

Можно несколькими путями усовершенствовать функцию *rvalue* на рис. 2.45 и генерировать меньшее количество трехадресных команд.

- Уменьшить количество команд копирования в фазе последовательной оптимизации. Например, команды `t = i+1` и `i = t` могут быть объединены в `i = i+1`, если временная переменная `t` в дальнейшем не используется.
- Генерировать меньше команд с учетом контекста. Например, если левая часть трехадресного присваивания представляет собой обращение к массиву `a[t]`, то правая часть должна быть именем, константой или временной переменной — все они используют только один адрес. Но если левая часть представляет собой имя `x`, то правая часть может быть операцией `y` **op** `z`, которая использует два адреса.

Избежать некоторых команд копирования можно путем изменений в функции трансляции, чтобы она частично генерировала команду для вычисления, скажем, `j+k`, но не указывала, куда должен быть помещен результат, указывая для него адрес **null**:

null `= j + k` (2.8)

Позже нулевой адрес заменяется либо идентификатором, либо временной переменной. Идентификатор будет использован, если присваивание `j+k` находится в правой части присваивания, например `i=j+k;`. В этом случае (2.8) становится

```
i = j + k
```

Но если `j+k` является подвыражением, как в случае `j+k+1`, то адрес **null** в (2.8) заменяется новой временной переменной `t` и генерируется новая неполная команда

```
   t = j + k
null = t + 1
```

Многие компиляторы делают все возможное для генерации кода, качество которого не хуже качества ассемблерного кода, написанного вручную экспертом в этой области. При использовании методов оптимизации кода наподобие описываемых в главе 9 эффективная стратегия может состоять в применении простейшего способа генерации промежуточного кода в надежде на устранение лишних команд оптимизатором кода.

2.8.5 Упражнения к разделу 2.8

Упражнение 2.8.1. Конструкции **for** в C и Java имеют вид

`for (` $expr_1$ `;` $expr_2$ `;` $expr_3$ `)` *stmt*

Первое выражение выполняется до входа в цикл и обычно используется для инициализации счетчика цикла. Второе выражение представляет собой проверку, которая выполняется перед каждой итерацией цикла; цикл завершается, если значение этого выражения становится равным 0. Сам цикл может рассматриваться как инструкция `{`*stmt* $expr_3$`;}`. Третье выражение выполняется в конце каждой итерации и обычно используется для увеличения значения счетчика цикла. Смысл инструкции **for** аналогичен конструкции

$expr_1$`; while (` $expr_2$ `) {` *stmt* $expr_3$`; }`

Определите класс *For* для конструкции **for**, аналогичный классу *If* на рис. 2.43.

Упражнение 2.8.2. Язык программирования C не имеет булева типа. Покажите, как компилятор C может транслировать инструкцию **if** в трехадресный код.

2.9 Резюме к главе 2

Синтаксически управляемые методы из этой главы могут использоваться для создания начальной фазы компилятора, как проиллюстрировано рис. 2.46.

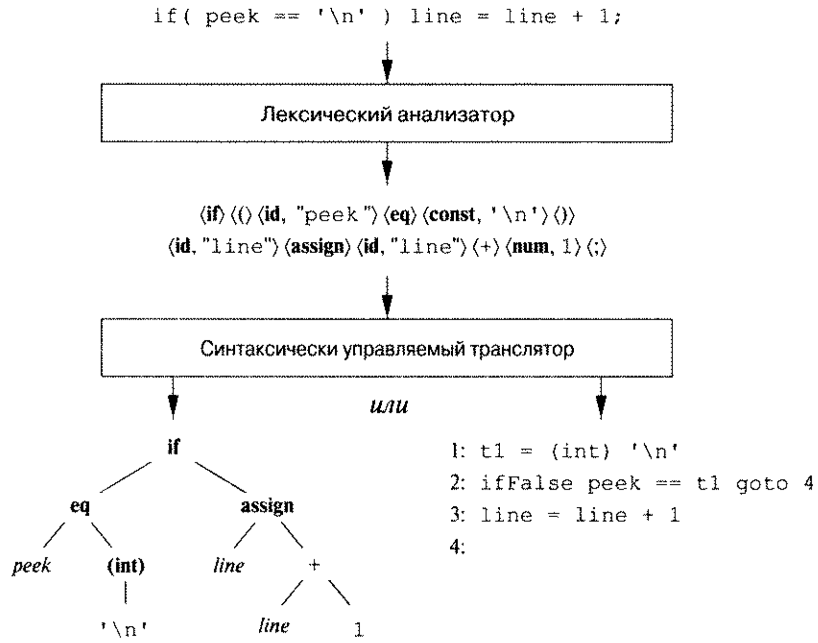


Рис. 2.46. Две возможные трансляции инструкции

- Отправной точкой для синтаксически управляемого транслятора является грамматика исходного языка. *Грамматика* описывает иерархическую структуру программы. Она определяется в терминах элементарных символов, именуемых *терминалами*, и переменных символов, именуемых *нетерминалами*. Эти символы представляют конструкции языка. Правила, или *продукции*, грамматики состоят из нетерминалов, называющихся *заголовками*, или *левыми частями* продукции, и последовательностей терминалов и нетерминалов, которые называются *телами*, или *правыми частями* продукции. Один из нетерминалов назначается *стартовым* символом.

- При определении транслятора программным конструкциям могут быть назначены *атрибуты*, которые представляют собой некоторую связанную с конструкциями информацию. Поскольку конструкции представлены символами грамматики, концепция атрибутов является их расширением. Примеры атрибутов включают целочисленные значения, связанные с термина-

лом **num**, который представляет числа, или строки, связанные с представляющим идентификаторы терминалом **id**.

- *Лексический анализатор* считывает входные данные по одному символу и выдает поток *токенов*; каждый токен состоит из терминального символа с дополнительной информацией в виде значения атрибута. На рис. 2.46 токены изображены в виде кортежей в угловых скобках ⟨⟩. Токен ⟨**id**, "peek"⟩ состоит из терминала **id** и указателя на запись в таблице символов, содержащую строку "peek". Транслятор использует таблицу для отслеживания зарезервированных слов и уже обработанных идентификаторов.

- *Синтаксический анализ*, или *разбор*, представляет собой задачу определения, каким образом строка терминалов может быть выведена из стартового символа грамматики путем многократного замещения каждого нетерминала телом одной из его продукций. Концептуально синтаксический анализатор строит дерево разбора, в котором корень помечен стартовым символом, каждый внутренний узел соответствует продукции, а каждый лист помечен либо терминалом, либо пустой строкой ϵ. Дерево разбора дает строку терминалов в листьях, считываемую слева направо.

- Эффективные синтаксические анализаторы можно создать вручную, используя нисходящий (от корня к листьям дерева разбора) метод, называющийся предиктивным синтаксическим анализом. *Предиктивный синтаксический анализатор* для каждого терминала имеет свою процедуру; тела процедур имитируют продукции для нетерминалов. Поток управления однозначно определяется при помощи просмотра одного дополнительного символа из входного потока. Другие методы синтаксического анализа рассматриваются в главе 4.

- Синтаксически управляемая трансляция выполняется путем назначения правил либо программных фрагментов продукциям грамматики. В этой главе были рассмотрены только *синтезируемые* атрибуты, значения которых в любом узле x зависят только от атрибутов в дочерних по отношению к x узлах, если таковые имеются. *Синтаксически управляемое определение* назначает продукциям правила вычисления значений атрибутов. *Схема трансляции* вставляет в тела продукций программные фрагменты, называющиеся *семантическими действиями*. Эти действия выполняются в том порядке, в котором при синтаксическом анализе используются продукции.

- Результатом синтаксического анализа является представление исходной программы, которое называется *промежуточным кодом*. На рис. 2.46 показаны два основных типа промежуточных кодов. *Абстрактное синтаксическое дерево* содержит узлы для программных конструкций; дочерние узлы

представляют значащие подконструкции. Альтернативным вариантом является *трехадресный код*, который представляет собой последовательность команд, в которой каждая команда выполняет единственную операцию.

- *Таблицы символов* являются структурами данных для хранения информации об идентификаторах. Информация помещается в таблицу символов при анализе объявления идентификатора. При последующем использовании идентификатора, например в качестве операнда в выражении, семантическое действие получает информацию о нем из таблицы символов.

Глава 3

Лексический анализ

В этой главе мы рассмотрим создание лексического анализатора. Для реализации лексического анализатора вручную сто́ит начать с диаграммы или иного описания лексем каждого токена. Затем можно написать код, который будет идентифицировать каждую встреченную во входном потоке лексему и возвращать информацию об обнаруженном токене.

Можно также получить лексический анализатор автоматически, определив шаблоны лексем для *генератора лексических анализаторов* и компилируя их в код, функционирующий в качестве лексического анализатора. Данный подход упрощает внесение изменений в лексический анализатор, поскольку для этого требуется переписать только измененные шаблоны, но не весь код программы. Он также ускоряет процесс реализации лексического анализатора, поскольку программист при этом работает только с очень высокоуровневыми шаблонами, а детальный код получается в результате работы генератора лексических анализаторов. В разделе 3.5 вы познакомитесь с одним из таких генераторов, который называется *Lex* (или *Flex* в более поздних вариантах).

Изучение генераторов лексических анализаторов начнется с регулярных выражений, удобной записи для определения шаблонов лексем. Вы узнаете, как можно преобразовать эту запись сначала в недетерминированный, а затем в детерминированный автомат. Последние две записи могут использоваться в качестве входных данных для "драйвера", т.е. кода, который имитирует работу этих автоматов и использует их в качестве руководства для определения следующего токена во входном потоке. Этот драйвер и спецификация автомата образуют ядро лексического анализатора.

3.1 Роль лексического анализатора

Поскольку лексический анализатор представляет собой первую фазу компилятора, его основная задача состоит в чтении входных символов исходной программы, их группировании в лексемы и вывод последовательностей токенов для всех лексем исходной программы. Поток токенов пересылается синтаксическому анализатору для разбора. Обычно при работе лексический анализатор взаимодействует также с таблицей символов. Когда лексический анализатор встречается

с лексемой, составляющей идентификатор, эту лексему требуется внести в таблицу символов. В некоторых случаях лексический анализатор может получать из таблицы символов некоторую информацию об идентификаторах, которая может помочь ему верно определить передаваемый синтаксическому анализатору токен.

Схема описанного взаимодействия представлена на рис. 3.1. Обычно взаимодействие реализуется как вызов лексического анализатора синтаксическим анализатором. Этот вызов, представленный как команда *getNextToken*, заставляет лексический анализатор читать символы из входного потока, пока он не сможет идентифицировать очередную лексему и вернуть синтаксическому анализатору корректный токен.

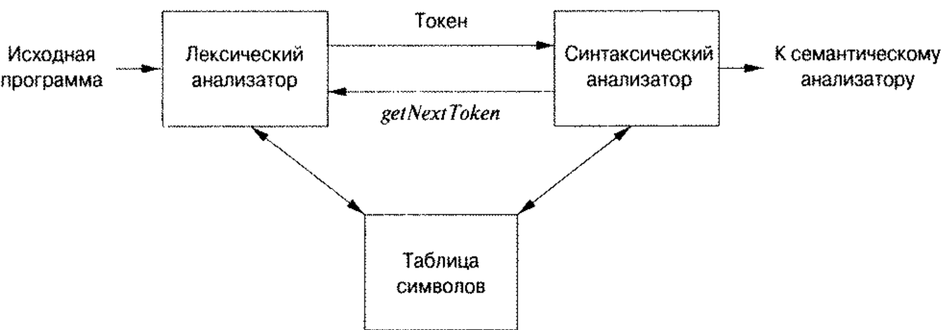


Рис. 3.1. Взаимодействие лексического анализатора с синтаксическим

Поскольку лексический анализатор является частью компилятора, которая читает исходный текст, он может заодно выполнять и некоторые другие действия, помимо идентификации лексем. Одной из таких задач является отбрасывание комментариев и *пробельных символов* (пробел, символы табуляции и новой строки, а также, возможно, некоторые другие символы, использующиеся для отделения токенов друг от друга во входном потоке). Еще одной задачей является синхронизация сообщений об ошибках, генерируемых компилятором, с исходной программой. Например, лексический анализатор может отслеживать количество символов новой строки, чтобы каждое сообщение об ошибке сопровождалось номером строки, в которой она обнаружена. В некоторых компиляторах лексический анализатор создает копию исходной программы с сообщениями об ошибках, вставленными в соответствующие места исходного текста. Если исходная программа использует макропрепроцессор, то раскрытие макросов также может выполняться лексическим анализатором.

Иногда лексические анализаторы разделяются на соединенные каскадом два процесса.

а) *Сканирование* состоит из простых процессов, не требующих токенизации входного потока, таких как удаление комментариев и уплотнение последовательностей пробельных символов в один.

б) Собственно *лексический анализ* — более сложная часть, которая генерирует токены из выходного потока сканера.

3.1.1 Лексический и синтаксический анализ

Имеется ряд причин, по которым фаза анализа компиляции разделяется на лексический и синтаксический анализ.

1. Наиболее важной причиной является упрощение разработки. Отделение лексического анализатора от синтаксического часто позволяет упростить как минимум одну из фаз анализа. Например, включить в синтаксический анализатор работу с комментариями и пробельными символами существенно сложнее, чем удалить их лексическим анализатором. При создании нового языка разделение лексических и синтаксических правил может привести к более четкому и ясному построению языка.

2. Увеличивается эффективность компилятора. Отдельный лексический анализатор позволяет применять более специализированные методики, предназначенные исключительно для решения лексических задач. Кроме того, применение методов буферизации для чтения входного потока может существенно увеличить производительность компилятора.

3. Увеличивается переносимость компилятора. Особенности входных устройств могут ограничивать возможности лексического анализатора.

3.1.2 Токены, шаблоны и лексемы

При рассмотрении лексического анализа используются три связанных, но различных термина.

- *Токен* представляет собой пару, состоящую из имени токена и необязательного атрибута. Имя токена — это абстрактный символ, представляющий тип лексической единицы, например конкретное ключевое слово или последовательность входных символов, составляющую идентификатор. Имена токенов являются входными символами, обрабатываемыми синтаксическим анализатором. Далее обычно мы будем записывать имя токена полужирным шрифтом и ссылаться на токен по его имени.

- *Шаблон* (pattern) — это описание вида, который может принимать лексема токена. В случае ключевого слова шаблон представляет собой просто

последовательность символов, образующих это ключевое слово. Для идентификаторов и некоторых других токенов шаблон представляет собой более сложную структуру, которой *соответствуют* (matched) многие строки.

- *Лексема* представляет собой последовательность символов исходной программы, которая соответствует шаблону токена и идентифицируется лексическим анализатором как экземпляр токена.

Пример 3.1. На рис. 3.2 приведены некоторые типичные токены, неформальное описание их шаблонов и некоторые примеры лексем. Чтобы увидеть использование этих концепций на практике, в инструкции на языке программирования C

```
printf("Total = %d\n", score);
```

`printf` и `score` представляют собой лексемы, соответствующие токену **id**, а `"Total = %d\n"` является лексемой, соответствующей токену **literal**. □

Токен	Неформальное описание	Примеры лексем
if	Символы `i`, `f`	`if`
else	Символы `e`, `l`, `s`, `e`	`else`
comparison	< или > или <= или >= или == или !=	<=, !=
id	Буква, за которой следуют буквы и цифры	`pi`, `score`, `D2`
number	Любая числовая константа	`3.14159`, `0`, `6.02e23`
literal	Все, кроме ", заключенное в двойные кавычки	`"core dumped"`

Рис. 3.2. Примеры токенов

Во многих языках программирования описанные далее ситуации охватывают большинство (если не все) токенов.

1. По одному токену для каждого ключевого слова. Шаблон для ключевого слова выглядит так же, как само ключевое слово.

2. Токены для операторов, либо отдельные, либо объединенные в классы, как это сделано в случае токена **comparison** на рис. 3.2.

3. Один токен, представляющий идентификаторы.

4. Один или несколько токенов, представляющих константы, такие как числа и строковые литералы.

5. Токены для каждого символа пунктуации, такие как левые и правые скобки, запятые или точки с запятыми.

3.1.3 Атрибуты токенов

Если шаблону могут соответствовать несколько лексем, лексический анализатор должен обеспечить дополнительную информацию о лексемах для последующих фаз компиляции. Например, шаблон для токена **number** соответствует как строке 0, так и строке 1, и при генерации кода крайне важно знать, какая именно лексема была найдена в исходной программе. Таким образом, во многих случаях лексический анализатор возвращает синтаксическому анализатору не только имя токена, но и значение атрибута, описывающее лексему, представляющую токен; имя токена влияет на принятие решений при синтаксическом анализе, а значение атрибута — на трансляцию токена после синтаксического анализа.

Мы считаем, что токены имеют не более одного связанного с ними атрибута, хотя этот атрибут может представлять собой структуру, объединяющую несколько блоков информации. Наиболее важным примером может служить токен **id**, с которым необходимо связывать большое количество информации. Обычно информация об идентификаторе — например, его лексема, тип и место в исходной программе, где он впервые встретился (эта информация может потребоваться, например, для сообщения об ошибке) — хранится в таблице символов. Таким образом, подходящим значением атрибута для идентификатора является указатель на запись в таблице символов для данного идентификатора.

Пример 3.2. Имена токенов и связанные с ними атрибуты в инструкции на языке программирования Fortran

```
E = M * C ** 2
```

приведены ниже как последовательность пар:

$\langle$**id**, Указатель на запись в таблице символов для E$\rangle$

$\langle$**assign_op**$\rangle$

$\langle$**id**, Указатель на запись в таблице символов для M$\rangle$

$\langle$**mult_op**$\rangle$

$\langle$**id**, Указатель на запись в таблице символов для C$\rangle$

$\langle$**exp_op**$\rangle$

$\langle$**number**, Целое значение 2$\rangle$

Обратите внимание, что в ряде пар, в частности для операторов, знаков препинания и ключевых слов, атрибуты излишни. В нашем примере токен **number** имеет целочисленный атрибут. На практике типичный компилятор вместо этого хранит строку символов, представляющую константу и в качестве атрибута для **number** использует указатель на нее. □

Тонкости распознавания токенов

Обычно для имеющихся шаблонов, описывающих лексемы токенов, выполнить распознавание лексем во входном потоке относительно просто. Однако в некоторых языках, встретив лексему, невозможно сразу же сказать, какому токену она соответствует. Приведенный далее пример взят из языка программирования Fortran, фиксированный формат в котором все еще разрешен в Fortran 90. В инструкции

```
DO 5 I = 1.25
```

до тех пор, пока мы не встретим десятичную точку после `1`, мы не можем утверждать, что первая лексема — `DO5I`, являющаяся экземпляром токена идентификатора (согласно архаичному соглашению в фиксированном формате Fortran символы пробела игнорируются). Если вместо точки окажется запятая, мы получим инструкцию `DO`

```
DO 5 I = 1,25
```

В ней первая лексема представляет собой ключевое слово `DO`.

3.1.4 Лексические ошибки

Без помощи других компонентов компилятора лексическому анализатору сложно обнаружить ошибки в исходном тексте программы. Например, если в программе на языке С строка `fi` впервые встретится в контексте

```
fi ( a == f(x)) ...
```

лексический анализатор не сможет определить, что именно представляет собой `fi` — неверно записанное слово `if` или необъявленный идентификатор функции. Поскольку `fi` является корректной лексемой для токена **id**, лексический анализатор должен вернуть этот токен синтаксическому анализатору и позволить другой фазе компилятора — в данном случае, по всей видимости, синтаксическому анализатору — обработать ошибку перестановки местами двух букв.

Однако представим ситуацию, когда лексический анализатор не способен продолжать работу, поскольку ни один из шаблонов не соответствует префиксу оставшегося входного потока. Простейшей стратегией в этой ситуации будет восстановление в "режиме паники". Мы просто удаляем входные символы до тех пор, пока лексический анализатор не встретит распознаваемый токен в начале оставшейся входной строки. Этот метод восстановления может запутать синтаксический

анализатор, но для интерактивной среды данная методика может быть вполне адекватной.

Существуют и другие возможные действия по восстановлению после ошибки.

1. Удаление одного символа из оставшегося входного потока.
2. Вставка пропущенного символа в оставшийся входной поток.
3. Замена символа другим.
4. Перестановка двух соседних символов.

Преобразования наподобие перечисленных пытаются исправить входной поток. Простейшая стратегия исправления состоит в выяснении, не может ли префикс оставшейся входной строки быть преобразован в корректную лексему при помощи одного действия. Такая стратегия имеет смысл, поскольку на практике большинство лексических ошибок вызываются единственным символом. Более глобальная стратегия состоит в поиске наименьшего количества трансформаций, необходимых для преобразования исходной программы в программу, состоящую только из корректных лексем, но на практике это слишком дорогой подход, не оправдывающий затрачиваемых усилий.

3.1.5 Упражнения к разделу 3.1

Упражнение 3.1.1. Разделите приведенный ниже фрагмент на языке программирования C на лексемы, используя материал раздела 3.1.2 в качестве руководства. Какие из лексем должны сопровождаться связанными лексическими значениями? Каковы эти значения?

```
float limitedSquare(x) float x {
    /* Возвращает x в квадрате, но не более 100 */
    return (x<=-10.0||x>=10.0)?100:x*x;
}
```

! **Упражнение 3.1.2.** Языки разметки наподобие HTML и XML отличаются от обычных языков программирования тем, что либо количество дескрипторов в них очень велико (как в HTML), либо дескрипторы определяются пользователем (как в XML). Кроме того, дескрипторы часто могут иметь параметры. Предложите разделение приведенного далее фрагмента HTML-документа на лексемы. Какие лексемы должны быть со связанными лексическими значениями и какими должны быть эти значения?

```
Это фотография <B>моего дома</B>:
<P><IMG SRC = "house.gif"><BR>
```

```
Если она вас заинтересовала, можете посмотреть
<A HREF = "morePix.html">и другие фотографии</A>
наподобие этой.<P>
```

3.2 Буферизация ввода

Перед тем как перейти к распознаванию лексем во входном потоке, рассмотрим несколько способов ускорения простой, но важной задачи — чтения исходной программы. Эта задача усложняется тем, что нам часто надо "заглянуть" на один или несколько символов вперед, за очередную лексему, чтобы убедиться в корректности распознавания. Пример, приведенный во врезке "Тонкости распознавания токенов" в разделе 3.1, конечно, экстремален, но имеется множество ситуаций, когда требуется просмотреть как минимум один символ за текущим считанным символом. Например, нельзя быть уверенным, что достигнут конец идентификатора, пока не встретится символ, не являющийся ни буквой, ни цифрой, а значит, не являющийся частью лексемы для токена **id**. В C односимвольный оператор наподобие -, = или < может быть началом двухсимвольного оператора наподобие ->, == или <=. Таким образом, мы начнем с двухбуферной схемы, способной безопасно работать с большими предпросмотрами. Затем будет рассмотрено усовершенствование, заключающееся в применении "ограничителей", экономящих время проверки концов буферов.

3.2.1 Пары буферов

Поскольку для больших исходных программ количество времени, затрачиваемого на обработку всех их символов, может быть достаточно большим, для уменьшения накладных расходов на обработку одного входного символа были разработаны специальные буферизующие методы. Одна из важных схем буферизации включает два по очереди загружаемых буфера, как показано на рис. 3.3.

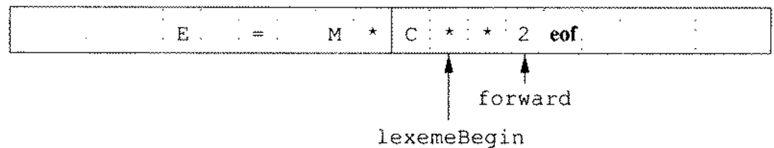


Рис. 3.3. Использование пары входных буферов

Оба буфера имеют один и тот же размер N, причем N обычно равно размеру дискового блока, например 4096 байт. При этом можно считать N символов в буфер одной командой чтения, не используя системный вызов для каждого символа по отдельности. Если во входном файле осталось менее N символов, конец ис-

ходного файла маркируется специальным символом **eof**, отличным от любого из возможных символов исходной программы.

При этом поддерживаются два указателя.

1. Указатель `lexemeBegin`, маркирующий начало текущей лексемы, протяженность которой мы пытаемся определить.

2. Указатель `forward`, который сканирует символы до тех пор, пока выполняется соответствие шаблону. Точная стратегия проверки выполнения соответствия рассматривается ниже в этой главе.

Как только определена очередная лексема, указатель `forward` устанавливается таким образом, чтобы указывать на символ, являющийся ее правым концом. Затем, после того так лексема записана в качестве значения атрибута токена, возвращаемого синтаксическому анализатору, указатель `lexemeBegin` устанавливается на символ, непосредственно следующий за только что обнаруженной лексемой. На рис. 3.3 указатель `forward` указывает на символ за концом текущей лексемы `**` (оператор возведения в степень в языке программирования Fortran) и должен быть перемещен на одну позицию влево.

При перемещении указателя `forward` первоначально необходимо выполнить проверку, не достигнут ли конец одного из буферов, и, если достигнут, заполнить другой буфер символами из входного потока, и перенести `forward` в начало этого только что заполненного буфера. Если нам не потребуется просматривать столько символов за фактической лексемой, что сумма длины лексемы и количества просматриваемых символов превышает N, то перезаписывания лексемы в буфере до ее определения не произойдет.

3.2.2 Ограничители

Если воспользоваться схемой из раздела 3.2.1 в том виде, в котором она описана, всякий раз при перемещении указателя `forward` придется проверять, не выходит ли он за пределы буфера, и, если выходит, загружать содержимое второго буфера. Таким образом, для каждого считанного символа необходимо выполнить два теста: один — не достигнут ли конец буфера, второй — какой именно символ считан (здесь возможно дальнейшее ветвление). Однако проверку конца буфера можно совместить с проверкой считанного символа, если расширить каждый буфер для хранения в его конце специального *ограничителя* (sentinel). Ограничитель представляет собой специальный символ, который не может быть частью исходной программы; естественным выбором является использование в качестве ограничителя символа **eof**.

На рис. 3.4 показана та же ситуация, что и на рис. 3.3, но с добавленными ограничителями. Обратите внимание, что символ **eof** используется не только в качестве ограничителя, но и, как и ранее, в качестве маркера конца входного потока.

Может ли не хватить размера буфера?

В большинстве современных языков лексемы достаточно коротки, и одного или двух символов предпросмотра вполне достаточно. Таким образом, размера буфера N порядка тысяч символов вполне достаточно и схема с двумя буферами из раздела 3.2.1 работает без каких-либо проблем. Однако имеются и рискованные ситуации. Например, если символьные строки могут быть очень длинными, распространяющимися на много строк исходного текста, то мы можем оказаться в ситуации, когда длина лексемы превысит N. Чтобы избежать такой проблемы с длинными строками, их можно рассматривать как конкатенацию компонентов, по одному на каждую строку исходного текста. Например, в Java удобно представлять длинные строки путем записи в каждой строке исходного текста части большой строки с оператором + в конце.

Более сложная проблема возникает при необходимости предпросмотра на произвольную глубину. Например, некоторые языки наподобие PL/I не рассматривают ключевые слова как *зарезервированные*; это значит, что можно использовать идентификаторы с именами, совпадающими с ключевыми словами наподобие `DECLARE`. Если лексический анализатор встречает исходный текст программы на языке программирования PL/I, который начинается с `DECLARE ( ARG1, ARG2, ...`, он не в состоянии определить, является ли `DECLARE` ключевым словом, а `ARG1` и далее — объявляемыми переменными, или это процедура с именем `DECLARE` и ее аргументы. По этой причине современные языки программирования тяготеют к резервированию своих ключевых слов. В указанной ситуации, однако, можно рассматривать ключевое слово `DECLARE` как неоднозначный идентификатор и позволить решить этот вопрос синтаксическому анализатору (вероятно, с использованием поиска в таблице символов).

На рис. 3.5 приведена схема алгоритма перемещения указателя `forward`. Обратите внимание на то, что для каждого символа, на который указывает `forward`, выполняется только один тест, приводящий к множественному ветвлению. Ис-

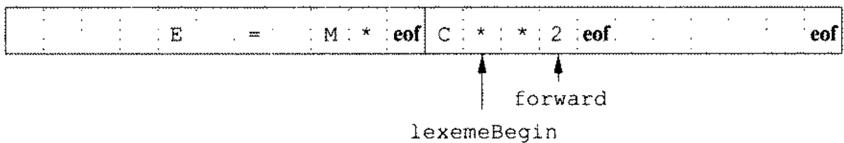


Рис. 3.4. Ограничитель в конце каждого буфера

ключением является только случай, когда указатель достигает конца буфера или конца входного потока.

```
switch ( *forward++ ) {
    case eof:
        if (forward в конце первого буфера ) {
            Загрузка второго буфера;
            forward = начало второго буфера;
        }
        else if (forward в конце второго буфера ) {
            Загрузка первого буфера;
            forward = начало первого буфера;
        }
        else /* eof в буфере маркирует конец входного потока */
            Завершение лексического анализа;
        break;
    Действия для прочих символов
}
```

Рис. 3.5. Предпросмотр с применением ограничителей

3.3 Спецификация токенов

Регулярные выражения представляют собой важный способ записи спецификации шаблонов лексем. Хотя они и не в состоянии выразить все возможные шаблоны, тем не менее регулярные выражения очень эффективны при определении реально используемых для токенов типов шаблонов. В этом разделе мы изучим формальную запись регулярных выражений, а в разделе 3.5 узнаем, как они используются в генераторе лексических анализаторов. Затем в разделе 3.7 будет показано, как построить лексический анализатор путем преобразования регулярных выражений в автомат для распознавания конкретных токенов.

3.3.1 Строки и языки

Термин *алфавит* означает любое конечное множество символов. Типичными примерами символов могут служить буквы, цифры и знаки препинания. Множество $\{0, 1\}$ представляет собой *бинарный алфавит*. Важным примером алфавита является код ASCII, использующийся во многих программных системах. Unicode, включающий примерно 100 000 символов из алфавитов всего мира, — еще один важный пример алфавита.

Реализация множественного ветвления

Может показаться, что инструкция `switch` на рис. 3.5 требует для выполнения много шагов и помещение первым случая **eof** — не самый разумный выбор. В действительности порядок перечисления конструкций `case` значения не имеет. На практике множественное ветвление, зависящее от значения символа, выполняется за один шаг путем перехода по адресу, указанному в проиндексированной символами таблице адресов.

Строка над некоторым алфавитом — это конечная последовательность символов, взятых из этого алфавита. В теории языков термины *предложение* (sentence) и *слово* (word) часто используются как синонимы термина *строка* (string). Длина строки s, обычно обозначаемая как $|s|$, равна количеству символов в строке. Например, длина строки `banana` равна шести. *Пустая* строка, обозначаемая как ϵ, представляет собой строку нулевой длины.

Язык представляет собой любое счетное множество строк над некоторым фиксированным алфавитом. Это определение весьма широко. Абстрактные языки наподобие $\emptyset$ (пустое множество) и $\{\epsilon\}$ (множество, содержащее только пустую строку) также являются языками согласно этому определению. Точно так же языками являются и множество всех синтаксически корректных программ на языке C, и все грамматически корректные предложения английского языка, хотя два последних множества существенно сложнее точно определить. Обратите внимание, что определение "языка" не требует, чтобы строкам языка был приписан какой-либо смысл. Методы описания "смысла" строк обсуждаются в главе 5.

Если x и y — строки, то *конкатенация* строк x и y, записываемая как xy, является строкой, образованной путем добавления y к x. Например, если $x =$ `dog`, а $y =$ `house`, то $xy =$ `doghouse`. Пустая строка представляет собой единичный элемент по отношению к операции конкатенации, т.е. для любой строки s справедливо соотношение $\epsilon s = s\epsilon = s$.

Если рассматривать конкатенацию как умножение, то можно определить "возведение в степень" следующим образом. Определим s^0 как ϵ, а для всех $i > 0$ определим s^i как $s^{i-1}s$. Поскольку $\epsilon s = s$, то $s^1 = s$. Соответственно, $s^2 = ss$, $s^3 = sss$ и т.д.

3.3.2 Операции над языками

В лексическом анализе наиболее важными операциями над языками являются объединение, конкатенация и замыкание, формально определенные на рис. 3.6. Объединение — привычная операция над множествами. Конкатенация языков

Термины для частей строк

Обычно используются следующие связанные со строками термины.

1. *Префиксом* (prefix) строки s является любая строка, полученная удалением нуля или нескольких последних символов строки s. Например, `ban`, `banana` и ϵ являются префиксами строки `banana`.

2. *Суффиксом* (suffix) строки s является любая строка, полученная удалением нуля или нескольких первых символов строки s. Например, `nana`, `banana` и ϵ являются суффиксами строки `banana`.

3. *Подстрока* (substring) строки s получается путем удаления произвольного префикса и произвольного суффикса из строки s. Например, `banana`, `nan` и ϵ являются подстроками `banana`.

4. *Правильными*, или *истинными* (proper), префиксами, суффиксами и подстроками s являются соответственно префиксы, суффиксы и подстроки s, которые не являются пустыми строками ϵ и не совпадают с самой строкой s.

5. *Подпоследовательностью* (subsequence) строки s является любая строка, образованная удалением нуля или нескольких не обязательно смежных позиций из строки s. Например, `baan` является подпоследовательностью `banana`.

представляет собой все строки, образованные путем взятия строки из первого языка и строки из второго языка всеми возможными способами с их последующей конкатенацией. *Замыкание Клини* языка L, обозначаемое как L^*, представляет собой множество строк, которые можно получить путем конкатенации L нуль или несколько раз. Заметим, что L^0, "конкатенация L нуль раз", по определению равна $\{\epsilon\}$ и по индукции L^i равно $L^{i-1}L$. Наконец, положительное замыкание, обозначаемое как L^+, представляет собой то же, что и замыкание Клини, но без члена L^0, т.е. ϵ отсутствует в L^+, если только оно не является частью самого L.

Пример 3.3. Пусть L — множество букв $\{\texttt{A}, \texttt{B}, \ldots, \texttt{Z}, \texttt{a}, \texttt{b}, \ldots, \texttt{z}\}$, а D — множество цифр $\{\texttt{0}, \texttt{1}, \ldots \texttt{9}\}$. Можно рассматривать множества L и D двумя по сути одинаковыми способами. С одной стороны, L и D представляют собой алфавиты, состоящие соответственно из прописных и строчных букв и из цифр. С другой стороны, множества L и D представляют собой языки, все строки которых имеют

Операция	Определение и обозначение
Объединение (union) L и M	$L \cup M = \{s \mid s \in L$ или $s \in M\}$
Конкатенация (concatenation) L и M	$LM = \{st \mid s \in L$ и $t \in M\}$
Замыкание Клини (Kleene closure) языка L	$L^* = \bigcup_{i=0}^{\infty} L^i$
Позитивное замыкание (positive closure) языка L	$L^+ = \bigcup_{i=1}^{\infty} L^i$

Рис. 3.6. Определения операций над языками

единичную длину. Вот несколько примеров других языков, построенных из L и D с применением операторов, приведенных на рис. 3.6.

1. $L \cup D$ представляет собой множество букв и цифр — строго говоря, язык с 62 строками единичной длины, каждая из которых состоит либо из одной буквы, либо из одной цифры.

2. LD — множество из 520 строк длиной 2, каждая из которых состоит из буквы, за которой следует цифра.

3. L^4 — множество всех четырехбуквенных строк.

4. L^* — множество всех строк из букв, включая пустую строку ϵ.

5. $L(L \cup D)^*$ — множество всех строк из букв и цифр, начинающихся с буквы.

6. D^+ — множество всех строк из одной или нескольких цифр. □

3.3.3 Регулярные выражения

Предположим, что мы хотим описать множество корректных идентификаторов C. Они почти точно описаны в п. 5 предыдущего подраздела; единственная неточность в том, что в множество букв включается также символ подчеркивания.

В примере 3.3 нам удалось описать идентификаторы, давая имена множествам букв и цифр и используя языковые операторы объединения, конкатенации и замыкания. Этот процесс оказался настолько практичным, что система обозначений, называющаяся *регулярными выражениями*, стала широко использоваться для описания всех языков, которые могут быть построены путем применения указанных операторов к символам некоторого алфавита. В этих обозначениях, если *letter_* означает любую букву или подчеркивание, а *digit* — любую цифру, мы можем описать идентификаторы языка программирования C как

$$letter_(letter_ \mid digit)^*$$

Вертикальная черта здесь означает объединение, скобки используются для группирования подвыражений, звездочка означает "нуль или несколько вхождений", а непосредственное соседство *letter_* с остальной частью выражения означает конкатенацию.

Регулярные выражения рекурсивно строятся из меньших регулярных выражений с использованием описанных ниже правил. Каждое регулярное выражение r описывает язык $L(r)$, который также рекурсивно определяется на основании языков, описываемых подвыражениями r. Вот правила, которые определяют регулярные выражения над некоторым алфавитом Σ, и языки, описываемые этими регулярными выражениями.

БАЗИС: Базис образован двумя правилами.

1. ϵ является регулярным выражением, а $L(\epsilon)$ представляет собой $\{\epsilon\}$, т.е. язык, единственный член которого — пустая строка.

2. Если a — символ в Σ, то $\mathbf{a}$ представляет собой регулярное выражение, а $L(\mathbf{a}) = \{a\}$, т.е. язык с одной строкой единичной длины, с символом a в единственной позиции. Заметим, что по соглашению для символов используется курсив, а для соответствующих им регулярных выражений — полужирный шрифт.[1]

ИНДУКЦИЯ: Имеется четыре правила индукции, посредством которых регулярные выражения строятся из более мелких. Предположим, что r и s являются регулярными выражениями, описывающими соответственно языки $L(r)$ и $L(s)$.

1. $(r) \mid (s)$ — регулярное выражение, описывающее язык $L(r) \cup L(s)$.

2. $(r)(s)$ — регулярное выражение, описывающее язык $L(r)\,L(s)$.

3. $(r)^*$ — регулярное выражение, описывающее язык $(L(r))^*$.

4. (r) — регулярное выражение, описывающее язык $L(r)$. Это правило говорит о том, что можно заключить выражение в скобки без изменения описываемого им языка.

Как было сказано, регулярные выражения часто содержат лишние пары скобок. Эти скобки можно опустить, если принять следующие соглашения.

а) Унарный оператор $*$ левоассоциативен и имеет наивысший приоритет.

б) Конкатенация имеет второй по величине приоритет и также левоассоциативна.

[1]Однако, говоря о конкретных символах из набора ASCII, мы обычно будем использовать моноширинный шрифт как для символов, так и для их регулярных выражений.

в) Оператор | левоассоциативен и имеет наименьший приоритет.

При этих соглашениях, например, можно заменить регулярное выражение $(\mathbf{a}) \mid ((\mathbf{b})^* (\mathbf{c}))$ выражением $\mathbf{a} \mid \mathbf{b}^*\mathbf{c}$. Оба выражения описывают множество строк, которые представляют собой либо единственный символ a, либо нуль или несколько b, за которыми следует единственный c.

Пример 3.4. Пусть $\Sigma = \{a, b\}$.

1. Регулярное выражение $\mathbf{a} \mid \mathbf{b}$ описывает язык $\{a, b\}$.

2. Регулярное выражение $(\mathbf{a} \mid \mathbf{b})(\mathbf{a} \mid \mathbf{b})$ описывает $\{aa, ab, ba, bb\}$ – множество всех строк из a и b длиной 2 над алфавитом Σ. Другое регулярное выражение для того же языка – $\mathbf{aa} \mid \mathbf{ab} \mid \mathbf{ba} \mid \mathbf{bb}$.

3. Регулярное выражение $\mathbf{a}^*$ описывает язык, состоящий из всех строк из нуля или более a, т.е. $\{\epsilon, a, aa, aaa, \ldots\}$.

4. Регулярное выражение $(\mathbf{a} \mid \mathbf{b})^*$ описывает множество всех строк, состоящих из нуля или нескольких экземпляров a или b, т.е. все строки из a и b: $\{\epsilon, a, b, aa, ab, ba, bb, aaa, \ldots\}$. Другое регулярное выражение для того же языка – $(\mathbf{a}^*\mathbf{b}^*)^*$.

5. Регулярное выражение $\mathbf{a} \mid \mathbf{a}^*\mathbf{b}$ описывает язык $\{a, b, ab, aab, aaab, \ldots\}$, т.е. строку a и все строки, состоящие из нуля или нескольких a и завершающиеся b. □

Язык, который может быть определен регулярным выражением, называется *регулярным множеством* (regular set). Если два регулярных выражения r и s описывают один и тот же язык, то r и s называются *эквивалентными*, что записывается как $r = s$. Например, $(\mathbf{a} \mid \mathbf{b}) = (\mathbf{b} \mid \mathbf{a})$. Имеется ряд алгебраических законов для регулярных выражений; каждый такой закон заключается в утверждении об эквивалентности двух разных видов регулярных выражений. На рис. 3.7 приведены некоторые из этих алгебраических законов для произвольных регулярных выражений r, s и t.

3.3.4 Регулярные определения

Для удобства записи определенным регулярным выражениям можно присваивать имена и использовать их в последующих выражениях так, как если бы это были символы. Если Σ является алфавитом базовых символов, то *регулярное*

Закон	Описание
$r \mid s = s \mid r$	Оператор | коммутативен
$r \mid (s \mid t) = (r \mid s) \mid t$	Оператор | ассоциативен
$r\,(st) = (rs)\,t$	Конкатенация ассоциативна
$r\,(s \mid t) = rs \mid rt$; $(s \mid t)\,r = sr \mid tr$	Конкатенация дистрибутивна над |
$\epsilon r = r\epsilon = r$	ϵ является единичным элементом по отношению к конкатенации
$r^* = (r \mid \epsilon)^*$	ϵ гарантированно входит в замыкание
$r^{**} = r^*$	Оператор * идемпотентен

Рис. 3.7. Алгебраические законы для регулярных выражений

определение представляет собой последовательность определений вида

$$\begin{array}{ccc} d_1 & \to & r_1 \\ d_2 & \to & r_2 \\ & \cdots & \\ d_n & \to & r_n \end{array}$$

Здесь

1) каждое d_i – новый символ, не входящий в Σ и не совпадающий ни с каким другим d;

2) каждое r_i – регулярное выражение над алфавитом $\Sigma \cup \{d_1, d_2, \ldots, d_{i-1}\}$.

Ограничивая каждое r_i символами из Σ и ранее определенными именами, можно избежать рекурсивных определений и построить регулярное выражение для любого r_i только над Σ. Для этого сначала в r_2 выполняется замена всех вхождений d_1 (в r_2 не могут использоваться никакие иные d), затем в r_3 выполняется замена d_1 и d_2 на r_1 и (уже модифицированное) r_2 и т.д. Наконец, в r_n заменяются все d_i, $i = 1, 2, \ldots, n-1$ модифицированными версиями r_i, каждая из которых содержит только символы из алфавита Σ.

Пример 3.5. Идентификаторы C представляют собой строки из букв, цифр и символов подчеркиваний. Далее приведено регулярное определение идентификаторов языка программирования C (для символов, определенных в регулярных определениях, использован курсив):

$$\begin{array}{rcl} letter_ & \rightarrow & \texttt{A} \mid \texttt{B} \mid \cdots \mid \texttt{Z} \mid \texttt{a} \mid \texttt{b} \mid \cdots \mid \texttt{z} \mid _ \\ digit & \rightarrow & \texttt{0} \mid \texttt{1} \mid \cdots \mid \texttt{9} \\ id & \rightarrow & letter_ \; (letter_ \mid digit)^* \end{array}$$

□

Пример 3.6. Беззнаковые числа (целые или с плавающей точкой) представляют собой строки типа `5280`, `0.01234`, `6.336E4` или `1.89E-4`. Точная спецификация этого множества строк представляет собой регулярное определение

$$\begin{array}{rcl} digit & \rightarrow & \texttt{0} \mid \texttt{1} \mid \cdots \mid \texttt{9} \\ digits & \rightarrow & digit\; digit^* \\ optionalFraction & \rightarrow & .\; digits \mid \epsilon \\ optionalExponent & \rightarrow & (\; \texttt{E}\; (\; + \mid - \mid \epsilon\;)\; digits) \mid \epsilon \\ number & \rightarrow & digits\; optionalFraction\; optionalExponent \end{array}$$

Данное определение гласит, что *optionalFraction* либо представляет собой десятичную точку, за которой следует одна или несколько цифр, либо отсутствует (является пустой строкой). В случае присутствия *optionalExponent* представляет собой `E`, за которым следует необязательный знак + или - и одна или несколько цифр. Заметьте, что за точкой должна следовать как минимум одна цифра, т.е. запись `1.` некорректна, в отличие от записи `1.0`. □

3.3.5 Расширения регулярных выражений

С тех пор как в 1950-х годах Клини (Kleene) ввел регулярные выражения с базовыми операторами для объединения, конкатенации и замыкания Клини, к ним было добавлено много расширений, повышающих их способность определять строковые шаблоны. Здесь будут упомянуты несколько расширений в записи регулярных выражений, которые первоначально появились в утилитах Unix, таких как `Lex`, и которые в особенности полезны при определении лексических анализаторов. В списке литературы к данной главе содержатся описания некоторых вариантов регулярных выражений, использующихся в настоящее время.

1. *Один или несколько экземпляров.* Унарный постфиксный оператор $^+$ представляет положительное замыкание регулярного выражения и его языка. Иными словами, если r — регулярное выражение, то $(r)^+$ описывает язык $(L(r))^+$. Оператор $^+$ имеет те же приоритет и ассоциативность, что и оператор *. Два полезных алгебраических закона, $r^* = r^+ \mid \epsilon$ и $r^+ = rr^* = r^*r$, связывают замыкание Клини и положительное замыкание.

2. *Нуль или один экземпляр.* Унарный постфиксный оператор ? означает "нуль или один экземпляр", т.е. запись $r?$ представляет собой сокращенную запись $r \mid \epsilon$ или, говоря иначе, $L(r?) = L(r) \cup \{\epsilon\}$. Оператор ? имеет те же приоритет и ассоциативность, что и операторы * и $^+$.

3. *Классы символов.* Регулярное выражение $a_1 \mid a_2 \mid \cdots \mid a_n$, где все a_i являются символами алфавита, может быть представлено сокращением $[a_1 a_2 \cdots a_n]$. Важно заметить, что, если $a_1, a_2, \ldots, a_n$ образуют логическую последовательность, т.е. последовательные прописные буквы, строчные буквы или цифры, их можно заменить выражением $a_1 - a_n$, т.е. первым и последним символами, разделенными дефисом. Например, [**abc**] представляет собой сокращение для **a** | **b** | **c**, а [**a** − **z**] — сокращение для **a** | **b** | $\cdots$ | **z**.

Пример 3.7. Используя указанные сокращения, можно переписать регулярное определение из примера 3.5 в виде

$$\begin{array}{rcl} \mathit{letter_} & \rightarrow & [\texttt{A-Za-z_}] \\ \mathit{digit} & \rightarrow & [\texttt{0-9}] \\ \mathit{id} & \rightarrow & \mathit{letter_}\ (\mathit{letter_} \mid \mathit{digit})^* \end{array}$$

Регулярное определение из примера 3.6 также можно упростить:

$$\begin{array}{rcl} \mathit{digit} & \rightarrow & [\texttt{0-9}] \\ \mathit{digits} & \rightarrow & \mathit{digit}^+ \\ \mathit{number} & \rightarrow & \mathit{digits}\ (.\ \mathit{digits})?\ (\texttt{E}\ [\texttt{+-}]?\ \mathit{digits})? \end{array}$$

□

3.3.6 Упражнения к разделу 3.3

Упражнение 3.3.1. Обратитесь к руководствам по языкам программирования а) C, б) C++, в) C#, г) Fortran, д) Java, е) Lisp, ж) SQL и определите 1) множество символов, образующих входной алфавит (исключая символы, которые могут встречаться только в строках символов или комментариях), 2) лексический вид числовых констант и 3) лексический вид идентификаторов.

! **Упражнение 3.3.2.** Опишите языки, соответствующие следующим регулярным выражениям:

а) $\mathbf{a}(\mathbf{a}|\mathbf{b})^*\mathbf{a}$;

б) $((\epsilon|\mathbf{a})\mathbf{b}^*)^*$;

в) $(\mathbf{a}|\mathbf{b})^*\mathbf{a}(\mathbf{a}|\mathbf{b})(\mathbf{a}|\mathbf{b})$;

г) $\mathbf{a}^*\mathbf{ba}^*\mathbf{ba}^*\mathbf{ba}^*$;

!! д) $(\mathbf{aa}|\mathbf{bb})^*((\mathbf{ab}|\mathbf{ba})(\mathbf{aa}|\mathbf{bb})^*(\mathbf{ab}|\mathbf{ba})(\mathbf{aa}|\mathbf{bb})^*)^*$.

Упражнение 3.3.3. Сколько в строке длиной n может содержаться

а) префиксов;

б) суффиксов;

в) истинных префиксов;

! г) подстрок;

! д) подпоследовательностей.

Упражнение 3.3.4. Большинство языков *чувствительны к регистру* (case sensitive), так что их ключевые слова могут записываться единственным образом, и регулярные выражения, описывающие соответствующие лексемы, очень просты. Однако некоторые языки наподобие SQL *нечувствительны к регистру* (case insensitive), так что ключевые слова могут быть записаны как строчными, так и прописными буквами, а также комбинацией букв в разных регистрах. Так, ключевое слово SQL `SELECT` может быть записано, например, как `select`, `Select` или `sElEcT`. Покажите, как написать регулярное выражение для ключевого слова в нечувствительном к регистру языке. Проиллюстрируйте ваше решение, написав регулярное выражение для "`select`" из SQL.

Упражнение 3.3.5. Напишите регулярные определения для следующих языков.

а) Все строки из строчных букв, содержащие пять гласных, a, e, i, o, u, в указанном порядке.

б) Все строки из строчных букв, в которых буквы находятся в возрастающем лексикографическом порядке.

в) Комментарии, представляющие собой строки, заключенные в `/*` и `*/`, без промежуточных символов `*/` (кроме случаев, когда они заключены в двойные кавычки).

!! г) Все строки из неповторяющихся цифр. *Указание*: попробуйте сначала решить задачу для нескольких цифр, например для $\{0, 1, 2\}$.

!! д) Все строки из цифр, причем в строке может повторяться не более одной цифры.

!! е) Все строки из a и b, в которых четное количество a и нечетное – b.

ж) Множество шахматных ходов в неформальной записи, таких как `p-k4` и `kbp×qn`.

!! з) Все строки из a и b, не содержащие подстроку abb.

и) Все строки из a и b, не содержащие подпоследовательность abb.

Упражнение 3.3.6. Запишите классы символов для следующих множеств.

а) Первые десять букв (по "j" включительно) как в верхнем, так и в нижнем регистрах.

б) Строчные согласные.

в) "Цифры" шестнадцатеричного числа (для "цифр", бóльших 9, могут использоваться либо строчные, либо прописные буквы).

г) Символы, могущие находиться в конце корректного предложения на английском языке (например, восклицательный знак).

В следующих упражнениях, по 3.3.10 включительно, используется расширенная запись регулярных выражений из Lex (генератор лексических анализаторов, который будет рассматриваться в разделе 3.5). Выражения расширенной записи приведены на рис. 3.8.

Выражение	Соответствие	Пример
c	Один неоператорный символ c	`a`
$\backslash c$	Символ c буквально	`*`
"s"	Строка s буквально	`"**"`
.	Любой символ, кроме символа новой строки	`a.*b`
^	Начало строки	`^abc`
$	Конец строки	`abc$`
$[s]$	Любой символ из s	`[abc]`
$[\hat{}s]$	Любой символ, не входящий в s	`[^abc]`
$r*$	Нуль или более строк, соответствующих r	`a*`
$r+$	Одна или более строк, соответствующих r	`a+`
$r?$	Нуль или одно r	`a?`
$r\{m,n\}$	От m до n повторений r	`a{1,5}`
r_1r_2	r_1, за которым следует r_2	`ab`
$r_1 \mid r_2$	r_1 или r_2	`a|b`
(r)	То же, что и r	`(a|b)`
r_1/r_2	r_1, если за ним следует r_2	`abc/123`

Рис. 3.8. Регулярные выражения Lex

Упражнение 3.3.7. Обратите внимание на то, что в регулярных выражениях следующие символы (*операторные*) имеют особый смысл:

```
\ " . ^ $ [ ] * + ? { } | /
```

Этот особый смысл можно отключить, если требуется, чтобы такой символ представлял в строке самого себя. Этого можно добиться, заключив строку длиной не менее 1 в двойные кавычки; например, регулярному выражению `"**"` соответствует строка `**`. Можно также назначить буквальное значение операторному символу, поместив перед ним обратную косую черту. Так, регулярному выражению `**` также соответствует строка `**`. Напишите регулярное выражение, которому соответствует строка `"\`.

Упражнение 3.3.8. В Lex *дополняющий класс символов* представляет любой символ, кроме перечисленных в классе символов. Обозначением дополняющего класса является знак дополнения ^ в качестве первого символа. Сам по себе этот символ не является частью дополняющего класса, если только он не указан в этом классе в другом месте. Таким образом, `[^A-Za-z]` соответствует любому символу, не являющемуся прописной или строчной буквой английского алфавита, а `[^\^]` представляет любой символ, кроме знака дополнения (или символа новой строки, поскольку он не может входить ни в какой класс символов). Покажите, что для каждого регулярного выражения с дополняющими классами существует эквивалентное регулярное выражение без дополнений.

! **Упражнение 3.3.9.** Регулярное выражение $r\{m,n\}$ представляет от m до n повторений шаблона r. Например, `a{1,5}` соответствует строкам, состоящим из символов `a` от одного до пяти. Покажите, что для каждого регулярного выражения с таким оператором повторения существует эквивалентное регулярное выражение без этого оператора.

! **Упражнение 3.3.10.** Оператор ^ соответствует левому концу строки, а `$` — правому концу. Это тот же оператор, который используется в классе-дополнении, и его конкретный смысл определяется контекстом его применения. Например, `^[^aeiou]*$` соответствует любой полной строке, не содержащей ни одной строчной гласной.

а) Как определить, какое именно значение имеет то или иное вхождение символа ^ в регулярное выражение?

б) Всегда ли можно заменить регулярное выражение с операторами ^ и $ эквивалентным регулярным выражением без этих операторов?

! **Упражнение 3.3.11.** В операционной системе UNIX оболочка `sh` использует операторы, приведенные на рис. 3.9, при указании имен файлов для описания множества файлов. Например, выражение `*.o` соответствует всем файлам, имена которых заканчиваются на `.o`; `sort1.?` соответствует всем именам вида `sort1.`*c*,

где c — произвольный символ. Покажите, как выражения для имен файлов в `sh` могут быть заменены регулярными выражениями с применением только базовых операторов объединения, конкатенации и замыкания.

Выражение	Соответствие	Пример
$'s'$	Строка s буквально	`'\'`
$\backslash c$	Символ c буквально	`\'`
$*$	Любая строка	`*.o`
$?$	Любой символ	`sort.?`
$[s]$	Любой символ из s	`sort.[cso]`

Рис. 3.9. Выражения для имен файлов, используемые sh

Упражнение 3.3.12. В SQL присутствует рудиментарное использование шаблонов с двумя символами со специальным смыслом: символ подчеркивания (_) означает любой один символ, а символ процента (`%`) — любую строку из нуля или большего количества символов. Кроме того, программист может определить любой символ, скажем, e, как служебный, так что его предшествование символам _, `%` или самому e возвращает им буквальное значение. Покажите, как записать любой шаблон SQL в виде регулярного выражения, если известно, какой именно символ выступает в роли служебного.

3.4 Распознавание токенов

Из предыдущего раздела вы узнали, как записывать шаблоны с использованием регулярных выражений. Теперь нужно изучить, как из шаблонов для всех необходимых токенов построить часть кода, которая исследует входной поток и находит в нем префикс, который является лексемой для одного из шаблонов. Мы воспользуемся для этого следующим примером.

Пример 3.8. Фрагмент грамматики на рис. 3.10 описывает простые инструкцию ветвления и условные выражения. Приведенный синтаксис похож на синтаксис языка программирования Pascal тем, что после условия явно указывается ключевое слово **then**. Для **relop** используются операторы сравнения, подобные операторам в Pascal и SQL, где = означает "равно", а <> — "не равно", поскольку они представляют интересную структуру лексем.

Терминалы грамматики **if**, **then**, **else**, **relop**, **id** и **number** представляют собой при рассмотрении лексического анализатора имена токенов. Шаблоны для этих токенов описаны с применением регулярных выражений на рис. 3.11. Шаблоны для *id* и *number* аналогичны шаблонам из примера 3.7.

$$
\begin{array}{lcl}
stmt & \rightarrow & \textbf{if } expr \textbf{ then } stmt \\
 & | & \textbf{if } expr \textbf{ then } stmt \textbf{ else } stmt \\
 & | & \epsilon \\
expr & \rightarrow & term \textbf{ relop } term \\
 & | & term \\
term & \rightarrow & \textbf{id} \\
 & | & \textbf{number}
\end{array}
$$

Рис. 3.10. Грамматика для инструкций ветвления

$$
\begin{array}{rcl}
digit & \rightarrow & \mathtt{[0-9]} \\
digits & \rightarrow & digit^{+} \\
number & \rightarrow & digits\ (\mathtt{.}\ digits)?\ (\ \mathtt{E}\ [+-]?\ digits\ )? \\
letter & \rightarrow & \mathtt{[A-Za-z]} \\
id & \rightarrow & letter\ (\ letter \mid digit\ )^{*} \\
if & \rightarrow & \mathtt{if} \\
then & \rightarrow & \mathtt{then} \\
else & \rightarrow & \mathtt{else} \\
relop & \rightarrow & \mathtt{<} \mid \mathtt{>} \mid \mathtt{<=} \mid \mathtt{>=} \mid \mathtt{=} \mid \mathtt{<>}
\end{array}
$$

Рис. 3.11. Шаблоны токенов из примера 3.8

В этом языке лексический анализатор должен распознавать ключевые слова `if`, `then` и `else`, а также лексемы, которые соответствуют шаблонам *relop*, *id* и *number*. Для упрощения считаем, что ключевые слова являются *зарезервированными*, т.е. несмотря на то, что они соответствуют шаблону идентификаторов, они не являются идентификаторами.

Кроме того, лексическому анализатору будет поручено удалить пробельные символы путем распознавания "токена" *ws*, определенного как

$$ws \rightarrow (\ \textbf{blank} \mid \textbf{tab} \mid \textbf{newline}\ )^{+}$$

Здесь **blank**, **tab** и **newline** являются абстрактными символами, которые использованы для того, чтобы выразить соответствующие символы ASCII – пробелы, символы табуляции и новой строки. Токен *ws* отличается от других токенов тем, что после распознавания он не возвращается синтаксическому анализатору, и лексический анализ выполняется заново, с символа, следующего за пробельным, в поисках токена, который будет возвращен синтаксическому анализатору.

Действия нашего лексического анализатора подытожены на рис. 3.12. В приведенной здесь таблице для каждой лексемы или семейства лексем указано, ка-

кое имя токена должно быть возвращено синтаксическому анализатору и каким должно быть значение атрибута (см. раздел 3.1.3). Обратите внимание на то, что для шести операторов сравнения в качестве атрибутов используются символьные константы `LT`, `LE` и другие, которые указывают, какие именно экземпляры токена **relop** обнаружены. Конкретный обнаруженный оператор определяет, какой именно код будет сгенерирован компилятором. □

Лексема	Имя токена	Значение атрибута
Любой *ws*	—	—
`if`	**if**	—
`then`	**then**	—
`else`	**else**	—
Любой *id*	**id**	Указатель на запись в таблице
Любой *number*	**number**	Указатель на запись в таблице
<	**relop**	`LT`
<=	**relop**	`LE`
=	**relop**	`EQ`
<>	**relop**	`NE`
>	**relop**	`GT`
>=	**relop**	`GE`

Рис. 3.12. Токены, их шаблоны и значения атрибутов

3.4.1 Диаграммы переходов

В качестве промежуточного шага при построении лексического анализатора вначале преобразуем шаблоны в стилизованные блок-схемы, именуемые диаграммами переходов (transition diagram). В данном разделе мы выполняем преобразование шаблонов в виде регулярных выражений в диаграммы переходов вручную, но в разделе 3.6 будет показано наличие механического способа построения таких диаграмм из коллекций регулярных выражений.

Диаграмма переходов содержит ряд узлов, или кружков, именуемых *состояниями* (state). Каждое состояние представляет ситуацию, которая может возникнуть в процессе сканирования входного потока в поисках лексемы, соответствующей одному из нескольких шаблонов. Состояния можно рассматривать как подытоженную информацию обо всех просмотренных символах между указателями *lexemeBegin* и *forward* (как в ситуации на рис. 3.3).

Дуги (edge) представляют собой направленные линии из одного состояния в другое. Каждая дуга *помечена* символом или множеством символов. Если мы находимся в некотором состоянии s и следующий входной символ — a, мы ищем дугу, исходящую из s и помеченную a (а возможно, и какими-то другими символами). Если мы находим такую дугу, то перемещаем указатель *forward* и входим в новое состояние диаграммы переходов, в которое нас приводит упомянутая дуга. Мы полагаем, что все наши диаграммы переходов *детерминированные* (deterministic), т.е. что имеется не более одной дуги, выходящей из данного состояния с данным символом среди ее меток. В разделе 3.5 требование детерминированности будет ослаблено, что облегчит жизнь разработчикам лексических анализаторов и усложнит — реализующим их программистам. Вот некоторые важные соглашения о диаграммах переходов.

1. Некоторые состояния являются *допускающими* (*принимающими*, accepting), или *конечными* (final). Эти состояния указывают, что искомая лексема найдена, хотя фактически лексема может и не состоять из всех позиций между указателями *lexemeBegin* и *forward*. Конечные состояния на диаграмме изображены как двойные кружкú, и если должно выполняться какое-то действие — обычно возврат токена и значения его атрибута синтаксическому анализатору, — то оно указывается у конечного состояния.

2. Кроме того, если необходимо вернуть указатель *forward* на одну позицию назад (т.е. лексема не включает символ, который привел нас в конечное состояние), то возле такого узла дополнительно ставится символ *. В нашем примере никогда не потребуется возвращать указатель *forward* более чем на одну позицию, но если это необходимо, то у конечного состояния можно указать соответствующее количество звездочек.

3. Одно состояние является *стартовым* (start), или *начальным* (initial); оно указывается при помощи дуги, помеченной словом “start” и входящей в состояние ниоткуда. Диаграмма переходов всегда начинается в стартовом состоянии (перед тем как будет считан хотя бы один символ из входного потока).

Пример 3.9. На рис. 3.13 приведена диаграмма переходов, которая распознает лексемы, соответствующие токену **relop**. Работа начинается со стартового состояния 0. Если первым входным символом оказывается <, то среди лексем, соответствующих шаблону для **relop**, могут быть только <, <> и <=. Мы переходим в состояние 1 и смотрим на следующий символ. Если это =, то оказывается распознанной лексема <=; мы переходим в состояние 2 и возвращаем токен **relop** с атрибутом LE, символьной константой, представляющей конкретный оператор сравнения. Если в состоянии 1 следующим символом является >, то оказывается

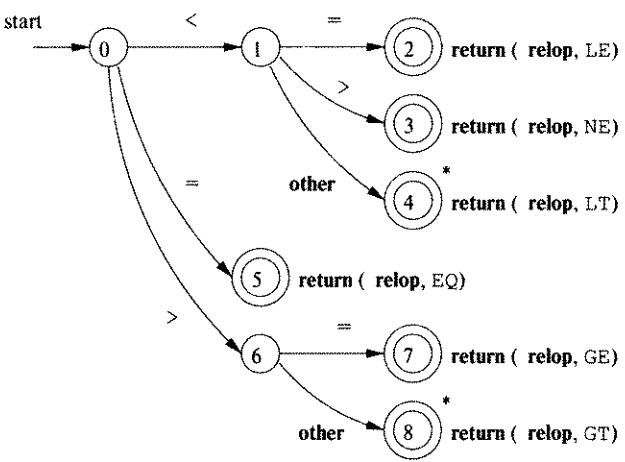


Рис. 3.13. Диаграмма переходов для relop

распознанной лексема <>; мы переходим в состояние 3 и возвращаем синтаксическому анализатору информацию о том, что найден оператор "не равно". В случае любых других символов найденная лексема — <, и мы переходим в состояние 4 для возврата этой информации. Обратите внимание, что состояние 4 помечено звездочкой, чтобы указать, что мы должны вернуться на одну позицию назад во входном потоке.

С другой стороны, если в состоянии 0 первый встреченный нами символ — =, то этот единственный символ и является лексемой. Мы немедленно возвращаем синтаксическому анализатору этот факт из состояния 5. Последняя возможность заключается в том, что первый символ представляет собой >. В таком случае мы должны войти в состояние 6 и на основании очередного символа решить, является ли лексема лексемой >= (если очередной считанный символ — =) или лексемой > (во всех остальных случаях). Заметим также, что если в состоянии 0 мы считываем символ, отличный от <, = или >, то прочесть лексему, соответствующую токену **relop**, мы не в состоянии и приведенная диаграмма не может быть использована.□

3.4.2 Распознавание зарезервированных слов и идентификаторов

Распознавание ключевых слов и идентификаторов представляет собой определенную проблему. Обычно ключевые слова наподобие `if` и `then` зарезервированы (как в нашем примере), так что они не являются идентификаторами, даже если они *выглядят*, как идентификаторы. Таким образом, хотя для поиска лексем идентификаторов обычно используется диаграмма переходов, приведенная на

рис. 3.14, она распознает также ключевые слова `if`, `then` и `else` из нашего примера.

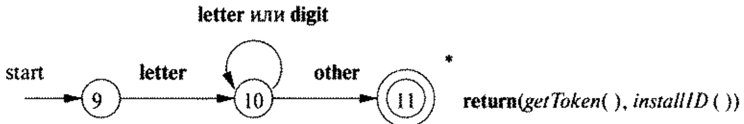


Рис. 3.14. Диаграмма переходов для идентификаторов и ключевых слов

Есть два способа обработки зарезервированных слов, выглядящих, как идентификаторы.

1. *Изначально внести зарезервированные слова в таблицу символов.* Поле записи в таблице символов указывает, что эти строки не могут быть обычными идентификаторами, и определяет, какие именно токены они представляют. Предполагается, что именно такой метод используется на рис. 3.14. Когда мы находим идентификатор, вызов *installID* помещает его в таблицу символов, если он еще не находится там, и возвращает указатель на запись в таблице символов для найденной лексемы. Само собой, никакой идентификатор, который в момент лексического анализа не находится в таблице символов, не может быть зарезервированным словом, так что его токен — **id**. Функция *getToken* ищет запись для найденной лексемы в таблице символов и возвращает имя токена в соответствии с информацией из таблицы символов — **id** или один из токенов ключевых слов, изначально установленных в таблице.

2. *Создать отдельные диаграммы переходов для каждого ключевого слова.* Пример такой диаграммы приведен на рис. 3.15. Заметим, что такая диаграмма переходов построена из состояний, представляющих ситуацию после каждой очередной прочитанной буквы ключевого слова; за последней буквой выполняется проверка, что считанный символ не является ни буквой, ни цифрой, т.е. символом, который может быть продолжением идентификатора. Такая проверка завершения идентификатора необходима, иначе можно вернуть токен **then** в ситуациях, когда корректным токеном является токен **id**, например в случае лексемы `thenextvalue`, истинным префиксом которой является `then`. Если принять данный подход, то необходимо назначить токенам приоритеты, чтобы в случае соответствия лексемы двум шаблонам токены зарезервированных слов при распознавании имели преимущество перед токеном **id**. В нашем примере данный подход *не используется*, поэтому состояния на рис. 3.15 не пронумерованы.

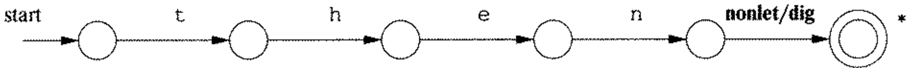


Рис. 3.15. Гипотетическая диаграмма переходов для ключевого слова then

3.4.3 Завершение примера

Диаграмма переходов для **id**, показанная на рис. 3.14, имеет простую структуру. Начиная с состояния 9, она проверяет, что лексема начинается с буквы, и в этом случае выполняется переход в состояние 10. В состоянии 10 мы находимся до тех пор, пока нам встречаются буквы и цифры. Первый же символ, не являющийся ни буквой, ни цифрой, приводит к переходу в состояние 11, в котором лексема является распознанной. Поскольку последний символ частью лексемы не является, требуется возврат на одну позицию назад; кроме того, как говорилось в разделе 3.4.2, найденная лексема должна быть внесена в таблицу символов, и при этом мы выясняем, что же именно найдено — ключевое слово или истинный идентификатор.

Диаграмма переходов для токена **number** показана на рис. 3.16, и пока что это самая сложная из виденных нами диаграмм. Начиная с состояния 12, мы переходим в состояние 13, если первый встреченный символ — цифра. В этом состоянии мы можем считать любое количество дополнительных символов. Если нам попадется символ, отличный от цифры, точки или E, значит, мы имеем дело с целым числом наподобие 123. Этот случай обрабатывается состоянием 20, в котором возвращаются токен **number** и указатель на таблицу констант, в которую внесена найденная лексема. Эти действия не показаны на диаграмме, но они аналогичны действиям, выполняемым при работе с идентификаторами.

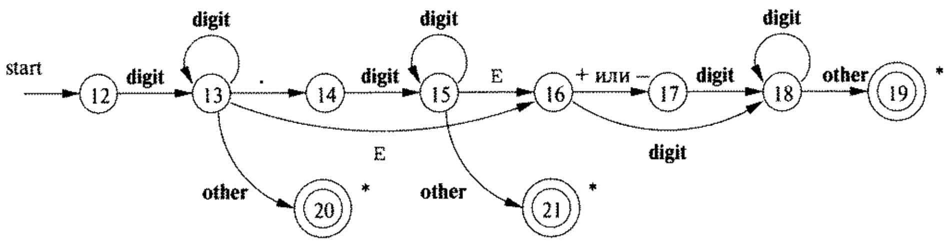


Рис. 3.16. Диаграмма переходов для беззнаковых чисел

Если в состоянии 13 нам встречается точка, значит, у нас есть "необязательная дробная часть". Мы переходим в состояние 14 и ожидаем одну или несколько дополнительных цифр (для этой цели используется состояние 15). Если нам встречается символ E (как и в состоянии 13), значит, наше число содержит "необязательный показатель степени", распознавание которого — обязанность, возлагаемая

на состояния 16–19. Если же в состоянии 15 нам встречается символ, не являющийся ни E, ни цифрой, значит, достигнут конец дробной части числа, у которого нет показателя степени, и мы возвращаем найденную лексему в состоянии 21.

Последняя диаграмма, показанная на рис. 3.17, предназначена для пробельных символов. На этой диаграмме мы ищем один или несколько "пробельных" символов, показанных как **delim**, — обычно это пробелы, символы табуляции и новой строки, а возможно, и иные символы, которые не рассматриваются языком как часть какого бы то ни было токена.

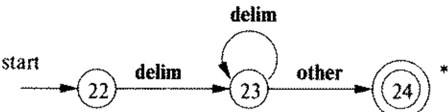


Рис. 3.17. Диаграмма переходов для пробельных символов

Обратите внимание, что в состояние 24 мы попадаем, обнаружив блок последовательных пробельных символов, за которыми следует символ, не являющийся пробельным. Поэтому необходимо вернуться на один символ назад, но, как уже говорилось ранее, возврата из лексического анализатора в синтаксический анализатор при этом не происходит. Вместо этого процесс лексического анализа после пробельных символов начинается заново.

3.4.4 Архитектура лексического анализатора на основе диаграммы переходов

Существует несколько способов применения набора диаграмм переходов для построения лексического анализатора. Независимо от общей стратегии каждое состояние представляет собой фрагмент кода. Можно представить переменную `state`, хранящую номер текущего состояния в диаграмме переходов. Конструкция `switch`, построенная на основе значения переменной `state`, дает нам код для каждого из возможных состояний, где и находятся действия, выполняемые в том или ином состоянии. Зачастую код состояния сам представляет собой конструкцию `switch` или множественное ветвление, которые путем чтения и исследования очередного символа входного потока определяют следующее состояние, в которое должен быть выполнен переход.

Пример 3.10. На рис. 3.18 приведен набросок функции `getRelop()` на языке программирования C++, работа которой состоит в моделировании диаграммы переходов на рис. 3.13 и возврате объекта типа `TOKEN`, т.е. пары, состоящей из имени токена (в данном случае — **relop**) и значения атрибута (в данном случае — кода для одного из шести возможных операторов сравнения). Функция `getRelop()` сна-

чала создает новый объект `retToken` и инициализирует его первый компонент символьным кодом `RELOP` для токена **relop**.

```
TOKEN getRelop()
{
   TOKEN retToken = new(RELOP);
   while(1) { /* Обработка символов до тех пор, пока
                 не будет выполнен возврат из функции
                 или не будет обнаружена ошибка */
      switch(state) {
         case 0: c = nextChar();
                 if ( c == '<' ) state = 1;
                 else if ( c == '=' ) state = 5;
                 else if ( c == '>' ) state = 6;
                 else fail(); /* Это не лексема relop */
                 break;
         case 1: ...
         ...
         case 8: retract();
                 retToken.attribute = GT;
                 return(retToken);
      }
   }
}
```

Рис. 3.18. Набросок реализации диаграммы переходов для relop

Типичное поведение состояния можно увидеть на примере состояния 0. Функция `nextChar()` получает очередной символ из входного потока и присваивает его локальной переменной `c`. Затем выполняется проверка соответствия `c` трем ожидаемым символам, и в каждом случае выполняется переход в состояние, определяемое диаграммой переходов на рис. 3.13. Например, если считанный символ — =, выполняется переход в состояние 5.

Если считанный символ не принадлежит к множеству символов, с которых может начинаться оператор сравнения, вызывается функция `fail()`. Какие именно действия выполняет эта функция, зависит от глобальной стратегии лексического анализатора по восстановлению после ошибок. Она должна сбросить указатель `forward` в значение `lexemeBegin` для того, чтобы дать возможность применить другую диаграмму переходов к началу необработанной части входного потока. Затем она может изменить значение переменной `state` на стартовое состояние для другой диаграммы переходов, которая будет заниматься поиском другого токе-

на. Или, если оставшихся неиспользованными диаграмм переходов больше нет, функция `fail()` может инициализировать фазу исправления ошибок, которая попытается исправить входной поток и найти лексему так, как это описано в разделе 3.1.4.

На рис. 3.18 показаны также действия для состояния 8. Поскольку состояние 8 помечено звездочкой, требуется вернуть указатель входного потока на одну позицию назад (т.е. вернуть `c` назад во входной поток). Эта задача решается вызовом функции `retract()`. Поскольку состояние 8 представляет распознанную лексему >, второй компонент (с именем `attribute`) возвращаемого объекта устанавливается равным GT, коду для данного оператора. □

Давайте рассмотрим, каким образом код наподобие приведенного на рис. 3.18 может быть встроен в лексический анализатор.

1. Можно последовательно испытывать диаграммы переходов для каждого токена. В таком случае функция `fail()` из примера 3.10 при вызове сбрасывает значение указателя `forward` и приступает к новой диаграмме переходов. Этот метод позволяет нам использовать диаграммы переходов для отдельных ключевых слов наподобие диаграммы, предложенной на рис. 3.15. Однако, чтобы ключевые слова были зарезервированными, мы должны использовать их диаграммы переходов до диаграммы переходов для **id**.

2. Можно работать с разными диаграммами переходов "параллельно", передавая очередной считанный символ им всем и выполняя соответствующий переход в каждой из диаграмм переходов. При использовании этой стратегии нужно быть очень осторожным в ситуации, когда одна диаграмма переходов находит лексему, соответствующую ее шаблону, но при этом другая диаграмма (или даже несколько) все еще способна обрабатывать входной поток. Обычно используемая стратегия состоит в выборе самого длинного префикса входного потока, соответствующего какому-либо из шаблонов. Это правило вынуждает, например, предпочесть идентификатор `thenext` ключевому слову `then`, а оператор `->` — оператору `-`.

3. Предпочтительный подход — и именно он будет использован в последующих разделах — состоит в объединении всех диаграмм переходов в одну. Эта диаграмма переходов считывает символы до тех пор, пока возможные следующие состояния не оказываются исчерпаны. После этого выбирается наибольшая лексема, соответствующая некоторому шаблону (правило, рассматривавшееся в предыдущем пункте). В нашем примере такое объединение выполняется достаточно просто, поскольку никакие два токена не начинаются с одного и того же символа, так что первый же символ сразу говорит нам, какой токен мы ищем. Таким образом, можно просто

объединить состояния 0, 9, 12 и 22 в одно стартовое состояние, оставив все остальные состояния без изменений. Однако, как мы вскоре увидим, в общем случае задача объединения диаграмм переходов для нескольких токенов существенно более сложная.

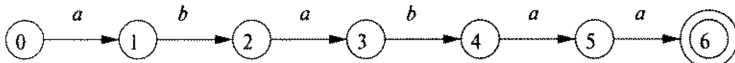
3.4.5 Упражнения к разделу 3.4

Упражнение 3.4.1. Разработайте диаграммы переходов для распознавания языков, определяемых регулярными выражениями в упражнении 3.3.2.

Упражнение 3.4.2. Разработайте диаграммы переходов для распознавания языков из условия упражнения 3.3.5.

Следующие упражнения, по 3.4.12 включительно, знакомят читателей с алгоритмом Ахо–Корасик (Aho–Corasick) для распознавания набора ключевых слов в текстовой строке за время, пропорциональное длине текста и сумме длин ключевых слов. В этом алгоритме используются диаграммы переходов специального вида, которые называются *лучами* (trie)[2]. Луч представляет собой диаграмму переходов с древовидной структурой с различными метками на дугах, ведущих от узла к дочерним по отношению к нему узлам. Листья луча представляют распознанные ключевые слова.

Кнут (Knuth), Моррис (Morris) и Пратт (Pratt) представили алгоритм (алгоритм Кнута–Морриса–Пратта, КМП) для распознавания одного ключевого слова $b_1 b_2 \dots b_n$ в текстовой строке. Здесь луч представляет собой диаграмму переходов с n состояниями, от 0 до n. Состояние 0 — начальное, а состояние n — допускающее, т.е. представляющее распознанное ключевое слово. Для каждого состояния s от 0 до $n-1$ имеется переход в состояние $s+1$, помеченный символом b_{s+1}. Например, луч для ключевого слова ababaa имеет вид



Для быстрой обработки текстовой строки и поиска в ней ключевого слова можно для ключевого слова $b_1 b_2 \dots b_n$ и позиции s в нем (соответствующей состоянию s в луче) определить *функцию отказа* (failure function) $f(s)$, вычисляемую так, как показано на рис. 3.19. Смысл этой функции в том, что $b_1 b_2 \dots b_{f(s)}$ представляет собой наибольший истинный префикс строки $b_1 b_2 \dots b_s$, который одновременно является суффиксом $b_1 b_2 \dots b_s$. Причина, по которой функция $f(s)$ так важна, заключается в том, что если мы попытаемся сопоставить текстовую строку слову $b_1 b_2 \dots b_n$ и в первых s позициях будет обнаружено совпадение,

[2]Здесь использован перевод этого термина из книги Кнут Д. *Искусство программирования. Т.3.* — М.: Издательский дом "Вильямс", 2000 (раздел 6.3, с. 527). — *Прим. пер.*

а в следующей позиции — несовпадение (т.е. очередной символ текстовой строки не будет совпадать с b_{s+1}), то $f(s)$ — самый длинный префикс слова $b_1 b_2 \ldots b_n$, который может соответствовать текстовой строке до текущей точки. Само собой разумеется, следующий символ текста при этом должен быть $b_{f(s)+1}$, иначе проблема несовпадения останется и придется рассматривать более короткий префикс, $b_{f(f(s))}$.

```
1)  t = 0;
2)  f(1) = 0;
3)  for (s = 1; s < n; s + +) {
4)      while (t > 0 && b_{s+1} != b_{t+1}) t = f(t);
5)      if (b_{s+1} == b_{t+1}) {
6)          t = t + 1;
7)          f(s + 1) = t;
        }
8)      else f(s + 1) = 0;
    }
```

Рис. 3.19. Алгоритм для вычисления функции отказа для ключевого слова $b_1 b_2 \ldots b_n$

В качестве примера приведем функцию отказа для луча, построенного ранее для слова `ababaa`:

s	1	2	3	4	5	6
$f(s)$	0	0	1	2	3	1

Например, состояния 3 и 1 представляют префиксы `aba` и `a` соответственно. $f(3) = 1$, поскольку `a` является самым длинным истинным префиксом `aba`, который одновременно является суффиксом `aba`. $f(2) = 0$, поскольку самый длинный префикс `ab`, являющийся одновременно ее суффиксом, — пустая строка.

Упражнение 3.4.3. Постройте функцию отказа для следующих строк:

а) `ababaab`;

б) `aaaaaa`;

в) `abbaabb`.

! **Упражнение 3.4.4.** Докажите по индукции по s, что алгоритм на рис. 3.19 корректно вычисляет функцию отказа.

!! **Упражнение 3.4.5.** Покажите, что присваивание $t = f(t)$ в строке 4 алгоритма на рис. 3.19 выполняется максимум n раз. Покажите, что, следовательно, весь алгоритм требует $O(n)$ времени для обработки ключевого слова длиной n.

Вычислив функцию отказа для ключевого слова $b_1 b_2 \dots b_n$, мы можем сканировать строку $a_1 a_2 \dots a_m$ в поисках данного ключевого слова за время $O(m)$. Алгоритм, приведенный на рис. 3.20, перемещает ключевое слово вдоль строки, сравнивая символы ключевого слова с символами текстовой строки. Если происходит несовпадение после соответствия s символов, то алгоритм смещает ключевое слово вправо на $s - f(s)$ позиций, так что рассматриваются как совпадающие со строкой только первые $f(s)$ символов ключевого слова.

```
1)  s = 0;
2)  for (i = 1; i ≤ m; i++) {
3)          while (s > 0 && a_i != b_{s+1}) s = f(s);
4)          if (a_i == b_{s+1}) s = s + 1;
5)          if (s == n) return "yes";
    }
6)  return "no";
```

Рис. 3.20. Алгоритм КМП для проверки наличия в строке $a_1 a_2 \dots a_m$ ключевого слова $b_1 b_2 \dots b_n$ в качестве подстроки за время $O(m + n)$

Упражнение 3.4.6. Примените алгоритм КМП к проверке вхождения ключевого слова ababaa в качестве подстроки в

а) abababaab;

б) abababbaa.

!! **Упражнение 3.4.7.** Покажите, что алгоритм на рис. 3.20 корректно определяет, является ли ключевое слово подстрокой данной строки. *Указание*: примените индукцию по i. Покажите, что для всех i значение s после строки 4 представляет собой длину наибольшего префикса ключевого слова, являющегося суффиксом строки $a_1 a_2 \dots a_i$.

!! **Упражнение 3.4.8.** Покажите, что время работы алгоритма на рис. 3.20 составляет $O(m + n)$, в предположении, что функция f уже вычислена и ее значения хранятся в массиве, проиндексированном s.

Упражнение 3.4.9. *Строки Фибоначчи* определяются следующим образом.

1. $s_1 = \texttt{b}$.
2. $s_2 = \texttt{a}$.
3. $s_k = s_{k-1} s_{k-2}$ при $k > 2$.

Например, $s_3 = \mathtt{ab}$, $s_4 = \mathtt{aba}$, $s_5 = \mathtt{abaab}$.

а) Какова длина строки s_n?

б) Постройте функцию отказа для s_6.

в) Постройте функцию отказа для s_7.

!! г) Покажите, что функция отказа для любой строки s_n может быть выражена следующим образом: $f(1) = f(2) = 0$, а для $2 < j \leqslant |s_n|$ значения функции $f(j) = j - |s_{k-1}|$, где k — наибольшее целое число, такое, что $|s_k| \leqslant j + 1$.

!! д) Чему равно наибольшее число последовательных применений функции отказа в алгоритме КМП при попытке определить, входит ли ключевое слово s_k в текстовую строку s_{k+1}?

Ахо (Aho) и Корасик (Corasick) обобщили алгоритм КМП для распознавания любого множества ключевых слов в текстовой строке. В этом случае луч является истинным деревом с ветвлением в корне. Имеется по одному состоянию для каждой строки, которая является префиксом (не обязательно истинным) некоторого ключевого слова. Родительским по отношению к состоянию, соответствующему строке $b_1 b_2 \dots b_k$, является состояние, соответствующее $b_1 b_2 \dots b_{k-1}$. Состояние является конечным, если оно соответствует полному ключевому слову. Например, на рис. 3.21 показан луч для ключевых слов `he`, `she`, `his` и `hers`.

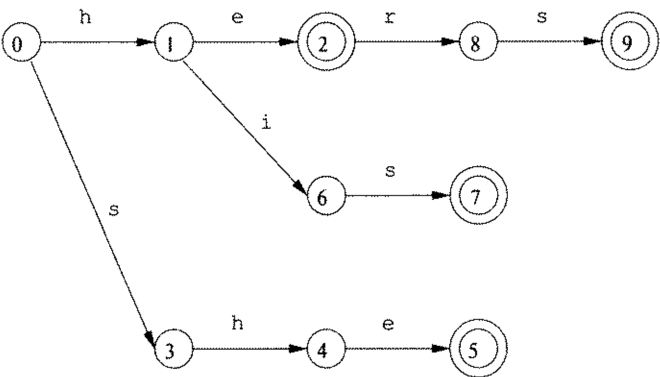


Рис. 3.21. Луч для ключевых слов he, she, his и hers

Функция отказа для луча в общем случае определяется следующим образом. Пусть s — состояние, соответствующее строке $b_1 b_2 \dots b_n$. Тогда $f(s)$ — это состояние, соответствующее самому длинному истинному суффиксу $b_1 b_2 \dots b_n$, который является также префиксом *некоторого* ключевого слова. Например, вот как выглядит функция отказа для луча на рис. 3.21:

s	1	2	3	4	5	6	7	8	9
$f(s)$	0	0	0	1	2	0	3	0	3

! **Упражнение 3.4.10.** Модифицируйте алгоритм, приведенный на рис. 3.19, для вычисления функции отказа для лучей в общем случае. *Указание*: основное отличие заключается в том, что мы не можем просто проверить равенство или неравенство b_{s+1} и b_{t+1} в строках 4 и 5 на рис. 3.19. Из любого состояния могут иметься несколько переходов для ряда символов, как, например, переходы для `e` и `i` из состояния 1 на рис. 3.21. Любой из этих переходов может привести в состояние, которое представляет самый длинный суффикс, являющийся также префиксом.

Упражнение 3.4.11. Постройте лучи и вычислите функцию отказа для следующих множеств ключевых слов:

а) `aaa`, `abaaa` и `ababaaa`;

б) `all`, `fall`, `fatal`, `llama` и `lame`;

в) `pipe`, `pet`, `item`, `temper` и `perpetual`.

! **Упражнение 3.4.12.** Покажите, что разработанный вами алгоритм из упражнения 3.4.10 будет выполняться за время, линейно пропорциональное сумме длин ключевых слов.

3.5 Генератор лексических анализаторов Lex

В этом разделе мы познакомимся с программным инструментом под названием Lex (или, в более поздних реализациях, Flex), который позволяет определить лексический анализатор, указывая регулярные выражения для описания шаблонов токенов. Входные обозначения для Lex обычно называют *языком Lex*, а сам инструмент — *компилятором Lex*. Компилятор Lex преобразует входные шаблоны в диаграмму переходов и генерирует код (в файле с именем `lex.yy.c`), имитирующий данную диаграмму переходов. Механизм преобразования регулярных выражений в диаграммы переходов является темой следующих разделов данной главы; здесь же мы остановимся на языке Lex.

3.5.1 Использование Lex

На рис. 3.22 показана схема использования Lex. Входной файл `lex.l` написан на языке Lex и описывает генерируемый лексический анализатор. Компилятор Lex преобразует `lex.l` в программу на языке программирования C в файле с именем `lex.yy.c`. Этот файл компилируется компилятором C в файл с названием

a.out, как обычно. Выход компилятора C представляет собой работающий лексический анализатор, который может получать поток входных символов и выдавать поток токенов.

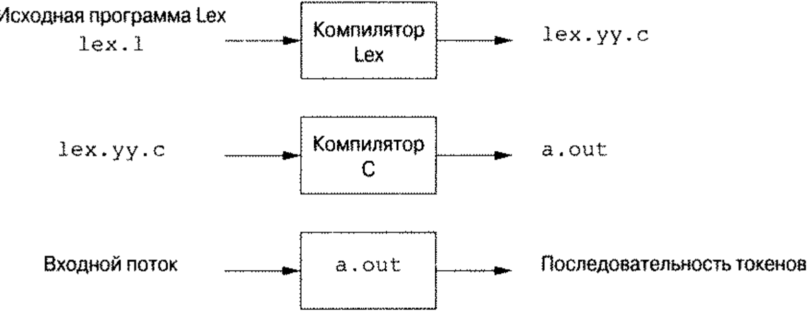


Рис. 3.22. Создание лексического анализатора с помощью Lex

Обычно скомпилированная программа на языке C, которая на рис. 3.22 показана как a.out, используется в качестве подпрограммы синтаксического анализатора. Это функция на языке программирования C, которая возвращает целое число, представляющее собой код одного из возможных имен токенов. Значение атрибута, которое может быть другим числовым кодом, указателем на запись таблицы символов или просто отсутствовать, помещается в глобальную переменную yylval[3], которая совместно используется лексическим и синтаксическим анализаторами. Этот метод позволяет вернуть из функции как имя токена, так и значение его атрибута.

3.5.2 Структура программ Lex

Программа Lex имеет следующий вид:

```
Объявления
%%
Правила трансляции
%%
Вспомогательные функции
```

Раздел объявлений включает объявления переменных, *именованные константы* (manifest constant – идентификаторы констант, например имена токенов) и регулярные определения в стиле раздела 3.3.4.

[3]Кстати, сочетание yy, встречающееся в yylval и lex.yy.c, связано с генератором синтаксических анализаторов Yacc, который будет рассматриваться в разделе 4.9 и который обычно используется в связке с Lex.

Правила трансляции имеют вид

Шаблон { Действие }

Каждый шаблон является регулярным выражением, которое может использовать регулярные определения из раздела объявлений. Действия представляют собой фрагменты кода, обычно написанные на языке программирования C, хотя созданы многие разновидности Lex для других языков программирования.

Третий раздел содержит различные дополнительные функции, используемые в действиях. Эти функции могут также компилироваться отдельно и загружаться лексическим анализатором во время работы.

Лексический анализатор, созданный Lex, работает во взаимодействии с синтаксическим анализатором. При вызове синтаксическим анализатором лексический анализатор считывает по одному символу оставшийся непрочитанным входной поток, пока не будет найден самый длинный префикс входного потока, соответствующий одному из шаблонов P_i. Затем лексический анализатор выполняет связанные с ним действия A_i. Обычно A_i содержит возврат в синтаксический анализатор, но если это не так (например, если P_i описывает пробельные символы или комментарии), то лексический анализатор продолжает поиск дополнительных лексем, пока одно из соответствующих действий не приведет к возврату из функции лексического анализатора в синтаксический анализатор. Лексический анализатор возвращает синтаксическому анализатору единственное значение, имя токена, но использует при этом совместно используемую целочисленную переменную `yylval` для передачи при необходимости дополнительной информации о найденной лексеме.

Пример 3.11. На рис. 3.23 приведена программа Lex, которая распознает токены, представленные на рис. 3.12, и возвращает найденный токен синтаксическому анализатору. Рассмотрение приведенного кода поможет нам изучить многие важные возможности Lex.

В разделе объявлений мы встречаем пару специальных скобок — `%{` и `%}`. Все, что заключено в эти скобки, копируется непосредственно в `lex.yy.c` и не рассматривается как регулярное определение. Обычно здесь помещаются определения именованных констант с использованием конструкции `#define` языка программирования C для назначения каждой именованной константе уникального целочисленного кода. В нашем примере мы перечислили в комментарии именованные константы `LT`, `IF` и другие, но не привели определения, назначающие им конкретные числовые значения.[4]

[4]Если Lex используется вместе с Yacc, то обычно константы определяются в программе Yacc и используются в программе Lex без определений. Поскольку файл `lex.yy.c` компилируется совместно с выходом Yacc, эти константы окажутся доступными действиям в программе Lex.

```
%{
    /* Определения именованных констант
    LT, LE, EQ, NE, GT, GE,
    IF, THEN, ELSE, ID, NUMBER, RELOP */
%}

/* Регулярные определения */
delim      [ \t\n]
ws         {delim}+
letter     [A-Za-z]
digit      [0-9]
id         {letter}({letter}|{digit})*
number     {digit}+(\.{digit}+)?(E[+-]?{digit}+)?

%%

{ws}       {/* Нет действий и выхода из функции */}
if         {return(IF);}
then       {return(THEN);}
else       {return(ELSE);}
{id}       {yylval = (int) installID(); return(ID);}
{number}   {yylval = (int) installNum(); return(NUMBER);}
"<"        {yylval = LT; return(RELOP);}
"<="       {yylval = LE; return(RELOP);}
"="        {yylval = EQ; return(RELOP);}
"<>"       {yylval = NE; return(RELOP);}
">"        {yylval = GT; return(RELOP);}
">="       {yylval = GE; return(RELOP);}

%%

int installID() {/* Функция для внесения лексемы
                    (на первый символ которой указывает
                    указатель yytext и длина которой равна
                    yyleng) в таблицу символов. Возвращает
                    указатель на соответствующую запись
                    таблицы символов */
}

int installNum() {/* Аналогична функции installID, но
                     помещает числовые константы в
                     отдельную таблицу */
}
```

Рис. 3.23. Программа Lex для распознавания токенов на рис. 3.12

Кроме того, в разделе объявлений имеется последовательность регулярных определений. Они используют расширенную запись регулярных выражений, описанную в разделе 3.3.5. Регулярные определения, используемые в следующих за ними определениях или шаблонах правил трансляции, помещаются в фигурные скобки. Так, например, *delim* определено как сокращение для класса символов, состоящего из пробела, символа табуляции и символа новой строки (последние два символа представлены, как во всех командах UNIX, при помощи обратной косой черты, за которой следует соответственно буква `t` или `n`). Затем *ws* определено как один или несколько разделителей при помощи регулярного выражения `{delim}+`.

Обратите внимание, что в определениях *id* и *number* скобки используются в качестве метасимволов группирования и не обозначают сами себя. Напротив, буква `E` в определении *number* означает саму себя. Если мы хотим использовать один из метасимволов Lex, такой как любая скобка, `.`, `+`, `*` или `?`, так, чтобы он обозначал сам себя, то его следует предварить обратной косой чертой. Например, в определении *number* имеется последовательность `\.`, представляющая обычную точку, поскольку просто точка в регулярных выражениях является метасимволом, представляющим “любой символ”, как это принято в регулярных выражениях UNIX.

В разделе вспомогательных функций имеются функции `installID()` и `installNum()`. Так же, как и часть раздела объявлений, находящаяся между `%{...%}`, все, что находится во вспомогательном разделе, просто копируется в файл `lex.yy.c`, но может использоваться в действиях.

Наконец, рассмотрим некоторые из шаблонов и правил в среднем разделе на рис. 3.23. Первым указан идентификатор *ws*, объявленный в первом разделе и связанный с пустым действием. Если встречается пробельный символ, то выполняется не возврат в синтаксический анализатор, а поиск новой лексемы. Второй токен имеет простой шаблон регулярного выражения `if`. Если во входном потоке встречаются две буквы `if`, за которыми не следует другая буква или цифра (в этом случае мы имеем дело с более длинным префиксом входной строки, соответствующим шаблону для **id**), лексический анализатор выбирает эти символы из входного потока и возвращает имя токена `IF`, т.е. целое число, которому соответствует именованная константа `IF`. Ключевые слова `then` и `else` обрабатываются аналогично.

Пятый токен имеет шаблон, определенный при помощи *id*. Заметим, что, хотя ключевые слова наподобие `if` соответствуют как своему шаблону, так и данному, Lex в ситуации, когда наибольший префикс соответствует двум или большему количеству шаблонов, выбирает первый из них. Действие при обнаружении соответствия шаблону *id* включает следующее.

1. Вызывается функция installID(), которая помещает найденную лексему в таблицу символов.

2. Эта функция возвращает указатель на запись в таблице символов, который сохраняется в глобальной переменной yylval и может быть использован синтаксическим анализатором или другим компонентом компилятора. Функция installID() имеет доступ к двум переменным, которые автоматически устанавливаются генерируемым Lex лексическим анализатором:

   а) указатель yytext на начало лексемы — аналог указателя lexemeBegin на рис. 3.3;

   б) целочисленная переменная yyleng, содержащая длину найденной лексемы.

3. Синтаксическому анализатору возвращается имя токена ID.

При обнаружении лексемы, соответствующей шаблону *number*, выполняются аналогичные действия, но с использованием вспомогательной функции installNum(). □

3.5.3 Разрешение конфликтов в Lex

Следует упомянуть о двух правилах, которыми руководствуется Lex при выборе лексемы в случае, когда несколько префиксов входной строки соответствуют одному или нескольким шаблонам.

1. Предпочтение всегда отдается более длинному префиксу.

2. Если наибольший возможный префикс соответствует двум и более шаблонам, предпочтение отдается шаблону, указанному в программе Lex первым.

Пример 3.12. Первое правило гласит, что необходимо продолжать чтение букв и цифр для поиска наибольшего префикса из этих символов, который и будет представлять собой искомый идентификатор. Оно же гласит, что <= следует рассматривать как единую лексему, а не как две лексемы, < и =. Второе правило делает ключевые слова зарезервированными, если они перечислены до **id** в программе Lex. Например, если наибольший префикс входной строки, соответствующий какому-либо из шаблонов — then, и при этом шаблон then в программе предваряет шаблон {id}, как на рис. 3.23, то лексический анализатор возвращает токен THEN, а не токен ID. □

3.5.4 Прогностический оператор

Lex автоматически считывает один дополнительный символ за последним символом, образующим лексему, а затем выполняет возврат его во входной поток с тем, чтобы из потока была изъята только распознанная лексема. Однако иногда требуется, чтобы входная строка соответствовала определенному шаблону только тогда, когда за ней следуют некоторые (предопределенные) другие символы. В этом случае в шаблоне можно использовать символ косой черты, который указывает конец той части шаблона, которой должна соответствовать лексема. Все, что следует за /, представляет собой дополнительный шаблон, которому должен соответствовать остаток входной строки после найденной лексемы, чтобы было принято решение о том, что найденная лексема соответствует рассматриваемому токену. Однако часть входной строки, соответствующая второй части шаблона, не входит в найденную лексему и остается во входном потоке для последующей обработки.

Пример 3.13. В Fortran и некоторых других языках ключевые слова не являются зарезервированными. Это создает много проблем. Так, например, в инструкции

```
IF(I,J) = 3
```

IF представляет собой имя массива, а не ключевое слово. В противоположность этому в инструкции вида

```
IF ( condition ) THEN ...
```

IF представляет собой ключевое слово. К счастью, за ключевым словом IF всегда следуют левая скобка, некоторый текст (условие), который может содержать скобки, правая скобка и буква. Таким образом, правило Lex для ключевого слова IF можно записать как

```
IF / \( .* \) {letter}
```

Это правило гласит, что лексема, соответствующая шаблону, состоит из двух букв IF. Косая черта указывает, что имеется и дополнительный шаблон, не соответствующий лексеме. Первым символом этого шаблона является левая скобка. Поскольку скобка является метасимволом языка Lex, она должна быть предварена обратной косой чертой, указывающей, что имеется в виду буквальное значение данного символа. Точка со звездочкой означает "любая строка без символа начала новой строки". Точка в языке Lex представляет собой метасимвол, который означает "любой символ, кроме символа начала новой строки". Далее следует правая скобка, предваренная обратной косой чертой, обозначающей буквальное значение следующего за ней метасимвола. Наконец, за правой скобкой следует символ *letter*, который является регулярным определением, представляющим класс символов, состоящий из всех букв.

Для надежной работы этого шаблона из входного потока предварительно должны быть удалены пробельные символы. В шаблоне не приняты меры ни на случай наличия пробельных символов, ни на случай, когда условие не содержится целиком в одной строке, поскольку точка не соответствует символу новой строки.

Предположим, например, что выясняется соответствие этого шаблона следующему префиксу входной строки:

```
IF(A<(B+C)*D)THEN...
```

Первые два символа соответствуют `IF`, следующий — комбинации `\(`, девять символов после левой скобки — шаблону `.*`, а идущие за ними два символа соответствуют `\)` и *letter*. Обратите внимание на то, что первая правая скобка (после `C`) не рассматривается, поскольку за ней следует символ, не являющийся буквой. В результате мы делаем вывод, что буквы `IF` составляют лексему, являющуюся экземпляром токена **if**. □

3.5.5 Упражнения к разделу 3.5

Упражнение 3.5.1. Опишите, какие изменения надо внести в программу `Lex` на рис. 3.23 для того, чтобы

а) добавить ключевое слово `while`;

б) заменить множество операторов сравнения операторами сравнения языка программирования C;

в) позволить использовать в именах символ подчеркивания (_);

! г) добавить новый шаблон с токеном `STRING`. Этот шаблон состоит из двойных кавычек ("), произвольной строки символов и завершающих двойных кавычек. Если символ двойных кавычек содержится в строке, он должен быть предварен обратной косой чертой (\); соответственно, сама обратная косая черта в строке должна быть представлена двумя обратными косыми чертами. Лексическое значение представляет собой строку без охватывающих ее двойных кавычек и без символов обратной косой черты, использованных для изменения смысла символов двойных кавычек и обратной косой черты в строке на буквальный. Строки вносятся в отдельную таблицу строк.

Упражнение 3.5.2. Напишите программу `Lex`, которая копирует файл, заменяя все непустые последовательности пробельных символов одним пробелом.

Упражнение 3.5.3. Напишите программу `Lex`, которая копирует программу на языке программирования C, заменяя все вхождения ключевого слова `float` на `double`.

! **Упражнение 3.5.4.** Напишите программу `Lex`, которая преобразует текстовый файл в "поросячью латынь".[5] В частности, считаем, что файл представляет собой последовательность слов (групп символов), разделенных пробельными символами. Всякий раз, когда встречается слово, выполняются следующие действия.

1. Если первая буква согласная, она перемещается в конец слова и за ней добавляются буквы `ay`.

2. Если первая буква гласная, к концу слова просто добавляется `ay`.

Все небуквенные символы копируются без изменений.

! **Упражнение 3.5.5.** В SQL ключевые слова и идентификаторы нечувствительны к регистру. Напишите программу на языке `Lex`, которая распознает ключевые слова `SELECT`, `FROM` и `WHERE` (с любыми сочетаниями верхних и нижних регистров) и токен `ID`, который в данном упражнении может быть любой последовательностью букв и цифр, начинающейся с буквы. Вносить идентификаторы в таблицу символов не требуется, но следует указать, чем именно функция для внесения в таблицу символов отличается от таковой для идентификаторов, чувствительных к регистру, как на рис. 3.23.

3.6 Конечные автоматы

Сейчас мы рассмотрим, как `Lex` преобразует входную программу в лексический анализатор. Ключевым моментом этого преобразования является формализм, известный как *конечный автомат* (finite automata). Конечные автоматы, по сути, представляют собой графы, подобные диаграммам переходов, но с небольшими отличиями.

1. Конечные автоматы являются *распознавателями* (recognizer); они просто говорят "да" или "нет" для каждой возможной входной строки.

2. Конечные автоматы разделяются на два класса.

   а) *Недетерминированные конечные автоматы* (nondeterministic finite automata — NFA) не имеют ограничений на свои дуги. Символ может быть меткой нескольких дуг, исходящих из одного и того же состояния; кроме того, одна из возможных меток — пустая строка ϵ.

[5]Из англо-русского словаря: **pig Latin** — *детск. жарг.* "поросячья латынь" (манера коверкать слова, переставляя первый согласный звук в конец слова и добавляя слог "ay", например Oodgay orningmay = Good morning). — *Прим. пер.*

б) *Детерминированные конечные автоматы* (deterministic finite automata — DFA) для каждого состояния и каждого символа входного алфавита имеют ровно одну дугу с указанным символом, покидающим это состояние.

Как детерминированные, так и недетерминированные конечные автоматы способны распознавать одни и те же языки. По сути, это те же языки, именуемые *регулярными языками* (regular language), которые могут быть описаны регулярными выражениями.[6]

3.6.1 Недетерминированные конечные автоматы

Недетерминированный конечный автомат (НКА) состоит из

1) множества состояний S;

2) множества входных символов Σ (*входного алфавита*); считаем, что символ ϵ, обозначающий пустую строку, не является членом Σ;

3) *функции переходов*, которая для каждого состояния и каждого символа из $\Sigma \cup \{\epsilon\}$ дает множество *последующих состояний* (next state);

4) состояния s_0 из S, известного как *стартовое* (*начальное*);

5) множества состояний F, являющегося подмножеством S, известных как *допускающие* (*конечные*).

Как недетерминированный, так и детерминированный конечные автоматы можно представить в виде *графа переходов*, узлы которого представляют состояния, а помеченные дуги представляют функцию переходов. Дуга из состояния s в состояние t, помеченная как a, существует тогда и только тогда, когда t — одно из последующих состояний для состояния s и входного символа a. Этот граф очень похож на диаграмму переходов, за исключением того, что

а) один и тот же символ может помечать дуги, исходящие из одного состояния в несколько разных других;

б) дуга может быть помечена пустой строкой ϵ вместо символа входного алфавита (или вместе с ним).

[6] Здесь имеется небольшое упущение: в том виде, в котором мы их определили, регулярные выражения не могут описывать пустой язык, поскольку мы никогда не используем этот шаблон на практике. Однако конечный автомат *может* определять пустой язык. Теоретически $\emptyset$ рассматривается как дополнительное регулярное выражение для единственной цели — определения пустого языка.

Пример 3.14. На рис. 3.24 показан граф переходов НКА, распознающего язык регулярного выражения (**a** | **b**)* **abb**. Этот абстрактный язык, распознающий строки из a и b, заканчивающиеся подстрокой abb, будет использоваться в этом разделе и далее. Кстати, он похож на регулярные выражения, описывающие реальные языки, например на выражение, описывающее все файлы, имена которых заканчиваются на `.o` — **any***`.o`, где **any** обозначает любой печатный символ.

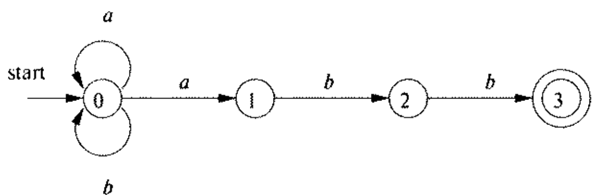


Рис. 3.24. Недетерминированный конечный автомат

В соответствии с принятыми соглашениями для диаграмм переходов двойной кружок вокруг состояния 3 указывает, что это — допускающее состояние. Единственный способ попасть из состояния 0 в допускающее — проследовать по некоторому пути, который некоторое время остается в состоянии 0, а затем проходит по состояниям 1, 2 и 3, считывая из входного потока символы abb. Таким образом, в допускающее состояние ведут только те строки, которые заканчиваются на abb. □

3.6.2 Таблицы переходов

НКА можно также представить в виде *таблицы переходов* (transition table), строки которой соответствуют состояниям, а столбцы — входным символам и ϵ. Запись для некоторого состояния и входного символа представляет собой значение функции переходов для данных аргументов. Если функция переходов не содержит информации о некоторой паре "состояние — входной символ", в таблице в этом месте помещается $\emptyset$.

Пример 3.15. На рис. 3.25 приведена таблица переходов для НКА, показанного на рис. 3.24. □

3.6.3 Принятие входной строки автоматом

НКА *допускает*, или *принимает* (accept), входную строку x тогда и только тогда, когда в графе переходов существует некоторый путь от начального состояния к одному из допускающих, такой, что метки дуг вдоль этого пути соответствуют строке x. Заметим, что метки ϵ на этом пути игнорируются, поскольку не вносят никакого вклада в рассматриваемую строку.

Состояние	a	b	ϵ
0	$\{0,1\}$	$\{0\}$	$\varnothing$
1	$\varnothing$	$\{2\}$	$\varnothing$
2	$\varnothing$	$\{3\}$	$\varnothing$
3	$\varnothing$	$\varnothing$	$\varnothing$

Рис. 3.25. Таблица переходов для НКА, показанного на рис. 3.24

Пример 3.16. Строка *aabb* принимается НКА на рис. 3.24. Путь, помеченный как *aabb*, из состояния 0 в состояние 3 демонстрирует этот факт:

$$0 \xrightarrow{a} 0 \xrightarrow{a} 1 \xrightarrow{b} 2 \xrightarrow{b} 3$$

Заметим, что может быть несколько путей для одной и той же строки, приводящих в разные состояния. Например,

$$0 \xrightarrow{a} 0 \xrightarrow{a} 0 \xrightarrow{b} 0 \xrightarrow{b} 0$$

представляет собой еще один путь из состояния 0, помеченный строкой *aabb*. Этот путь приводит в состояние 0, не являющееся допускающим. Вспомним, однако, что НКА принимает строку, если существует *некоторый* путь, помеченный этой строкой, из начального состояния в допускающее. Существование других путей, приводящих в непринимающие состояния, не имеет значения. □

Язык, определяемый (или *допускаемый*) *НКА*, представляет собой множество строк, помечающих некоторые пути от стартового состояния до допускающего. Как уже упоминалось, НКА на рис. 3.24 определяет тот же язык, что и регулярное выражение $(\mathbf{a} \mid \mathbf{b})^* \, \mathbf{abb}$, т.е. все строки из алфавита $\{a,b\}$, заканчивающиеся на *abb*. Для обозначения языка, принимаемого автоматом A, будем использовать обозначение $L(A)$.

Пример 3.17. На рис. 3.26 показан НКА, принимающий $L(\mathbf{aa}^* \mid \mathbf{bb}^*)$. Строка *aaa* принимается этим автоматом в силу существования пути

$$0 \xrightarrow{\varepsilon} 1 \xrightarrow{a} 2 \xrightarrow{a} 2 \xrightarrow{a} 2$$

Обратите внимание, что ϵ при конкатенация "исчезает", так что метка всего пути — *aaa*. □

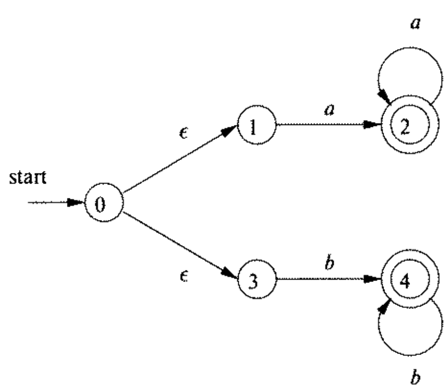


Рис. 3.26. НКА, принимающий **aa*** | **bb***

3.6.4 Детерминированный конечный автомат

Детерминированный конечный автомат (ДКА) представляет собой частный случай НКА, в котором

а) нет переходов для входа ϵ;

б) для каждого состояния s и входного символа a имеется ровно одна дуга, выходящая из s и помеченная a.

Если воспользоваться для представления ДКА таблицей переходов, то каждая запись в ней будет являться единственным состоянием (так что его можно указывать без фигурных скобок, означающих множество).

В то время как НКА — это абстрактное представление алгоритма для распознавания строк некоторого языка, ДКА является простым, конкретным алгоритмом распознавания строк. К счастью, каждое регулярное выражение и каждый НКА могут быть преобразованы в ДКА, принимающий тот же язык. Мы говорим "к счастью", поскольку при построении лексического анализатора реализуется или моделируется именно детерминированный конечный автомат. Приведенный далее алгоритм показывает применение ДКА к строке.

Алгоритм 3.18. Моделирование ДКА

ВХОД: входная строка x, завершенная символом конца файла **eof**, и детерминированный конечный автомат D с начальным состоянием s_0, принимающими состояниями F и функцией переходов *move*.

ВЫХОД: ответ "да", если D принимает x, и "нет" в противном случае.

МЕТОД: применение алгоритма, приведенного на рис. 3.27, ко входной строке x. Функция *move* (s, c) дает состояние, в которое из состояния s ведет дуга при

входном символе c. Функция *nextChar* возвращает очередной символ из входной строки x. □

```
s = s0;
c = nextChar();
while ( c != eof ) {
        s = move(s, c);
        c = nextChar();
}
if ( s ∈ F ) return "да";
else return "нет";
```

Рис. 3.27. Моделирование ДКА

Пример 3.19. На рис. 3.28 показан граф ДКА, принимающий язык $(\mathbf{a} \mid \mathbf{b})^* \mathbf{abb}$, тот же, что и в случае НКА на рис. 3.24. Для данной входной строки *ababb* этот ДКА проходит последовательность состояний 0, 1, 2, 1, 2, 3 и возвращает "да". □

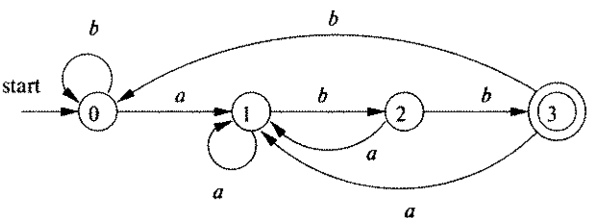


Рис. 3.28. ДКА, принимающий язык $(\mathbf{a} \mid \mathbf{b})^* \mathbf{abb}$

3.6.5 Упражнения к разделу 3.6

Упражнение 3.6.1. На рис. 3.19 приведен алгоритм вычисления функции отказа для алгоритма КМП. Покажите, как на базе данной функции отказа построить для ключевого слова $b_1 b_2 \dots b_n$ ДКА с $n + 1$ состояниями для распознавания $.^* b_1 b_2 \dots b_n$, где точка означает "произвольный символ". Кроме того, ДКА должен быть построен за время $O(n)$.

Упражнение 3.6.2. Разработайте конечные автоматы (детерминированные или недетерминированные) для каждого из языков из упражнения 3.3.5.

Упражнение 3.6.3. Укажите все пути, помеченные как *aabb*, для НКА на рис. 3.29. Принимает ли этот НКА строку *aabb*?

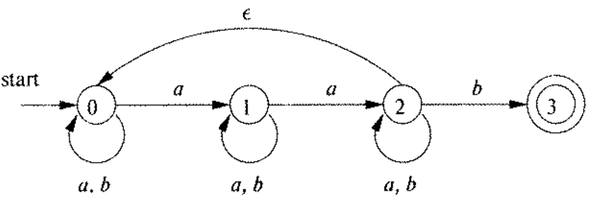


Рис. 3.29. НКА к упражнению 3.6.3

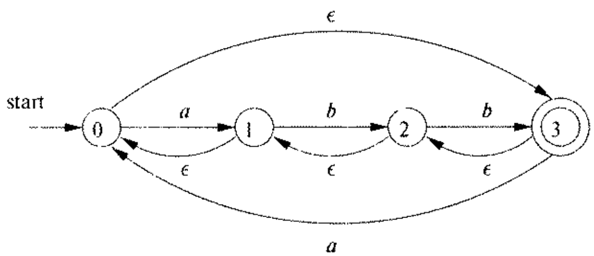


Рис. 3.30. НКА к упражнению 3.6.4

Упражнение 3.6.4. Повторите упражнение 3.6.3 для НКА на рис. 3.30.

Упражнение 3.6.5. Приведите таблицы переходов для НКА

а) из упражнения 3.6.3;

б) из упражнения 3.6.4;

в) на рис. 3.26.

3.7 От регулярных выражений к автоматам

Регулярное выражение представляет собой способ описания лексических анализаторов и другого программного обеспечения для работы с шаблонами. Однако реализация этого программного обеспечения требует моделирования ДКА (см. алгоритм 3.18) или, возможно, моделирования НКА. Поскольку при работе с НКА часто приходится делать выбор перехода для входного символа (например, на рис. 3.24 в состоянии 0 для символа a) или для ϵ (например, на рис. 3.26 в состоянии 0) либо даже выбор между переходом для реального входного символа или для ϵ, его моделирование существенно сложнее, чем моделирование ДКА. Таким образом, оказывается очень важной задача конвертации НКА в ДКА, который принимает тот же язык.

В этом разделе сначала будет показано, как конвертировать недетерминированные конечные автоматы в детерминированные. Затем мы воспользуемся методикой под названием "построение подмножеств" для получения практичного алгоритма для непосредственного моделирования недетерминированных конечных автоматов в ситуациях (отличных от лексического анализа), когда преобразование НКА в ДКА требует больше времени, чем непосредственное моделирование. Затем мы покажем, как преобразовывать регулярные выражения в недетерминированные конечные автоматы, из которых при желании могут быть построены детерминированные конечные автоматы. Завершится раздел обсуждением компромиссов между потреблением памяти и временем работы, присущих различным методам реализации регулярных выражений, и выяснением, как выбрать подходящий метод для конкретного приложения.

3.7.1 Преобразование НКА в ДКА

Общая идея, лежащая в основе построения подмножеств, заключается в том, что каждое состояние строящегося ДКА соответствует множеству состояний НКА. После чтения входной строки $a_1a_2 \ldots a_n$ ДКА находится в состоянии, соответствующем множеству состояний, которых может достичь из своего стартового состояния НКА по пути, помеченному $a_1a_2 \ldots a_n$.

Возможна ситуация, когда количество состояний ДКА экспоненциально зависит от количества состояний НКА, что может привести к сложностям при реализации такого ДКА. Однако одно из преимуществ применения подхода к лексическому анализу на основе автоматов заключается в том, что для реальных языков НКА и ДКА имеют примерно одинаковое количество состояний, без экспоненциального поведения.

Алгоритм 3.20. *Построение подмножества* (subset construction) ДКА из НКА

ВХОД: НКА N.

ВЫХОД: ДКА D, принимающий тот же язык, что и N.

МЕТОД: алгоритм строит таблицу переходов *Dtran* для D. Каждое состояние D представляет собой множество состояний НКА, и *Dtran* строится таким образом, чтобы "параллельно" моделировать все возможные переходы, которые N может выполнить для данной входной строки. Наша первая проблема — в корректной обработке ϵ-переходов N. На рис. 3.31 приведены определения некоторых функций, которые описывают базовые вычисления состояний N, необходимые для данного алгоритма. Здесь s — единственное состояние N, в то время как T — множество состояний N.

Мы должны исследовать те множества состояний, в которых N может оказаться после обработки некоторой входной строки. Начнем с того, что перед прочтением первого символа N может находиться в любом из состояний $\epsilon\text{-}closure(s_0)$,

Операция	Описание
ϵ-*closure* (s)	Множество состояний НКА, достижимых из состояния s при одном ϵ-переходе
ϵ-*closure* (T)	Множество состояний НКА, достижимых из состояния s из множества T при одном ϵ-переходе; $= \cup_{s \in T} \epsilon\text{-}closure\,(s)$
move (T, a)	Множество состояний НКА, в которые имеется переход из некоторого состояния $s \in T$ при входном символе a

Рис. 3.31. Операции над состояниями НКА

где s_0 — начальное состояние. Предположим, что после чтения строки x автомат N может находиться в множестве состояний T. Если следующий прочитанный символ — a, то N может перейти в любое из состояний *move* (T, a). Однако после чтения a может быть выполнено несколько ϵ-переходов; таким образом, после прочтения *xa* конечный автомат N может быть в любом состоянии из множества ϵ-*closure*(*move* (T, a)). Построение множества состояний *Dstates* конечного автомата D и его функции переходов *Dtran* в соответствии с этими идеями показано на рис. 3.32.

```
Изначально в Dstates содержится только одно состояние,
ϵ-closure (s0), и оно не помечено
while ( в Dstates имеется непомеченное состояние T ) {
      Пометить T;
      for ( каждый входной символ a ) {
            U = ϵ-closure (move (T, a));
            if ( U ∉ Dstates )
                  Добавить U в Dstates как непомеченное состояние;
            Dtran [T, a] = U;
      }
}
```

Рис. 3.32. Построение подмножества

Стартовым состоянием D является ϵ-*closure* (s_0), а принимающими состояниями D являются те множества состояний N, которые включают как минимум одно принимающее состояние N. Для завершения описания построения подмножества требуется показать, как вычислить ϵ-*closure* (T) для произвольного множества состояний T недетерминированного конечного автомата. Этот процесс показан на рис. 3.33 и представляет собой простой поиск в графе множества состояний. В данном случае представьте, что в графе имеются только ребра с метками ϵ. □

Поместить все состояния T в стек *stack*;
Инициализировать ϵ-*closure* (T) множеством T;
while (*stack* не пуст) {
    Снять со стека *stack* верхний элемент t;
    for (каждое состояние u с дугой из t в u, помеченной ϵ)
        if ($u \notin \epsilon$-*closure* (T)) {
            Добавить u в ϵ-*closure* (T);
            Поместить u в *stack*;
        }
}

Рис. 3.33. Вычисление ϵ-*closure* (T)

Пример 3.21. На рис. 3.34 показан еще один НКА, принимающий $(\mathbf{a} \mid \mathbf{b})^* \mathbf{abb}$; это автомат, который позже будет построен непосредственно из регулярного выражения. Давайте применим алгоритм 3.20 к рис. 3.34.

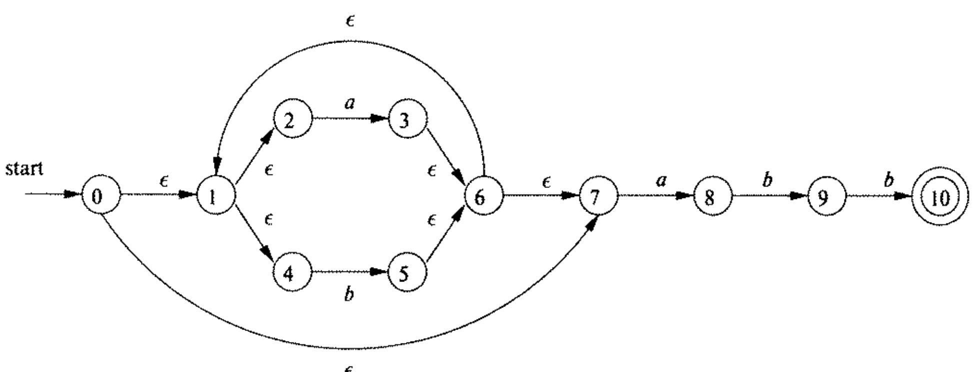


Рис. 3.34. НКА N для $(\mathbf{a} \mid \mathbf{b})^* \mathbf{abb}$

Стартовое состояние A эквивалентного ДКА — ϵ-*closure* (0), или $A = \{0, 1, 2, 4, 7\}$, так как это именно те состояния, в которые можно перейти из состояния 0 по пути, у которого все ребра имеют метки ϵ. Заметим, что путь может состоять из нуля дуг, так что состояние 0 достижимо само из себя по ϵ-пути.

Входной алфавит представляет собой $\{a, b\}$. Наш первый шаг состоит в том, чтобы пометить A и вычислить $Dtran[A, a] = \epsilon\text{-}closure(move(A, a))$ и $Dtran[A, b] = \epsilon\text{-}closure(move(A, b))$. Среди состояний 0, 1, 2, 4 и 7 только 2 и 7 имеют переходы по a в состояния 3 и 8 соответственно. Таким образом, $move\,(A, a) = \{3, 8\}$. Далее, ϵ-*closure* $(\{3, 8\}) = \{1, 2, 3, 4, 6, 7, 8\}$, так что мы можем найти

$$Dtran\,[A, a] = \epsilon\text{-}closure\,(move\,(A, a)) = \epsilon\text{-}closure\,(\{3, 8\}) = \{1, 2, 3, 4, 6, 7, 8\}$$

Назовем это множество B, так что $Dtran[A, a] = B$.

Теперь мы должны вычислить $Dtran[A, b]$. Среди состояний A только 4 имеет переход по b, и это переход в состояние 5. Таким образом,

$$Dtran[A, b] = \epsilon\text{-}closure(\{5\}) = \{1, 2, 4, 5, 6, 7\}$$

Назовем это множество C, так что $Dtran[A, b] = C$.

Состояния НКА	Состояния ДКА	a	b
$\{0, 1, 2, 4, 7\}$	A	B	C
$\{1, 2, 3, 4, 6, 7, 8\}$	B	B	D
$\{1, 2, 4, 5, 6, 7\}$	C	B	C
$\{1, 2, 4, 5, 6, 7, 9\}$	D	B	E
$\{1, 2, 4, 5, 6, 7, 10\}$	E	B	C

Рис. 3.35. Таблица переходов *Dtran* для ДКА D

Если продолжить процесс с непомеченными состояниями B и C, в конечном счете мы достигнем точки, в которой все состояния ДКА будут помечены. Это заключение справедливо, потому что имеется "всего" 2^{11} различных подмножеств множества из 11 состояний НКА. Фактически мы построили пять различных состояний ДКА, которые соответствуют множеству состояний НКА. Соответствующая таблица переходов показана на рис. 3.35, а граф переходов D — на рис. 3.36. Состояние A — начальное, а состояние E, которое содержит состояние 10 НКА, является единственным принимающим состоянием ДКА D.

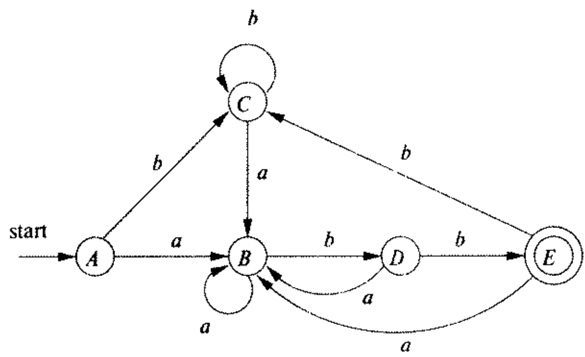


Рис. 3.36. Результат применения построения подмножеств к НКА на рис. 3.34

Обратите внимание, что ДКА D имеет на одно состояние больше, чем ДКА для того же языка на рис. 3.28. Состояния A и C имеют одну и ту же функцию

переходов, так что их можно объединить. Минимизацию количества состояний ДКА мы рассмотрим в разделе 3.9.6. □

3.7.2 Моделирование НКА

Стратегия, использующаяся в ряде текстовых редакторов, заключается в построении НКА из регулярного выражения и его моделировании с использованием методики, сходной с построением подмножеств "на лету". Далее приведен набросок схемы такого моделирования.

Алгоритм 3.22. Моделирование НКА

ВХОД: входная строка x с завершающим символом **eof**. НКА N с начальным состоянием s_0, принимающими состояниями F и функцией переходов *move*.

ВЫХОД: ответ "да", если N принимает x; ответ "нет" в противном случае.

МЕТОД: алгоритм поддерживает множество текущих состояний S, которые достигаются из s_0 по пути, помеченному считанными символами входной строки. Если c — очередной входной символ, считанный функцией *nextChar* (), то сначала вычисляется *move* (S, c), а затем — замыкание с применением ϵ-*closure* (). Набросок алгоритма приведен на рис. 3.37. □

```
1)  S = ϵ-closure(s0);
2)  c = nextChar();
3)  while ( c != eof ) {
4)          S = ϵ-closure(move(S, c));
5)          c = nextChar();
6)  }
7)  if ( S ∩ F != ∅ ) return "да";
8)  else return "нет";
```

Рис. 3.37. Моделирование НКА

3.7.3 Эффективность моделирования НКА

При аккуратной реализации алгоритм 3.22 может быть достаточно эффективным. Поскольку используемые в нем идеи полезны для применения в ряде подобных алгоритмов, включая поиск в графах, мы детально рассмотрим эту реализацию. Нам требуются следующие структуры данных.

1. Два стека, каждый из которых хранит множество состояний НКА. Один из стеков, *oldStates*, хранит "текущее" множество состояний, т.е. значение S в правой части строки 4 на рис. 3.37. Второй стек, *newStates*, хранит

"следующее" множество состояний — S в левой части строки 4. Этот шаг не показан, но при переходе в цикле от строки 3 к строке 6 *newStates* преобразуется в *oldStates*.

2. Булев массив *alreadyOn*, проиндексированный состояниями НКА, для указания, какие именно состояния входят в *newStates*. Хотя массив и стек хранят одну и ту же информацию, существенно быстрее опросить значение *alreadyOn* $[s]$, чем выполнять поиск состояния s в стеке *newStack*. Оба представления поддерживаются только для повышения эффективности.

3. Двумерный массив *move* $[s, a]$, хранящий таблицу переходов НКА. Записи этой таблицы, являющиеся множествами состояний, представлены связанными списками.

Для реализации строки 1 на рис. 3.37 нам надо установить все элементы массива *alreadyOn* равными FALSE, затем для каждого состояния s в ϵ-*closure* (s_0) поместить s в *oldStates* и установить *alreadyOn* $[s]$ равным TRUE. Реализация этой операции над состоянием s, а также реализация строки 4 облегчается при помощи вспомогательной функции $addState\,(s)$. Эта функция вносит состояние s в *newStates*, устанавливает значение *alreadyOn* $[s]$ равным TRUE и рекурсивно вызывает саму себя для состояний из *move* $[s, \epsilon]$ для вычисления ϵ-*closure* (s). Чтобы избежать дублирования, функцию *addState* нельзя вызывать для состояния, которое уже находится в стеке *newStates*. Набросок этой функции приведен на рис. 3.38.

```
 9)  addState(s) {
10)          Внести s в newStates;
11)          alreadyOn[s] = TRUE;
12)          for ( t из move[s, ϵ] )
13)                  if ( !alreadyOn(t) )
14)                          addState(t);
15)  }
```

Рис. 3.38. Добавление нового состояния s, отсутствующего в *newStates*

Строка 4 на рис. 3.37 реализуется путем просмотра каждого состояния из *oldStates*. Сначала мы находим множество состояний *move* $[s, c]$, где c — очередной входной символ, и к каждому из этих состояний, которое еще не входит в *newStates*, мы применяем функцию *addState*. Обратите внимание, что *addState* вычисляет ϵ-*closure* и добавляет все найденные состояния в *newStates*, если они еще там не находятся. Набросок этих действий показан на рис. 3.39.

Теперь предположим, что НКА N имеет n состояний и m переходов, т.е. m — сумма по всем состояниям количества символов (или ϵ), для которых имеется

```
16)  for ( s ∈ oldStates ) {
17)      for ( t ∈ move[s,c] )
18)          if ( !alreadyOn[t] )
19)              addState(t);
20)      Снять s со стека oldStates;
21)  }

22)  for ( s ∈ newStates ) {
23)      Снять s со стека newStates;
24)      Поместить s в oldStates;
25)      alreadyOn[s] = FALSE;
26)  }
```

Рис. 3.39. Реализация шага 4 на рис. 3.37

переход из данного состояния. Не учитывая вызов *addState* в строке 19 на рис. 3.39, мы находим, что время, затраченное на выполнение строк цикла 16–21, равно $O(n)$. Иными словами, мы можем выполнить цикл не более n раз, на каждой итерации выполняя постоянную работу, за исключением времени, затрачиваемого на вызов *addState*. То же самое можно сказать и о цикле в строках 22–26.

При выполнении кода на рис. 3.39, т.е. шага 4 на рис. 3.37, вызов *addState* для каждого состояния возможен не более одного раза. Причина этого в том, что, когда мы вызываем *addState* (s), в строке 11 на рис. 3.38 выполняется присваивание значения TRUE элементу массива *alreadyOn* $[s]$. Это значение при проверках в строках 13 на рис. 3.38 и 18 на рис. 3.39 предотвращает повторные вызовы функции *addState* для одного и того же состояния.

Время, затрачиваемое на один вызов *addState*, исключая рекурсивный вызов самой себя, равно $O(1)$ для строк 10 и 11. Для строк 12 и 13 время зависит от количества ϵ-переходов из состояния s. Мы не знаем этого количества для данного состояния, но знаем, что общее количество переходов для всех состояний равно m, так что общее время, затраченное в строках 12–13 для всех вызовов *addState* в процессе одного выполнения кода на рис. 3.39, составляет $O(m)$. Общее время, затраченное на все остальные шаги *addState*, равно $O(n)$, поскольку оно представляет собой константу для каждого вызова, а всего таких вызовов не больше n.

Итак, можно сделать вывод, что при корректной реализации время, затрачиваемое на выполнение строки 4 на рис. 3.37, составляет $O(n+m)$. Остальные строки цикла while с 3 по 6 требуют $O(1)$ времени на одну итерацию. Если входная строка x имеет длину k, то общее время работы данного цикла — $O(k(n+m))$. Строка 1 на рис. 3.37 может быть выполнена за время $O(n+m)$, поскольку она,

О-обозначения

Выражение наподобие $O(n)$ представляет собой сокращение для "не более чем некоторая константа, умноженная на n". Технически мы говорим, что некоторая функция $f(n)$ (вероятно, время работы некоторого алгоритма) представляет собой $O(g(n))$, если существуют константы c и n_0, такие, что для всех $n \geqslant n_0$ выполняется соотношение $f(n) \leqslant cg(n)$. Полезная идиома $O(1)$ означает "некоторая константа". Использование таких *О-обозначений* позволяет не вдаваться в детальный подсчет времени выполнения, ограничиваясь указанием скорости роста времени выполнения алгоритма.

по сути, представляет собой действия с рис. 3.39, когда *oldStates* содержит единственное состояние s_0. Строки 2, 7 и 8 требуют $O(1)$ времени каждая. Таким образом, время выполнения корректно реализованного алгоритма 3.22 составляет $O(k(n+m))$, т.е. пропорционально произведению длины входной строки на размер (количество узлов плюс количество дуг) графа переходов.

3.7.4 Построение НКА из регулярного выражения

Приведем теперь алгоритм для преобразования произвольного регулярного выражения в НКА, определяющий тот же язык. Этот алгоритм синтаксически управляемый, в том смысле, что он рекурсивно работает с деревом разбора регулярного выражения. Для каждого подвыражения алгоритм строит НКА с единственным принимающим состоянием.

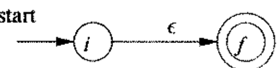
Алгоритм 3.23. Алгоритм Мак-Нотона–Ямады–Томпсона (McNaughton–Yamada–Thompson) для преобразования регулярного выражения в НКА

ВХОД: регулярное выражение r над алфавитом Σ.

ВЫХОД: НКА N, принимающий $L(r)$.

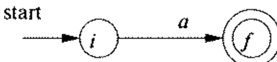
МЕТОД: начнем с разбора r на составляющие подвыражения. Правила построения НКА состоят из базисных правил для обработки подвыражений без операторов и индуктивных правил для построения бóльших НКА из НКА для непосредственных подвыражений данного выражения.

БАЗИС: для выражения ϵ строим НКА.



Здесь i — новое состояние, представляющее собой начальное состояние НКА, а f — другое новое состояние, являющееся принимающим состоянием НКА.

Для любого подвыражения a из Σ строим НКА



Здесь также i и f — новые состояния, являющиеся соответственно начальным и принимающим. Обратите внимание, что в обоих базовых случаях мы строим отдельные НКА с новыми состояниями для каждого ϵ и некоторого a, являющихся подвыражением r.

Индукция: предположим, что $N(s)$ и $N(t)$ — недетерминированные конечные автоматы для регулярных выражений s и t соответственно.

а) Пусть $r = s \mid t$. Тогда $N(r)$, НКА для r, строится так, как показано на рис. 3.40. Здесь i и f — новые состояния, являющиеся соответственно начальным и принимающим состояниями $N(r)$. Имеются также ϵ-переходы от i к стартовым состояниям $N(s)$ и $N(t)$ и ϵ-переходы от каждого из их принимающих состояний в f. Обратите внимание, что принимающие состояния $N(s)$ и $N(t)$ не являются таковыми в $N(r)$. Поскольку любой путь из i в f должен пройти либо исключительно по $N(s)$, либо исключительно по $N(t)$ и поскольку метки пути не изменяются при добавлении к ним ϵ при выходе из i и входе в f, можно заключить, что $N(r)$ принимает язык $L(s) \cup L(t)$, который представляет собой не что иное, как язык $L(r)$. Таким образом, на рис. 3.40 представлено корректное построение НКА для оператора объединения.

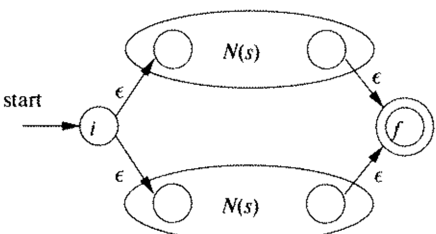


Рис. 3.40. НКА для объединения двух регулярных выражений

б) Пусть $r = st$. В этом случае построим $N(r)$ так, как показано на рис. 3.41. Начальное состояние $N(s)$ становится начальным состоянием $N(r)$, а принимающее состояние $N(t)$ — единственным принимающим состоянием $N(r)$. Принимающее состояние $N(s)$ и начальное состояние $N(t)$ объединяются в одно состояние со всеми входящими и исходящими переходами обоих состояний. Путь от i к f на рис. 3.41 должен сначала пройти через $N(s)$, так что его метка начинается с некоторой строки из $L(s)$. Затем путь

проходит через $N(t)$, так что его метка заканчивается строкой из $L(t)$. Как будет показано, принимающие состояния не имеют исходящих дуг, а начальные состояния — входящих, так что после того, как путь покинул $N(s)$, он не может вновь войти в него. Таким образом, $N(r)$ принимает исключительно строки из $L(s)\,L(t)$ и является корректным НКА для $r = st$.

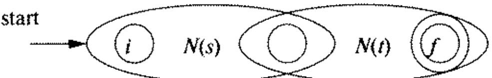


Рис. 3.41. НКА для конкатенации двух регулярных выражений

в) Пусть $r = s^*$. Тогда для r НКА $N(r)$ строится так, как показано на рис. 3.42. Здесь i и f — новые начальное и принимающее состояния $N(r)$. Для достижения f из i необходимо либо пройти по пути с меткой ϵ, соответствующему строке $L(s)^0$, либо перейти к начальному состоянию $N(s)$, пройти по этому НКА и вернуться из его принимающего состояния в начальное нуль или несколько раз. Эта возможность позволяет $N(r)$ принимать все строки $L(s)^1$, $L(s)^2$ и так далее, так что полное множество строк, принимаемое $N(r)$, представляет собой $L(s^*)$.

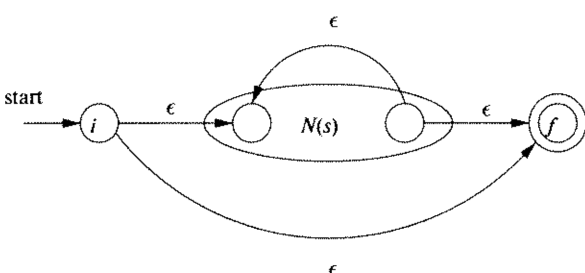


Рис. 3.42. НКА для замыкания регулярного выражения

г) Наконец, пусть $r = (s)$. Тогда $L(r) = L(s)$ и можно использовать НКА $N(s)$ как $N(r)$. □

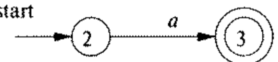
Описание алгоритма 3.23 содержит указания о том, почему корректны индуктивные построения. Мы не даем формального доказательства их корректности, но перечислим некоторые свойства построенных НКА в дополнение к тому главному факту, что $N(r)$ принимает язык $L(r)$. Эти свойства интересны как сами по себе, так и с точки зрения использования в формальном доказательстве.

1. Количество состояний в $N(r)$ не более чем в два раза превышает количество операторов и операндов в r. Эта граница следует из того факта, что каждый шаг алгоритма создает не более двух новых узлов.

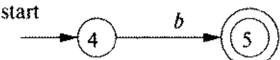
2. $N(r)$ имеет одно начальное и одно принимающее состояние. Принимающее состояние не имеет исходящих переходов, а начальное — входящих.

3. Каждое состояние $N(r)$, отличное от принимающего, имеет либо один исходящий переход по символу из Σ, либо два исходящих перехода, оба по ϵ.

Пример 3.24. Воспользуемся алгоритмом 3.23 для построения НКА для $r =$ $= (\mathbf{a} \mid \mathbf{b})^* \mathbf{abb}$. На рис. 3.43 показано дерево разбора r, подобное деревьям разбора для арифметических выражений из раздела 2.2.3. Для подвыражения r_1 — первого **a** — мы строим НКА



Номера состояний выбраны такими для согласования с дальнейшими стадиями построения НКА. Для r_2 строим



Теперь можно объединить $N(r_1)$ и $N(r_2)$ с использованием построения на рис. 3.40 и получить НКА для $r_3 = r_1 \mid r_2$; этот НКА показан на рис. 3.44.

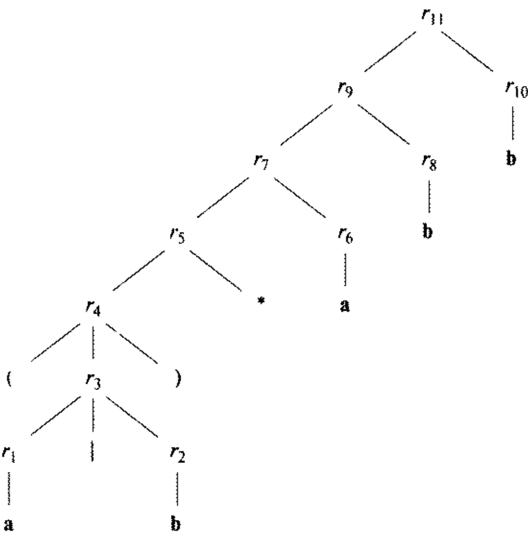


Рис. 3.43. Дерево разбора для $(\mathbf{a} \mid \mathbf{b})^* \mathbf{abb}$

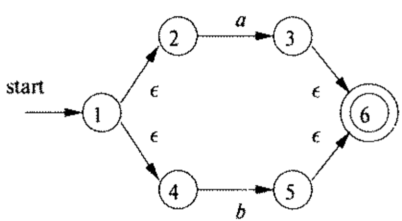


Рис. 3.44. НКА для r_3

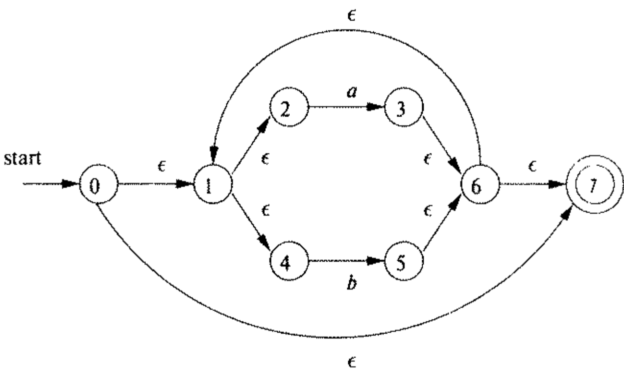
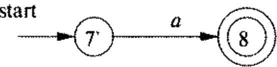


Рис. 3.45. НКА для r_5

НКА для $r_4 = (r_3)$ имеет тот же вид, что и для r_3. Конечный автомат для $r_5 = (r_3)^*$ приведен на рис. 3.45. Для его получения из НКА на рис. 3.44 было использовано построение, показанное на рис. 3.42.

Рассмотрим теперь подвыражение r_6, представляющее собой **a**. Для него мы опять воспользуемся базисным построением, но с новыми состояниями. Мы не можем повторно использовать НКА для r_1, несмотря на то, что r_1 и r_6 представляют собой одно и то же выражение. НКА для r_6 имеет следующий вид:



Для получения НКА для $r_7 = r_5 r_6$ мы применяем построение, показанное на рис. 3.41. Мы объединяем состояния 7 и 7′, что дает НКА, показанный на рис. 3.46. Продолжая работу с новым НКА для двух подвыражений **b** — r_8 и r_{10}, — в конечном счете мы построим НКА для регулярного выражения $(\mathbf{a} \mid \mathbf{b})^* \mathbf{abb}$, с которым уже встречались на рис. 3.34. □

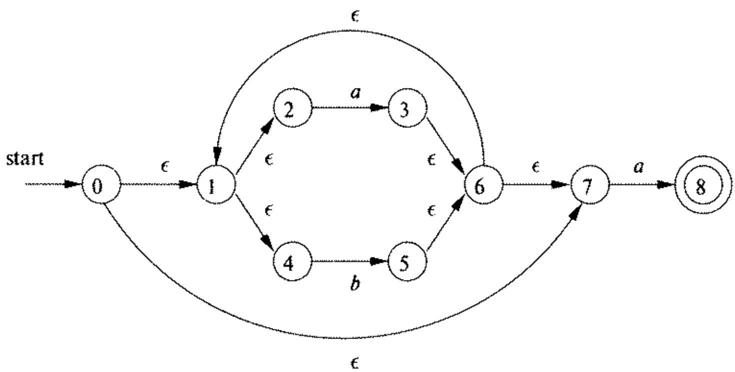


Рис. 3.46. НКА для r_7

3.7.5 Эффективность алгоритма обработки строк

Мы уже знаем, что алгоритм 3.18 обрабатывает строку x за время $O(|x|)$, а в разделе 3.7.3 мы выяснили, что можно смоделировать работу НКА за время, пропорциональное произведению $|x|$ на размер графа переходов НКА. Очевидно, что моделирование работы ДКА выполняется быстрее, чем НКА, так что у вас может возникнуть вопрос — какой вообще смысл в моделировании НКА?

Одна из причин в том, что построение подмножеств для НКА в наихудшем случае может вызвать экспоненциальный рост количества состояний. Хотя, в принципе, количество состояний ДКА не влияет на время работы алгоритма 3.18, оно может оказаться столь велико, что не будет помещаться в основной памяти, а это приведет к необходимости дисковых операций и существенному снижению эффективности.

Пример 3.25. Рассмотрим семейство языков, описываемое регулярными выражениями вида $L_n = (\mathbf{a} \mid \mathbf{b})^* \, \mathbf{a} \, (\mathbf{a} \mid \mathbf{b})^{n-1}$, т.е. каждый язык L_n состоит из строк из a и b, таких, что n-й символ, считая от правого конца строки, равен a. Легко построить НКА с $n+1$ состояниями. Конечный автомат может при любых входных символах как находиться в начальном состоянии, так и при входном символе a переходить в состояние 1. Из состояния 1 при любом входном символе НКА переходит в состояние 2, и так далее, пока не будет достигнуто состояние n. Данный НКА показан на рис. 3.47.

Однако любой ДКА для языка L_n должен иметь как минимум 2^n состояний. Мы не будем доказывать этот факт, но основная идея заключается в том, что любой ДКА для этого выражения должен отслеживать как минимум n символов входного потока; в противном случае может быть получен неверный ответ. Ясно, что для отслеживания всех возможных последовательностей из n символов a и b требуется по крайней мере 2^n состояний. К счастью, как уже упоминалось, такого

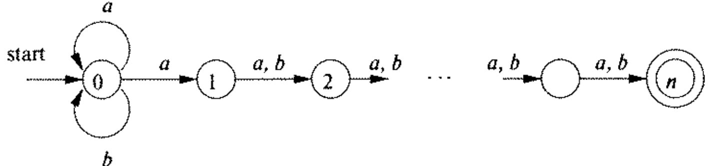


Рис. 3.47. НКА, имеющий существенно меньшее количество состояний, чем наименьший эквивалентный ДКА

рода выражения встречаются в приложениях лексического анализа не так часто, так что вряд ли вам придется на практике встретиться с таким ДКА с огромным количеством состояний. □

Однако генераторы лексических анализаторов и другие системы обработки строк часто начинают работу с регулярного выражения. Мы оказываемся перед выбором — преобразовывать регулярные выражения в ДКА или в НКА. Дополнительная стоимость перехода от регулярного выражения к ДКА, таким образом, по существу, представляет собой стоимость выполнения алгоритма 3.20 над НКА (можно непосредственно перейти от регулярного выражения к ДКА, но при этом, по сути, выполняется то же количество работы). Если обработка строк выполняется многократно, как в случае лексического анализа, то преобразование в ДКА стоит того. Однако в других приложениях обработки строк, таких как `grep`, в которых пользователь определяет одно регулярное выражение и один или несколько файлов, в которых выполняется поиск шаблона этого выражения, может оказаться более эффективным опустить построение ДКА и непосредственно моделировать работу НКА.

Рассмотрим стоимость преобразования регулярного выражения r в НКА при помощи алгоритма 3.23. Ключевым шагом является построение дерева разбора для r. В главе 4 приводится несколько методов построения этого дерева за линейное время, т.е. за время $O(|r|)$, где $|r|$ означает *размер* r — сумму количества операторов и операндов в r. Легко также убедиться, что все базовые и индуктивные построения алгоритма 3.23 требуют константного времени, так что общее время, необходимое для преобразования регулярного выражения в НКА, составляет $O(|r|)$.

Кроме того, как мы выяснили в разделе 3.7.4, строящийся НКА имеет не более $2|r|$ состояний и не более $4|r|$ переходов, т.е. в терминах анализа из раздела 3.7.3 $n \leqslant |r|$ и $m \leqslant 2|r|$. Таким образом, моделирование этого НКА для входной строки x потребует времени $O(|r| \times |x|)$. Это время является доминирующим по отношению ко времени построения НКА, $O(|r|)$, так что мы можем сделать вывод, что для данных регулярного выражения r и строки x можно выяснить, принадлежит ли x языку $L(r)$, за время $O(|r| \times |x|)$.

Время, требующееся для построения подмножеств, сильно зависит от количества состояний в получающемся ДКА. Для начала заметим, что при построении подмножеств на рис. 3.32 ключевой шаг, построение множества состояний U на основе множества состояний T и входного символа a, очень похож на построение нового множества состояний из старого множества при моделировании работы НКА в алгоритме 3.22. Но мы уже выяснили, что при корректной реализации этот шаг требует времени, не более чем пропорционального количеству состояний и переходов в НКА.

Предположим, что мы начинаем работу с регулярным выражением r и преобразуем его в НКА. Этот НКА имеет не более $2\,|r|$ состояний и не более $4\,|r|$ переходов. Кроме того, имеется не более $|r|$ входных символов. Таким образом, для каждого создаваемого состояния ДКА мы должны построить не более $|r|$ новых состояний, и каждое из них требует не более чем $O\left(|r| + 2\,|r|\right)$ времени. Таким образом, время, необходимое для построения ДКА с s состояниями, — $O(|r|^2\,s)$.

В распространенном случае, когда s примерно равно $|r|$, построение подмножеств занимает время $O(|r|^3)$. Однако в наихудшем случае, как в примере 3.25, это время оказывается равным $O(|r|^2\,2^{|r|})$. На рис. 3.48 приведена информация о разных вариантах построения распознавателя для выяснения принадлежности одной или нескольких строк x языку $L\left(r\right)$.

Автомат	Начальное построение	Работа над строкой
НКА	$O\left(\|r\|\right)$	$O\left(\|r\| \times \|x\|\right)$
ДКА: типичный случай	$O(\|r\|^3)$	$O\left(\|x\|\right)$
ДКА: наихудший случай	$O(\|r\|^2\,2^{\|r\|})$	$O\left(\|x\|\right)$

Рис. 3.48. Стоимость начального построения и обработки одной строки различных методов распознавания языка регулярного выражения

Если доминирует стоимость обработки одной строки, как в случае построения лексического анализатора, очевидно, что следует предпочесть ДКА. Однако в программах наподобие `grep`, в которых автомат работает только с одной строкой, обычно предпочтительнее использовать НКА. Пока $|x|$ не становится порядка $|r|^3$, нет смысла даже думать о преобразовании в ДКА.

Существует, однако, составная стратегия, которая оказывается столь же хорошей, как наилучшая из стратегий НКА и ДКА для каждого регулярного выражения r и строки x. Начнем с моделирования НКА, но по мере вычисления будем запоминать состояния НКА (т.е., по сути, состояния ДКА) и их переходы. Перед обработкой текущего множества состояний НКА и сходного символа сначала проверим, не был ли уже вычислен соответствующий переход, и, если был вычислен, воспользуемся уже готовой информацией.

3.7.6 Упражнения к разделу 3.7

Упражнение 3.7.1. Преобразуйте в ДКА НКА, приведенный на

а) рис. 3.26;

б) рис. 3.29;

в) рис. 3.30.

Упражнение 3.7.2. Воспользуйтесь алгоритмом 3.22 для моделирования НКА, приведенного на

а) рис. 3.29;

б) рис. 3.30,

для входной строки *aabb*.

Упражнение 3.7.3. Преобразуйте приведенные ниже регулярные выражения в детерминированные конечные автоматы с использованием алгоритмов 3.23 и 3.20:

а) $(\mathbf{a} \mid \mathbf{b})^*$;

б) $(\mathbf{a}^* \mid \mathbf{b}^*)^*$;

в) $((\epsilon \mid \mathbf{a})\,\mathbf{b}^*)^*$;

г) $(\mathbf{a} \mid \mathbf{b})^*\mathbf{abb}\,(\mathbf{a} \mid \mathbf{b})^*$.

3.8 Разработка генератора лексических анализаторов

В этом разделе мы применим представленные в разделе 3.7 методы для того, чтобы разобраться в архитектуре генераторов лексических анализаторов, таких как Lex. Мы рассмотрим два подхода, основанных на НКА и ДКА; последний из них, по сути, и является реализацией Lex.

3.8.1 Структура генерируемого анализатора

На рис. 3.49 приведена архитектура лексического анализатора, генерируемого Lex. Программа, служащая в качестве лексического анализатора, включает фиксированную программу, моделирующую автомат; пока что мы оставим открытым вопрос о том, какой именно автомат — детерминированный или недетерминированный — моделируется. Остальная часть лексического анализатора состоит из компонентов, которые создаются самим Lex из программы Lex.

Эти компоненты следующие.

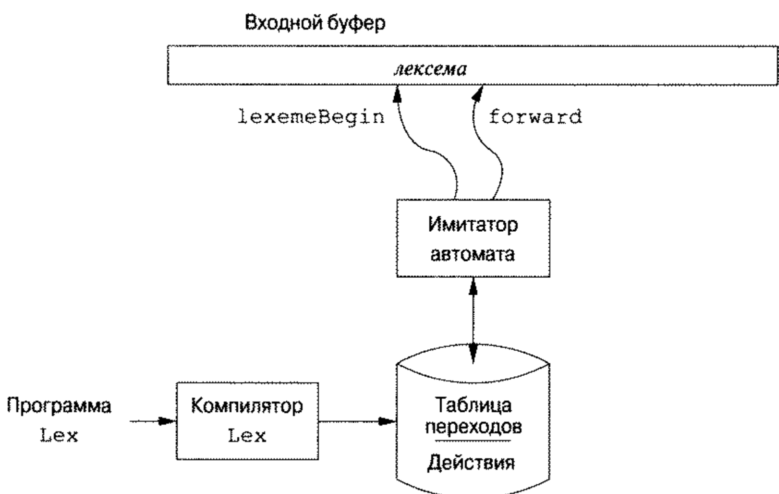


Рис. 3.49. Программа Lex превращается в таблицу переходов и действия, используемые имитатором конечного автомата

1. Таблица переходов автомата.
2. Функции, которые передаются непосредственно на выход генератора (см. раздел 3.5.2).
3. Действия из входной программы, которые представляют собой фрагменты кода, вызываемые имитатором автомата в соответствующие моменты времени.

Построение автомата начнем с поочередного превращения регулярных выражений из программы Lex в недетерминированные конечные автоматы с использованием алгоритма 3.23. Нам нужен один автомат, который будет распознавать лексемы, соответствующие любому шаблону в программе, так что мы объединим все НКА в один, вводя новое начальное состояние с ϵ-переходами в каждое из стартовых состояний НКА N_i для шаблона p_i. Эта конструкция показана на рис. 3.50.

Пример 3.26. Проиллюстрируем эти идеи на следующем простом абстрактном примере:

a	{ действие A_1 для шаблона p_1 }
abb	{ действие A_2 для шаблона p_2 }
a*b⁺	{ действие A_3 для шаблона p_3 }

Обратите внимание на наличие конфликтов, о которых мы говорили в разделе 3.5.3. В частности, строка *abb* соответствует как второму, так и третьему шаблону, но ее следует рассматривать как лексему для шаблона p_2, поскольку этот

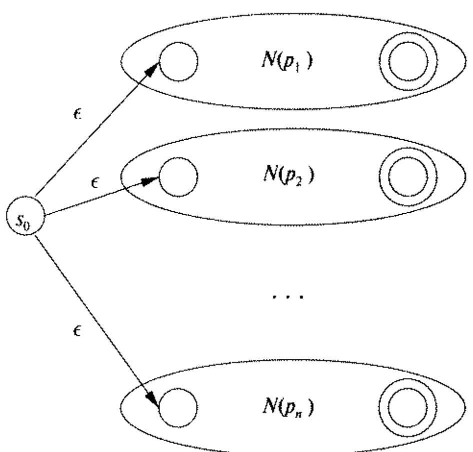


Рис. 3.50. НКА, построенный на основе программы Lex

шаблон указан в приведенной программе Lex первым. Далее, входные строки наподобие $aabbb\cdots$ имеют много префиксов, соответствующих третьему шаблону. Правило Lex гласит, что должен быть выбран самый длинный префикс, так что чтение b должно продолжаться до тех пор, пока не встретится a, после чего можно сообщить о том, что найдена лексема из начальных a, за которыми следуют b в обнаруженном количестве.

На рис. 3.51 показаны три НКА, распознающие указанные три шаблона. Третий представляет собой упрощенную версию автомата, полученного при применении алгоритма 3.23. На рис. 3.52 показан комбинированный НКА, полученный путем добавления начального состояния 0 и трех ϵ-переходов. □

3.8.2 Распознавание шаблонов на основе НКА

Если лексический анализатор моделирует НКА, такой как на рис. 3.52, то он начинает чтение входного потока с точки, которую мы обозначаем как *lexemeBegin*. При перемещении указателя *forward* по входному потоку при помощи алгоритма 3.22 выполняется вычисление множества состояний для каждой точки потока.

В конечном счете моделирование НКА достигает точки входного потока, для которой нет следующего состояния. Очевидно, что в этот момент не приходится рассчитывать на то, что некоторый более длинный префикс приведет НКА в принимающее состояние; более того, множество состояний всегда будет пустым. Таким образом, в этот момент обнаружен наиболее длинный префикс, представляющий собой лексему, соответствующую некоторому шаблону.

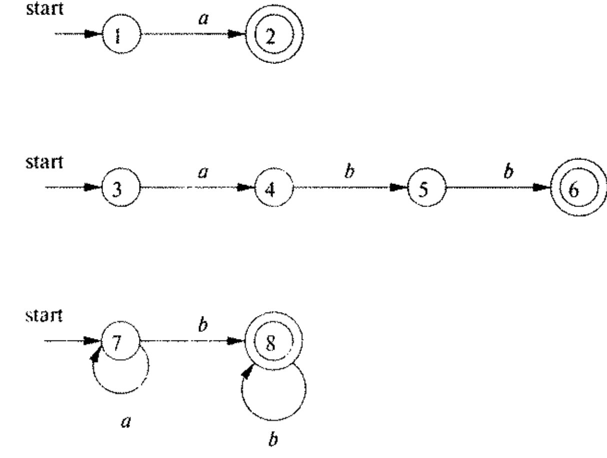


Рис. 3.51. НКА для a, abb и $\mathbf{a}^*\mathbf{b}^+$

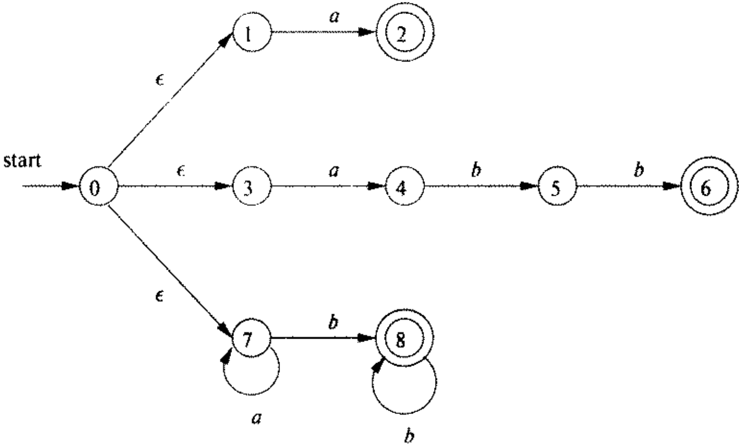


Рис. 3.52. Комбинированный НКА

Далее выполняется откат по последовательности состояний, пока не будет найдено множество, которое включает одно или несколько принимающих состояний. Если в таком множестве имеется несколько принимающих состояний, выбирается то из них, которое связано с наиболее ранним шаблоном p_i в списке программы Lex. Указатель *forward* перемещается назад к концу лексемы и выполняется действие A_i, связанное с шаблоном p_i.

Пример 3.27. Предположим, что у нас есть шаблоны из примера 3.26 и входная строка, начинающаяся с *aaba*. На рис. 3.53 показаны множества состояний НКА, приведенного на рис. 3.52, в которые мы попадаем, начиная с ϵ-*closure* для

начального состояния 0 (представляющего собой множество $\{0, 1, 3, 7\}$), и через которые следуем при обработке указанной входной строки. После чтения четвертого входного символа мы оказываемся в пустом множестве состояний, поскольку из состояния 8 на рис. 3.52 для входного символа a переходов нет.

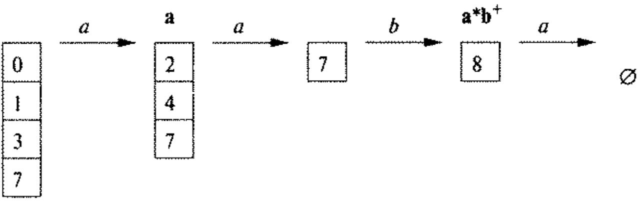


Рис. 3.53. Последовательность множеств состояний при обработке входной строки aa*b*a

Таким образом, мы должны вернуться назад в поисках множества состояний, в которое входит принимающее состояние. Заметим, что, как показано на рис. 3.53, после чтения a мы оказываемся в множестве состояний, которое включает состояние 2 и, следовательно, указывает на соответствие шаблону **a**. Однако после чтения aab мы попадаем в состояние 8, указывающее на соответствие шаблону $\mathbf{a}^*\mathbf{b}^+$; префикс aab — самый длинный префикс, приводящий нас в принимающее состояние. Таким образом, мы выбираем aab в качестве лексемы и выполняем действие A_3, которое должно включать возврат синтаксическому анализатору указания о том, что найден токен, шаблон которого — $p_3 = \mathbf{a}^*\mathbf{b}^+$. □

3.8.3 ДКА для лексических анализаторов

Другая архитектура использует преобразование НКА для всех шаблонов в эквивалентный ДКА с использованием построения подмножеств по алгоритму 3.20. В каждом состоянии ДКА при наличии в нем одного или нескольких принимающих состояний НКА определяется первый шаблон программы `Lex`, представленный принимающим состоянием, и этот шаблон является выходом данного состояния ДКА.

Пример 3.28. На рис. 3.54 показана диаграмма переходов, основанная на разработанном при помощи метода построения подмножеств ДКА, соответствующем НКА на рис. 3.52. Принимающие состояния помечены шаблонами, идентифицируемыми этими состояниями. Например, состояние $\{6, 8\}$ содержит два принимающих состояния, соответствующих шаблонам **abb** и $\mathbf{a}^*\mathbf{b}^+$. Поскольку шаблон **abb** указан в программе `Lex` первым, именно он является шаблоном, связанным с состоянием $\{6, 8\}$. □

ДКА в лексическом анализаторе используется практически так же, как и НКА. Мы моделируем работу ДКА до тех пор, пока не окажемся в некоторой точке,

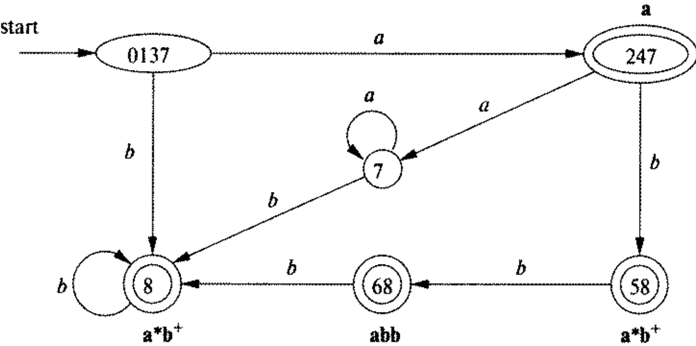


Рис. 3.54. Граф переходов для ДКА, обрабатывающего шаблоны a, abb и $\mathbf{a^*b^+}$

у которой нет следующего состояния (или, строго говоря, у которой следующее состояние — $\emptyset$, *тупиковое состояние* (dead state), соответствующее пустому множеству состояний НКА). Из этой точки мы должны выполнить откат по пройденной последовательности состояний и, как только при откате будет встречено принимающее состояние ДКА, выполняется действие, соответствующее шаблону этого состояния.

Пример 3.29. Предположим, что ДКА на рис. 3.54 передана входная строка *abba*. В этом случае последовательность состояний ДКА, через которую мы проходим, — 0137, 247, 58, 68 и для последнего a во входной строке из состояния 68 перехода нет. Теперь мы рассматриваем полученную последовательность от конца к началу и обнаруживаем, что первым принимающим состоянием является 68, соответствующее шаблону $p_2 = \mathbf{abb}$. □

3.8.4 Реализация прогностического оператора

Вспомним, что в разделе 3.5.4 мы столкнулись с тем, что иногда в шаблоне Lex r_1/r_2 необходимо использовать прогностический оператор /, поскольку для корректной идентификации лексемы шаблону r_1 некоторого токена может требоваться дополнительный контекст r_2. При преобразовании r_1/r_2 в НКА мы рассматриваем /, как если бы это было ϵ, так что в действительности мы не ищем / во входной строке. Однако, если НКА распознает префикс xy во входном буфере как соответствующий данному регулярному выражению, конец лексемы находится не там, где НКА входит в принимающее состояние. Конец лексемы находится в состоянии s, таком, что

1) s имеет ϵ-переход для (воображаемого) символа /;

Тупиковые состояния в ДКА

Технически автомат на рис. 3.54 — не совсем ДКА. Причина этого в том, что ДКА имеет переход из каждого состояния для каждого входного символа алфавита. Здесь же мы опустили переходы в тупиковое состояние $\emptyset$ и, соответственно, переходы из тупикового состояния в него же для любого входного символа. Предыдущие примеры преобразования НКА в ДКА не имели путей из начального состояния в $\emptyset$, однако в случае НКА на рис. 3.52 такой путь имеется.

Однако при построении ДКА для использования в лексическом анализаторе очень важно рассматривать тупиковое состояние отдельно, поскольку мы должны знать, когда все возможности распознавания более длинной лексемы исчерпаны. Таким образом, мы предлагаем всегда как опускать переходы в тупиковое состояние, так и удалять само тупиковое состояние. В действительности проблема сложнее, чем кажется, поскольку построение ДКА из НКА может привести к нескольким состояниям, из которых невозможно достичь ни одного принимающего состояния, и мы должны знать, когда при моделировании окажется достигнуто одно из этих состояний. В разделе 3.9.6 рассматривается объединение всех таких состояний в одно тупиковое, так что их идентификация становится очень простой. Интересно также заметить, что при построении ДКА из регулярного выражения с использованием алгоритмов 3.20 и 3.23 получающийся автомат не имеет ни одного состояния, кроме $\emptyset$, из которого нельзя попасть в принимающее состояние.

2) существует путь из начального состояния НКА в состояние s, который соответствует строке x;

3) существует путь из состояния s в принимающее состояние, который соответствует строке y;

4) строка x имеет наибольшую длину среди всех возможных xy, удовлетворяющих условиям 1–3.

Если в НКА имеется ровно одно состояние с ϵ-переходом для воображаемого /, то конец лексемы, как проиллюстрировано в следующем примере, определяется последним входом в это состояние. Если в НКА больше одного состояния с ϵ-переходом для воображаемого /, то поиск корректного состояния s существенно усложняется.

Пример 3.30. На рис. 3.55 показан НКА для шаблона с прогностическим оператором, соответствующего `IF` в языке программирования Fortran из примера 3.13.

Обратите внимание, что ϵ-переход из состояния 2 в состояние 3 представляет прогностический оператор. Состояние 6 указывает наличие ключевого слова `IF`. Однако сама лексема `IF` определяется обратным сканированием от состояния 6 к последнему вхождению в состояние 2. □

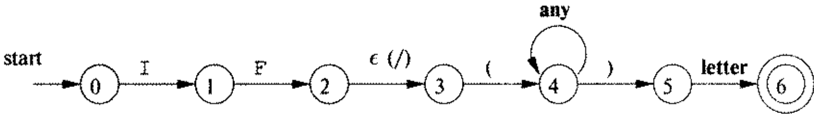


Рис. 3.55. НКА, распознающий ключевое слово IF

3.8.5 Упражнения к разделу 3.8

Упражнение 3.8.1. Предположим, имеется два токена: 1) ключевое слово `if` и 2) идентификаторы, которые представляют собой строки из букв, отличные от `if`. Постройте для этих токенов

а) НКА;

б) ДКА.

Упражнение 3.8.2. Повторите упражнение 3.8.1 для токенов, представляющих собой 1) ключевое слово `while`, 2) ключевое слово `when` и 3) идентификаторы, представляющие собой строки из букв и цифр, начинающиеся с буквы.

! **Упражнение 3.8.3.** Предположим, мы изменили определение ДКА так, что из каждого состояния для каждого входного символа может быть нуль или один переход (в отличие от ровно одного перехода в стандартном определении ДКА). В этом случае некоторые регулярные выражения будут иметь меньшие по размеру “ДКА”, чем при стандартном определении. Приведите пример такого регулярного выражения.

!! **Упражнение 3.8.4.** Разработайте алгоритм для распознавания шаблонов с прогностическим оператором вида r_1/r_2, где r_1 и r_2 представляют собой регулярные выражения. Покажите, как ваш алгоритм работает со следующими шаблонами:

а) $(\mathbf{abcd} \mid \mathbf{abc})/\mathbf{d}$;

б) $(\mathbf{a} \mid \mathbf{ab})/\mathbf{ba}$;

в) $\mathbf{aa}^*/\mathbf{a}^*$.

3.9 Оптимизация распознавателей на основе ДКА

В этом разделе будут представлены три алгоритма, использующиеся для реализации и оптимизации построенных на основе регулярных выражений распознавателей шаблонов.

1. Первый алгоритм используется в компиляторе Lex, поскольку он строит ДКА непосредственно из регулярного выражения, без создания промежуточного НКА. Получающийся в результате ДКА имеет меньшее количество состояний, чем ДКА, строящийся на основании НКА.

2. Второй алгоритм минимизирует количество состояний любого ДКА путем объединения состояний, которые имеют одно и то же поведение. Этот алгоритм сам по себе достаточно эффективен; время его работы — $O(n \log n)$, где n — количество состояний ДКА.

3. Третий алгоритм дает более компактное представление таблиц переходов по сравнению со стандартной двумерной таблицей.

3.9.1 Важные состояния НКА

Перед тем как приступить к рассмотрению непосредственного перехода от регулярных выражений к ДКА, сначала следует исследовать построение НКА при помощи алгоритма 3.23 и рассмотреть роли, которые играют различные состояния автоматов. Будем называть состояние НКА *важным*, если оно имеет исходящий не-ϵ-переход. Заметим, что в процессе построения подмножеств (алгоритм 3.20) используются только важные состояния в множестве T при вычислении ϵ-*closure* (*move* (T, a)), множества состояний, достижимых из T при входном символе a. Таким образом, множество состояний *move* (s, a) непустое, только если состояние s — важное. В процессе построения подмножеств два множества состояний НКА могут отождествляться, т.е. рассматриваться как единое множество, если они

1) имеют одни и те же важные состояния;

2) либо оба содержат принимающие состояния, либо оба их не содержат.

При построении НКА из регулярных выражений при помощи алгоритма 3.23 единственными важными состояниями являются те, которые созданы базисом алгоритма в качестве начальных для конкретных символов в регулярных выражениях; т.е. каждое важное состояние соответствует некоторому операнду в регулярном выражении.

Построенный НКА имеет только одно принимающее состояние, но это состояние, у которого нет исходящих переходов, важным не является. Добавляя к регулярному выражению r справа уникальный ограничитель #[7], мы даем принимающему состоянию для r переход по символу #, делая это состояние важным в НКА для $(r)\#$. Другими словами, используя *расширенное* (augmented) регулярное выражение $(r)\#$, можно забыть о принимающих состояниях при построении подмножеств; когда построение будет завершено, любое состояние с переходом по символу # должно быть принимающим.

Важные состояния НКА соответствуют позициям в регулярном выражении, в которых хранятся символы алфавита. Как мы увидим, это удобно для представления регулярного выражения его *синтаксическим деревом*, в котором листья соответствуют операндам, а внутренние узлы — операторам. Внутренний узел называется *cat*-узлом, *or*-узлом или *star*-узлом, если он помечен соответственно оператором конкатенации (.), объединения (|) или звездочкой (*). Синтаксическое дерево для регулярного выражения может быть построено так же, как для арифметического выражения в разделе 2.5.1.

Пример 3.31. На рис. 3.56 показано синтаксическое дерево для регулярного выражения для нашего текущего примера регулярного выражения. Cat-узлы представлены кружка́ми. □

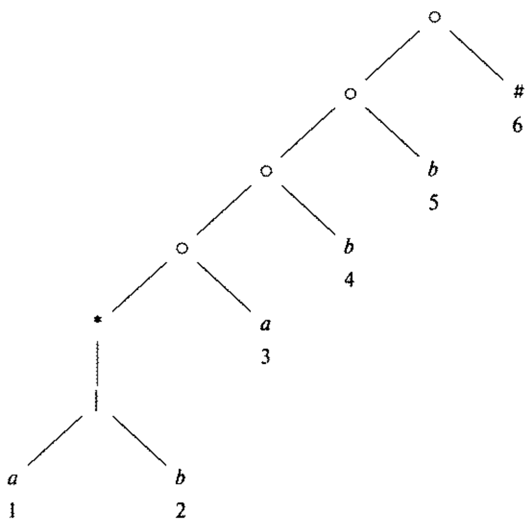


Рис. 3.56. Синтаксическое дерево для $(\mathbf{a} \mid \mathbf{b})^* \mathbf{abb}\#$

Листья синтаксического дерева помечаются символом ϵ или символами алфавита. Каждому листу, не помеченному ϵ, приписывается целое значение, уникаль-

[7]Этот символ уникален в том смысле, что не встречается во входном потоке. — *Прим. ред.*

ное в пределах дерева. Оно используется, как *позиция* листа, а также как позиция его символа. Заметим, что символ может иметь несколько позиций; например, на рис. 3.56 a имеет позиции 1 и 3. Позиции в синтаксическом дереве соответствуют важным состояниям построенного НКА.

Пример 3.32. На рис. 3.57 показан НКА для того же регулярного выражения, что и на рис. 3.56. Важные состояния пронумерованы, остальные представлены буквами. Пронумерованные состояния в НКА и позиции в синтаксическом дереве связаны друг с другом способом, который будет рассмотрен ниже. □

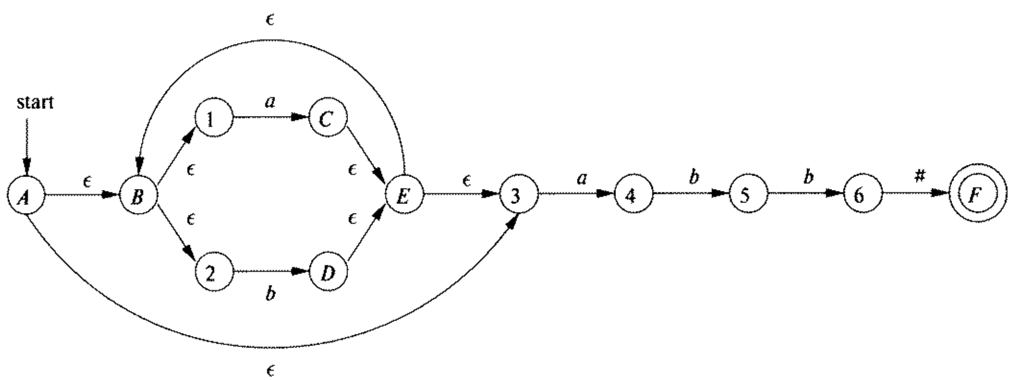


Рис. 3.57. НКА, построенный алгоритмом 3.23 для $(\mathbf{a} \mid \mathbf{b})^* \mathbf{abb}\#$

3.9.2 Функции, вычисляемые на синтаксическом дереве

Для построения ДКА непосредственно из регулярного выражения мы строим его синтаксическое дерево, а затем вычисляем четыре функции *nullable*, *firstpos*, *lastpos* и *followpos*, определяемые следующим образом (каждое определение использует синтаксическое дерево для расширенного регулярного выражения $(r)\,\#$).

1. Значение *nullable* (n) для узла n синтаксического дерева равно **true** тогда и только тогда, когда подвыражение, представленное n, содержит в своем языке ϵ. Иначе говоря, подвыражение может быть "сделано нулевым" или пустой строкой, хотя оно может представлять и другие, непустые строки.

2. *firstpos* (n) представляет собой множество позиций в поддереве с корнем n, которые соответствуют первому символу как минимум одной строки в языке подвыражения с корнем n.

3. *lastpos* (n) представляет собой множество позиций в поддереве с корнем n, которые соответствуют последнему символу как минимум одной строки в языке подвыражения с корнем n.

4. *followpos* (p) для позиции p представляет собой множество позиций q в синтаксическом дереве в целом, для которых существует строка $x = a_1 a_2 \ldots a_n$ языка $L((r)\#)$, обладающая тем свойством, что для некоторого i a_i соответствует позиции p, а a_{i+1} — позиции q.

Пример 3.33. Рассмотрим cat-узел n на рис. 3.56, который соответствует выражению $(\mathbf{a} \mid \mathbf{b})^* \mathbf{a}$. Мы утверждаем, что *nullable* (n) равно **false**, поскольку этот узел генерирует все строки из a и b, оканчивающиеся на a; он не может генерировать ϵ. С другой стороны, у star-узла ниже него значение функции *nullable* равно **true**, поскольку наряду со строками из a и b он генерирует и пустую строку ϵ.

firstpos $(n) = \{1, 2, 3\}$. В типичной сгенерированной строке наподобие aa первая позиция строки соответствует позиции 1 дерева, а первая позиция строки ba соответствует второй позиции дерева. Однако если сгенерированная строка представляет собой просто a, то это a соответствует третьей позиции в дереве.

lastpos $(n) = \{3\}$. Не имеет значения, какая именно строка генерируется из выражения для узла n, — последняя позиция в строке представляет собой a, получающееся из третьей позиции дерева.

Вычислить значение *followpos* несколько сложнее, но вскоре мы познакомимся с правилами, которые облегчают решение этой задачи. Здесь же мы просто приведем пример рассуждений для вычисления *followpos* $(1) = \{1, 2, 3\}$. Рассмотрим строку $\cdots ac \cdots$, где c — либо a, либо b, причем первое a получается из позиции 1 в дереве, т.е. это a является одним из символов, генерируемых **a** в выражении $(\mathbf{a} \mid \mathbf{b})^*$. За этим a может следовать другое a или b из того же подвыражения, т.е. в этом случае c получается из позиций 1 или 2 дерева. Может также оказаться, что это a — последнее в строке, сгенерированной выражением $(\mathbf{a} \mid \mathbf{b})^*$, и в этом случае символ c должен представлять собой a, получающееся из позиции 3 дерева. Следовательно, за позицией 1 могут следовать позиции 1, 2 и 3. □

3.9.3 Вычисление nullable, firstpos и lastpos

Вычислить *nullable*, *firstpos* и *lastpos* можно при помощи простой рекурсии по высоте дерева. Базисные и индуктивные правила для *nullable* и *firstpos* приведены на рис. 3.58. Правила для вычисления *lastpos*, по сути, те же, что и для вычисления *firstpos*, но в правиле для cat-узла роли дочерних узлов c_1 и c_2 должны поменяться.

Пример 3.34. Из всех узлов на рис. 3.56 значение функции *nullable* равно **true** только для star-узла. Из рис. 3.58 видно, что ни один лист не имеет значение функции *nullable*, равное **true**, так как все они соответствуют операндам, не являющимся ϵ. Or-узел также не может давать значение *nullable*, равное **true**, поскольку этого не делает ни один из его дочерних узлов. Star-узел имеет значение **true**, поскольку это свойство любого star-узла. Наконец, в cat-узле функция *nullable*

Узел n	*nullable* (n)	*firstpos* (n)
n — лист, помеченный ϵ	**true**	$\emptyset$
n — лист с позицией i	**false**	$\{i\}$
or-узел $n = c_1 \mid c_2$	*nullable* (c_1) **or** *nullable* (c_2)	*firstpos* $(c_1) \cup$ *firstpos* (c_2)
cat-узел $n = c_1 c_2$	*nullable* (c_1) **and** *nullable* (c_2)	**if** (*nullable* (c_1)) *firstpos* $(c_1) \cup$ *firstpos* (c_2) **else** *firstpos* (c_1)
star-узел $n = c_1^*$	**true**	*firstpos* (c_1)

Рис. 3.58. Правила для вычисления *nullable* и *firstpos*

принимает значение **false**, если таково значение функции хотя бы в одном из его дочерних узлов.

Вычисления функций *firstpos* и *lastpos* для каждого из узлов показаны на рис. 3.59 (значение *firstpos* (n) показано слева от узла n, а значение *lastpos* (n) — справа). Каждый лист в качестве значений функций *firstpos* и *lastpos* имеет сам себя в соответствии с правилом для не-ϵ-листьев на рис. 3.58. Значения функций для or-узлов представляют собой объединения значений функций в дочерних узлах. Правило для star-узла гласит, что значения функций *firstpos* и *lastpos* в нем те же, что и в его единственном дочернем узле.

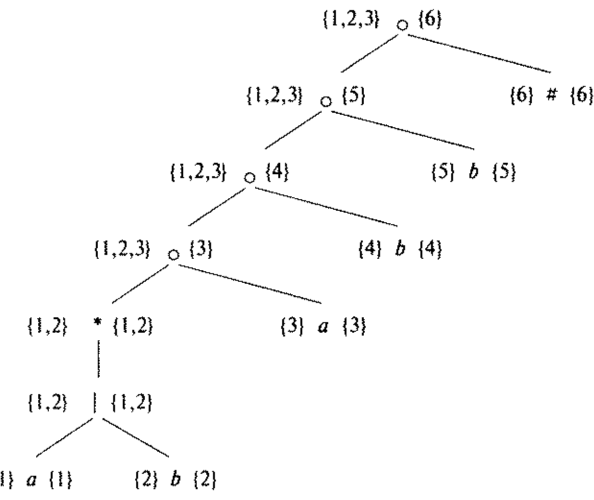


Рис. 3.59. Значения функций *firstpos* и *lastpos* в узлах синтаксического дерева для $(\mathbf{a} \mid \mathbf{b})^* \mathbf{abb}\#$

Теперь перейдем к самому нижнему cat-узлу, который назовем n. Вычисление *firstpos* (n) начнем с проверки значения функции *nullable* в его левом дочернем узле. В нашем случае оно равно **true**, а значит, *firstpos* в узле n представляет собой объединение значений *firstpos* в дочерних узлах, т.е. равно $\{1, 2\} \cup \{3\} = \{1, 2, 3\}$. Правила для *lastpos* в явном виде на рис. 3.58 не показаны, но, как уже говорилось, они такие же, как и для *firstpos*, но с взаимообменом дочерних узлов, т.е. вычисление *lastpos* мы должны начать с проверки значения функции *nullable* в правом дочернем узле (узел с позицией 3 в нашем случае). Поскольку оно равно **false**, значение *lastpos* (n) то же, что и значение *lastpos* в правом дочернем узле, т.е. $\{3\}$. □

3.9.4 Вычисление followpos

Наконец, мы должны выяснить, как же вычислять значения функции *followpos*. Имеется два пути следования одной позиции регулярного выражения за другой.

1. Если n — *cat*-узел с левым потомком c_1 и правым c_2, то для каждой из позиций i из *lastpos* (c_1) все позиции из *firstpos* (c_2) содержатся в *followpos* (i).

2. Если n — *star*-узел и i — позиция в *lastpos* (n), то все позиции из *firstpos* (n) содержатся в *followpos* (i).

Пример 3.35. Продолжим наш пример, для которого на рис. 3.59 были вычислены значения функций *firstpos* и *lastpos*. Правило 1 для *followpos* требует от нас, чтобы мы просмотрели каждый cat-узел и поместили каждую позицию из *firstpos* его правого дочернего узла в *followpos* для каждой позиции из *lastpos* его левого дочернего узла. Для самого нижнего cat-узла на рис. 3.59 это правило гласит, что позиция 3 находится в *followpos* (1) и *followpos* (2). Рассмотрение следующего, находящегося выше, cat-узла позволяет сделать вывод, что позиция 4 находится в *followpos* (3), а двух оставшихся cat-узлов — что 5 входит в *followpos* (4), а 6 — в *followpos* (5).

Мы должны также применить правило 2 к star-узлу. Это правило гласит, что позиции 1 и 2 находятся как в *followpos* (1), так и в *followpos* (2), поскольку для этого узла и *firstpos*, и *lastpos* равны $\{1, 2\}$. Полностью множества *followpos* показаны на рис. 3.60. □

Функцию *followpos* можно представить в виде ориентированного графа с узлами для каждой позиции и дугами, идущими из позиции i в позицию j тогда и только тогда, когда $j \in$ *followpos* (i). На рис. 3.61 показан такой ориентированный граф для функции *followpos* с рис. 3.60.

Не должно удивлять то, что ориентированный граф для *followpos* практически является НКА без ϵ-переходов для исходного регулярного выражения, и будет полностью таковым, если

Узел n	*followpos* (n)
1	$\{1, 2, 3\}$
2	$\{1, 2, 3\}$
3	$\{4\}$
4	$\{5\}$
5	$\{6\}$
6	$\varnothing$

Рис. 3.60. Значения функции *followpos*

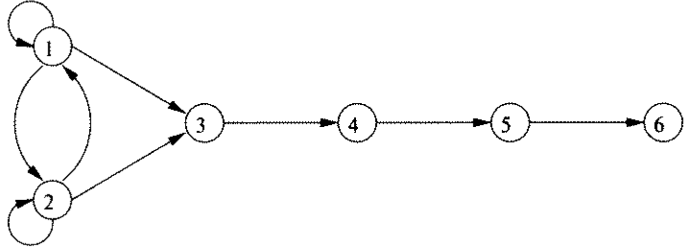


Рис. 3.61. Ориентированный граф для функции *followpos*

1) сделать все позиции из *firstpos* корня начальными состояниями;

2) пометить каждую дугу из i в j символом из позиции i;

3) сделать позицию, связанную с ограничителем #, единственным принимающим состоянием.

3.9.5 Преобразование регулярного выражения непосредственно в ДКА

Алгоритм 3.36. Построение ДКА из регулярного выражения r

Вход: регулярное выражение r.

Выход: ДКА D, распознающий язык $L(r)$.

Метод:

1. Построить синтаксическое дерево T из расширенного регулярного выражения $(r)\#$.

2. Вычислить для дерева T функции *nullable*, *firstpos*, *lastpos* и *followpos* с использованием методов из разделов 3.9.3 и 3.9.4.

3. Построить *Dstates* — множество состояний ДКА D и функцию переходов ДКА *Dtran* в соответствии с процедурой, приведенной на рис. 3.62. Состояния D представляют собой множества позиций T. Изначально все состояния "непомечены"; "помеченным" состояние становится непосредственно перед тем, как мы рассматриваем его исходящие переходы. Начальным состоянием D является *firstpos* (n_0), где узел n_0 — корень T. Принимающими являются состояния, которые содержат позицию для символа ограничителя #. □

```
Инициализируем Dstates единственным непомеченным
        состоянием firstpos(n0), где n0 — корень
        синтаксического дерева T для (r) #,
while ( в Dstates имеется непомеченное состояние S ) {
        Пометить S;
        for ( каждый входной символ a ) {
                Пусть U — объединение followpos (p)
                        для всех p из S, которые соответствуют a;
                if ( U ∉ Dstates )
                        Добавить U в качестве непомеченного
                                состояния в Dstates;
                Dtran [S, a] = U;
        }
}
```

Рис. 3.62. Построение ДКА непосредственно из регулярного выражения

Пример 3.37. Соберем воедино все ранее рассмотренные примеры и построим ДКА для регулярного выражения $r = (\mathbf{a} \mid \mathbf{b})^* \mathbf{abb}$. Синтаксическое дерево для $(r)\,\#$ показано на рис. 3.56. Мы уже знаем, что функция *nullable* имеет значение **true** только в star-узле. Функции *firstpos* и *lastpos* показаны на рис. 3.59, а значения *followpos* — на рис. 3.60.

Значение функции *firstpos* в корне равно $\{1, 2, 3\}$, так что это множество является начальным состоянием D. Обозначим это множество через A. Мы должны вычислить *Dtran* $[A, a]$ и *Dtran* $[A, b]$. Среди позиций A символу a соответствуют позиции 1 и 3, символу b — позиция 2. Таким образом, *Dtran* $[A, a] =$ *followpos*$(1) \cup$ $\cup$ *followpos*$(3) = \{1, 2, 3, 4\}$, а *Dtran* $[A, b] =$ *followpos* $(2) = \{1, 2, 3\}$. Последнее состояние представляет собой состояние A, так что его не надо добавлять в *Dstates*, но состояние $B = \{1, 2, 3, 4\}$ является новым, так что мы добавляем его в *Dstates* и вычисляем его переходы. Полностью ДКА показан на рис. 3.63. □

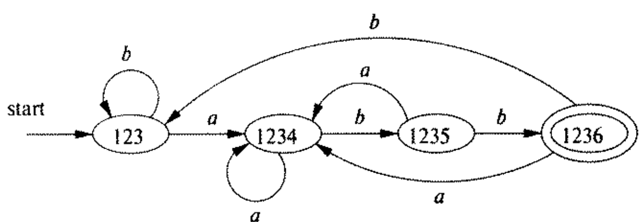


Рис. 3.63. ДКА для регулярного выражения $r = (\mathbf{a} \mid \mathbf{b})^* \mathbf{abb}$

3.9.6 Минимизация количества состояний ДКА

Один и тот же язык могут распознавать разные ДКА. Например, оба автомата, показанные на рис. 3.36 и 3.63, распознают язык $L((\mathbf{a} \mid \mathbf{b})^* \mathbf{abb})$. Такие автоматы могут иметь не только состояния с разными именами, но и разное количество состояний. При реализации лексического анализатора при помощи ДКА в общем случае предпочтительно иметь конечный автомат с минимально возможным количеством состояний, поскольку каждое состояние требует наличия записей в таблице, описывающей лексический анализатор.

Имена состояний значения не имеют. Мы будем говорить, что два автомата *одинаковы с точностью до имен состояний*, если один из них может быть получен из другого простым переименованием состояний. Однако существует тесная взаимосвязь между состояниями каждого из автоматов. Состояния A и C на рис. 3.36 в действительности эквивалентны, в том смысле что ни одно из них не является принимающим, и любой входной символ приводит к одинаковым переходам — в состояние B для входного символа a и в состояние C для входного символа b. Кроме того, и состояние A, и состояние C ведут себя так же, как и состояние 123 на рис. 3.63. Аналогично состояние B на рис. 3.36 ведет себя так же, как состояние 1234 на рис. 3.63, состояние D — как состояние 1235, а состояние E — как 1236.

Оказывается, для любого регулярного языка всегда существует единственный (с точностью до имен состояний) ДКА с минимальным количеством состояний. Более того, этот ДКА с минимальным количеством состояний может быть построен из любого ДКА для того же самого языка путем объединения множеств эквивалентных состояний. В случае языка $L((\mathbf{a} \mid \mathbf{b})^* \mathbf{abb})$ ДКА с минимальным количеством состояний представлен на рис. 3.63, и он может быть получен из ДКА на рис. 3.36 путем группирования состояний следующим образом: $\{A, C\}\,\{B\}\,\{D\}\,\{E\}$.

Чтобы понять, как работает алгоритм преобразования ДКА в эквивалентный ДКА с минимальным количеством состояний, нам надо рассмотреть, как входные строки отличают одно состояние от другого. Мы говорим, что строка x *отличает* (distinguish) состояние s от состояния t, если ровно одно из состояний, дости-

жимых из s и t по пути с меткой x, является принимающим, а другое — нет. Состояние s *отличимо* от состояния t, если существует некоторая строка, которая отличает их.

Пример 3.38. Пустая строка отличает любое принимающее состояние от непринимающего. На рис. 3.36 строка bb отличает состояние A от состояния B, поскольку bb приводит из A в непринимающее состояние C, а из B — в принимающее состояние E. □

Алгоритм, минимизирующий количество состояний, работает путем разбиения состояний ДКА на группы неотличимых состояний. Каждая группа состояний затем объединяется в одно состояние ДКА с минимальным количеством состояний. Алгоритм работает с разбиением, группы которого представляют собой множества состояний, отличие которых друг от друга пока что не установлено. Если никакая группа разбиения не может быть разделена на меньшие группы, мы получаем ДКА с минимальным количеством состояний.

Изначально разбиение состоит из двух групп: принимающие состояния и непринимающие. Основной шаг состоит в том, чтобы взять некоторую группу состояний, скажем, $A = \{s_1, s_2, \ldots, s_k\}$, и некоторый входной символ a и посмотреть, может ли a использоваться для отличия некоторых состояний в группе A. Мы рассматриваем переходы из каждого из состояний $s_1, s_2, \ldots, s_k$ для данного входного символа a, и если состояния переходят в две или более групп текущего разбиения, то A разделяется на набор групп таким образом, чтобы s_i и s_j оказывались в одной группе тогда и только тогда, когда они переходят в одну и ту же группу при данном входном символе a. Этот процесс повторяется до тех пор, пока не окажется ни одной группы, которая могла бы быть разделена некоторым входным символом. Эта идея формализована в следующем алгоритме.

Алгоритм 3.39. Минимизация количества состояний ДКА

Вход: ДКА D с множеством состояний S, входным алфавитом Σ, начальным состоянием s_0 и множеством принимающих состояний F.

Выход: ДКА D', принимающий тот же язык, что и D, и имеющий наименьшее возможное количество состояний.

Метод:

1. Начинаем с разбиения Π на две группы, F и $S - F$, состоящие из принимающих и не принимающих состояний D.

2. Применяем процедуру, приведенную на рис. 3.64, для построения нового разбиения Π_{new}.

3. Если $\Pi_{\text{new}} = \Pi$, принимаем $\Pi_{\text{final}} = \Pi$ и переходим к шагу 4. В противном случае переходим к шагу 2, заменяя Π на Π_{new}.

```
Изначально Π_new = Π;
for ( каждая группа G из Π ) {
    Разбиваем G на подгруппы, такие, что два состояния, s и t,
        находятся в одной подгруппе тогда и только тогда, когда
        для всех входных символов a состояния s и t имеют
        переходы по этому символу в состояния, принадлежащие
        одной и той же группе разбиения Π;
    /* В худшем случае подгруппа состоит из одного состояния */
    Заменяем G в Π_new множеством образованных подгрупп;
}
```

Рис. 3.64. Построение Π_{new}

4. Выбираем одно из состояний из каждой группы Π_{final} в качестве *представителя* этой группы. Представители будут состояниями ДКА с минимальным количеством состояний D'. Остальные компоненты D' строятся следующим образом.

   а) Начальным состоянием D' является представитель группы, содержащей стартовое состояние D.

   б) Принимающими состояниями D' являются принимающие состояния групп, содержащих принимающие состояния D. Заметим, что каждая группа содержит либо только принимающие состояния, либо только не принимающие, поскольку мы начинаем работу с отделения этих классов состояний друг от друга, а процедура на рис. 3.64 всегда образует новые группы, которые являются подгруппами ранее построенных.

   в) Пусть s — представитель некоторой группы G в Π_{final} и пусть переход для входного символа a в автомате D ведет из состояния s в состояние t. Пусть r — представитель группы H, в которую входит t. Тогда в D' имеется переход из s в r для входного символа a. Заметим, что в D каждое состояние из группы G должно перейти при входном символе a в некоторое состояние из группы H, поскольку иначе группа G должна была бы быть разделена процедурой на рис. 3.64. □

Пример 3.40. Обратимся еще раз к ДКА на рис. 3.36. Начальное разбиение состоит из двух групп, $\{A, B, C, D\}$ $\{E\}$, состоящих соответственно из не принимающих и принимающего состояний. Для построения Π_{new} процедура на рис. 3.64 рассматривает обе группы и входные символы a и b. Группа $\{E\}$ не может быть разделена, поскольку она состоит из одного состояния, так что в Π_{new} она остается неизменной.

Почему работает алгоритм минимизации количества состояний

Нам требуется доказать две вещи: что состояния, остающиеся в одной и той же группе Π_{final}, не отличимы никакой строкой, и что состояния, оказавшиеся в разных группах, отличимы. Первое следует из доказательства по индукции по i того факта, что если после i-й итерации шага 2 алгоритма 3.39 s и t находятся в одной и той же группе, то не существует строки длиной не более i, которая могла бы отличить их друг от друга. Детальное доказательство остается читателю в качестве упражнения.

Второе следует из доказательства по индукции по i того факта, что если состояния s и t помещаются в разные группы на i-й итерации шага 2, то существует строка, отличающая их. Базис индукции доказывается просто: если при начальном разделении s и t принадлежат разным группам, то одно из них является принимающим, а другое нет; как известно, пустая строка ϵ отличает такие состояния друг от друга. Далее, раз состояния s и t помещаются в разные группы, должен существовать символ a и состояния p и q, такие, что s и t при входном символе a переходят соответственно в состояния p и q, причем p и q должны к этой итерации находиться в разных группах. Тогда, в соответствии с гипотезой индукции, существует некоторая строка x, которая отличает p от q. Следовательно, строка ax отличает s от t.

Устранение тупикового состояния

Алгоритм минимизации количества состояний иногда приводит к ДКА с одним тупиковым состоянием — состоянием, не являющимся принимающим и для каждого входного символа переходящим само в себе. Такое состояние технически необходимо, поскольку ДКА по определению должен иметь переходы из каждого состояния для каждого входного символа. Однако, как говорилось в разделе 3.8.3, нам зачастую требуется знать, когда все возможности достичь принимающего состояния исчерпаны, чтобы определить, что корректная лексема уже была найдена. Таким образом, мы можем захотеть устранить тупиковое состояние и использовать автомат, в котором отсутствуют некоторые переходы. Такой автомат имеет на одно состояние меньше, чем ДКА с минимальным количеством состояний, но, строго говоря, он не является детерминированным конечным автоматом из-за отсутствия переходов в тупиковое состояние.

Группа $\{A, B, C, D\}$ может быть разделена, так что мы должны рассмотреть воздействие на нее каждого входного символа. При входном символе a все состояния группы переходят в состояние B, так что отличить эти состояния при помощи строки, начинающейся с a, невозможно. При входном символе b состояния A, B и C переходят в состояния, являющиеся членами группы $\{A, B, C, D\}$, в то время как состояние D переходит в E — член другой группы. Таким образом, в Π_{new} группа $\{A, B, C, D\}$ разбивается на $\{A, B, C\}\,\{D\}$, и Π_{new} на этой итерации представляет собой $\{A, B, C\}\,\{D\}\,\{E\}$.

На следующей итерации мы можем разбить $\{A, B, C\}$ на $\{A, C\}\,\{B\}$, поскольку и A, и C переходят при входном символе b в состояния, являющиеся членами группы $\{A, B, C\}$, в то время как состояние B для этого входного символа переходит в состояние из другой группы — $\{D\}$. Итак, после второй итерации $\Pi_{\text{new}} = \{A, C\}\,\{B\}\,\{D\}\,\{E\}$. На третьей итерации мы не можем разделить единственную оставшуюся группу с более чем одним состоянием, поскольку и A, и C переходят в одно и то же состояние (а следовательно, в одну и ту же группу) при любом входном символе. Таким образом, мы делаем вывод, что $\Pi_{\text{final}} = \{A, C\}\,\{B\}\,\{D\}\,\{E\}$.

Теперь мы должны построить ДКА с минимальным количеством состояний. Он имеет четыре состояния, соответствующие четырем группам Π_{final}, и в качестве представителей мы выбираем A, B, D и E. Начальным является состояние A, а единственным принимающим — состояние E. На рис. 3.65 показана функция переходов для этого ДКА. Например, переход из состояния E для входного символа b ведет в состояние A, поскольку в исходном ДКА E при входном символе b переходило в состояние C, а представителем группы, в которую входит C, является состояние A. По той же причине переход из состояния A для входного символа b ведет вновь в состояние A, в то время как все остальные переходы — такие же, как и на рис. 3.36. □

Состояние	a	b
A	B	A
B	B	D
D	B	E
E	B	A

Рис. 3.65. Таблица переходов ДКА с минимальным количеством состояний

3.9.7 Минимизация состояний в лексических анализаторах

Чтобы применить процедуру минимизации количества состояний к ДКА, сгенерированному в разделе 3.8.3, мы должны начать работу алгоритма 3.39 с разбиения, которое группирует все состояния, распознающие определенный токен, а также разместить в одной группе все состояния, которые не определяют ни один токен. Приведенный ниже пример поясняет сказанное.

Пример 3.41. Для ДКА на рис. 3.54 начальное разбиение имеет вид

$$\{0137, 7\}\,\{247\}\,\{8, 58\}\,\{68\}\,\{\varnothing\}$$

Здесь состояния 0137 и 7 принадлежат одной группе, поскольку они не соответствуют ни одному токену. Состояния 8 и 58 принадлежат другой группе, поскольку оба они указывают на токен **a*b**$^+$. Обратите внимание на то, что к состояниям добавлено тупиковое состояние $\varnothing$, которое имеет переходы для входных символов a и b само в себя. Тупиковое состояние является целевым для отсутствующих переходов для символа a из состояний 8, 58 и 68.

Далее мы должны разделить состояния 0137 и 7, поскольку при входном символе a их переходы ведут в разные группы. Следует также разделить состояния 8 и 58, поскольку их переходы для входного символа b также ведут в разные группы. Таким образом, каждое состояние образует отдельную группу, состоящую из одного этого состояния, и на рис. 3.54 изображен ДКА с минимальным количеством состояний, распознающий три соответствующих токена. Вспомним, что ДКА, служащий в качестве лексического анализатора, обычно попадает в тупиковое состояние, и отсутствие переходов рассматривается как сигнал о том, что обнаружен конец токена. □

3.9.8 Компромисс между скоростью и используемой памятью при моделировании ДКА

Простейший и наиболее быстрый способ представления функции переходов ДКА — двумерная таблица, проиндексированная состояниями и символами. Для данного состояния и очередного входного символа мы находим в таблице следующее состояние, а также специальные действия, которые должны быть предприняты, например возврат токена синтаксическому анализатору. Поскольку обычно лексический анализатор имеет ДКА с несколькими сотнями состояний и использует ASCII-алфавит из 128 символов, массив требует менее мегабайта памяти.

Однако компиляторы встречаются и в очень малых устройствах, для которых и мегабайт памяти — это слишком много. Для таких случаев имеется множество методов, которые могут использоваться для сжатия таблицы переходов. Например, каждое состояние может быть представлено списком переходов, т.е. парами

"состояние — символ", завершающимся состоянием по умолчанию, которое выбирается в случае, если входного символа нет в списке. Если выбрать в качестве состояния по умолчанию наиболее часто встречающееся следующее состояние, в большинстве случаев удается уменьшить количество необходимой памяти во много раз.

Имеется и более подходящая структура данных, которая позволяет объединить скорость доступа к массиву с экономным расходом памяти списками с состояниями по умолчанию. Эту структуру можно рассматривать как четыре массива, как показано на рис. 3.66.[8] Массив *base* используется для определения базового расположения записей для состояния s, хранящихся в массивах *next* и *check*. Массив *default* применяется для определения альтернативной базовой позиции, если массив *check* говорит нам, что данное значение *base* $[s]$ некорректно.

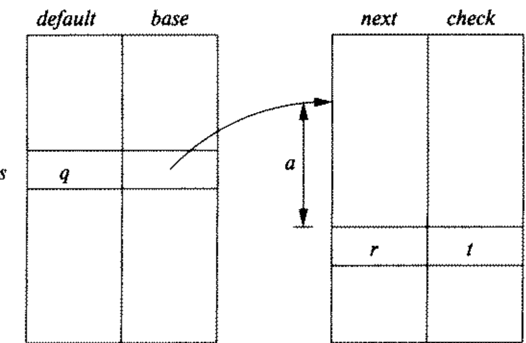


Рис. 3.66. Структура данных для представления таблиц переходов

Для вычисления перехода из состояния s для входного символа a *nextState* (s, a) мы проверяем значения записей *next* и *check* в позиции $l =$ *base* $[s] + a$, где символ a рассматривается как целое число, вероятно, из диапазона от 0 до 127. Если *check* $[l] = s$, то эта запись корректна, и следующим для состояния s и входного символа a является состояние *next* $[l]$. Если же *check* $[l] \neq s$, то мы находим другое состояние $t =$ *default* $[s]$ и повторяем процесс так, как если бы t было текущим состоянием. Более формально функция *nextState* определяется следующим образом:

```
int nextState (s, a) {
    if ( check [base [s] + a] == s ) return next [base [s] + a];
    else return nextState (default [s] , a);
}
```

[8]На практике может использоваться еще один массив, проиндексированный состояниями и содержащий связанные с состояниями действия, если таковые имеются.

Использование структуры, показанной на рис. 3.66, призвано сократить массивы *next* и *check* путем использования подобия состояний. Например, состояние t, являющееся состоянием по умолчанию для s, может быть состоянием, которое говорит "мы работаем с идентификатором", как, например, состояние 10 на рис. 3.14. Возможно, в состояние s мы попали после входной строки `th`, которая может быть как префиксом ключевого слова `then`, так и потенциально префиксом некоторой лексемы идентификатора. Если очередной входной символ — `e`, мы должны перейти из состояния s в специальное состояние, запоминающее, что мы считали входную строку `the`; в противном случае состояние s ведет себя так же, как и состояние t. Таким образом, мы устанавливаем $check\,[base\,[s] + e]$ равным s (чтобы подтвердить, что это корректная запись для s), а $next\,[base\,[s] + e]$ равным состоянию, запоминающему строку `the`. Кроме того, $default\,[s]$ устанавливается равным t.

Хотя мы можем оказаться не способными выбрать значения *base* так, чтобы не оставалось неиспользованных записей *next*/*check*, опыт показывает, что простейшая стратегия поочередного присваивания значений *base* состояниям и присваивания каждому $base\,[s]$ минимального целого значения, такого, что специальные записи могут заполняться без конфликтов с существующими, достаточно хороша. Более того, она использует объем памяти, лишь ненамного превышающий минимально возможный.

3.9.9 Упражнения к разделу 3.9

Упражнение 3.9.1. Расширьте таблицу на рис. 3.58 так, чтобы она включала операторы ? и $^+$.

Упражнение 3.9.2. Воспользуйтесь алгоритмом 3.36 для преобразования регулярных выражений из упражнения 3.7.3 непосредственно в детерминированные конечные автоматы.

! **Упражнение 3.9.3.** Можно доказать, что два регулярных выражения эквивалентны, показав, что их ДКА с минимальным количеством состояний одинаковы с точностью до имен состояний. Покажите таким способом, что регулярные выражения $(\mathbf{a} \mid \mathbf{b})^*$, $(\mathbf{a}^* \mid \mathbf{b}^*)^*$ и $((\epsilon \mid \mathbf{a})\,\mathbf{b}^*)^*$ эквивалентны. *Примечание*: ДКА для этих выражений вы должны были получить при решении упражнения 3.7.3.

! **Упражнение 3.9.4.** Постройте ДКА с минимальным количеством состояний для следующих регулярных выражений:

а) $(\mathbf{a} \mid \mathbf{b})^*\,\mathbf{a}\,(\mathbf{a} \mid \mathbf{b})$;

б) $(\mathbf{a} \mid \mathbf{b})^*\,\mathbf{a}\,(\mathbf{a} \mid \mathbf{b})\,(\mathbf{a} \mid \mathbf{b})$;

в) $(\mathbf{a} \mid \mathbf{b})^*\,\mathbf{a}\,(\mathbf{a} \mid \mathbf{b})\,(\mathbf{a} \mid \mathbf{b})\,(\mathbf{a} \mid \mathbf{b})$.

Вы не видите в построенных деревьях некоторого шаблона?

!! **Упражнение 3.9.5.** Чтобы формально обосновать неформальное утверждение из упражнения 3.25, покажите, что любой детерминированный конечный автомат для регулярного выражения

$$(\mathbf{a} \mid \mathbf{b})^* \, \mathbf{a} \, (\mathbf{a} \mid \mathbf{b}) \, (\mathbf{a} \mid \mathbf{b}) \cdots (\mathbf{a} \mid \mathbf{b}),$$

где $(\mathbf{a} \mid \mathbf{b})$ встречается в конце $n - 1$ раз, должен иметь как минимум 2^n состояний. *Указание*: рассмотрите шаблон из упражнения 3.9.4. Какое условие, связанное с историей поступления входных символов, представляет каждое состояние?

3.10 Резюме к главе 3

- *Токены.* Лексический анализатор сканирует исходную программу и выдает последовательность токенов, которая обычно передается синтаксическому анализатору по одному токену. Некоторые токены состоят из одного имени токена, в то время как другие могут иметь связанное с ними лексическое значение, несущее информацию о конкретном экземпляре токена, найденном во входном потоке.

- *Лексемы.* Каждый раз, когда лексический анализатор возвращает синтаксическому анализатору токен, с последним связана его лексема — последовательность входных символов, которую представляет токен.

- *Буферизация.* Поскольку зачастую для того, чтобы увидеть, где заканчивается очередная лексема, требуется опережающее сканирование входного потока (предпросмотр), лексический анализатор должен буферизовать входной поток. Две методики, ускоряющие процесс сканирования входного потока — циклическое использование пары буферов с применением ограничителей, которые предупреждают о достижении конца буфера.

- *Шаблоны.* Каждый токен имеет шаблон, который описывает, какая последовательность символов может формировать лексему, соответствующую этому токену. Множество слов, или строк символов, которые соответствуют данному шаблону, называется языком.

- *Регулярные выражения.* Эти выражения обычно используются для описания шаблонов. Регулярные выражения строятся из отдельных символов с использованием операторов объединения, конкатенации и замыкания Клини.

- *Регулярные определения.* Для определения сложных наборов языков, таких как шаблоны, описывающие токены языка программирования, часто используются регулярные определения, которые представляют собой последовательности инструкций, в которых регулярным выражениям сопоставляются обозначающие их переменные. Регулярное выражение для некоторой переменной может использовать ранее определенные переменные для других регулярных выражений.

- *Расширенная запись регулярных выражений.* В регулярных выражениях для упрощения может использоваться ряд дополнительных операторов, представляющих собой сокращения имеющихся регулярных выражений. Примерами таких операторов могут служить оператор + (одно или несколько вхождений), ? (нуль или одно вхождение), а также классы символов.

- *Диаграммы переходов.* Поведение лексического анализатора часто можно описать с использованием диаграмм переходов. Эти диаграммы включают состояния, каждое из которых представляет определенную историю просмотренных входных символов в процессе поиска текущей лексемы, соответствующей одному из возможных шаблонов. Из одних состояний в другие ведут стре́лки, или переходы, каждая из которых соответствует некоторому возможному входному символу, приводящему к изменению состояния лексического анализатора.

- *Конечные автоматы.* Существует формализация диаграмм переходов, которая включает указание одного начального и одного или нескольких принимающих состояний, а также множества состояний, входных символов и переходов между состояниями. Принимающие состояния указывают, что найдена лексема для некоторого токена. В отличие от диаграмм переходов конечные автоматы могут осуществлять переходы как для входных символов, так и для пустой входной строки.

- *Детерминированные конечные автоматы.* ДКА представляет собой специальный вид конечного автомата, у которого из каждого состояния для каждого входного символа имеется ровно один переход. Кроме того, запрещены переходы для пустых строк. ДКА легко моделируются и обеспечивают хорошую реализацию лексических анализаторов, аналогичную диаграммам переходов.

- *Недетерминированные конечные автоматы.* Автомат, не являющийся детерминированным, называется недетерминированным. НКА часто существенно проще спроектировать, чем ДКА. Еще одной возможной архитектурой лексического анализатора является табуляция для каждого возмож-

ного шаблона всех состояний НКА, в которых он может оказаться при сканировании входного потока.

- *Преобразования представлений шаблонов.* Можно преобразовать любое регулярное выражение в НКА примерно того же размера, распознающий определяемый этим регулярным выражением язык. Далее, любой НКА может быть преобразован в ДКА для того же шаблона, хотя в наихудшем случае (не встречающемся при работе с обычными языками программирования) размер ДКА может экспоненциально зависеть от размера НКА. Можно также преобразовать любой недетерминированный или детерминированный конечный автомат в регулярное выражение, которое определяет тот же язык, что и распознаваемый этим автоматом.

- *Lex.* Существует семейство генераторов лексических анализаторов, включающее `Lex` и `Flex`. Пользователь такого генератора определяет шаблоны для токенов с применением расширенной записи регулярных выражений. `Lex` преобразует эти выражения в лексический анализатор, который, по сути, представляет собой детерминированный конечный автомат, распознающий каждый из шаблонов.

- *Минимизация конечного автомата.* Для каждого ДКА существует ДКА с минимальным количеством состояний, принимающий тот же язык. Более того, ДКА с минимальным количеством состояний для данного языка является единственным с точностью до имен состояний автомата.

3.11 Список литературы к главе 3

Впервые регулярные выражения были разработаны Клини (Kleene) в 1950-х годах [9]. Клини интересовало описание событий, которые могли быть представлены конечно-автоматной моделью нервной деятельности Мак-Каллока (McCullough) и Питтса (Pitts) [12]. С того времени регулярные выражения и конечные автоматы стали широко использоваться в информатике.

Регулярные выражения в различных формах используются во многих популярных утилитах Unix, таких как `awk`, `ed`, `egrep`, `grep`, `lex`, `sed`, `sh` и `vi`. Стандарты IEEE 1003 и ISO/IEC 9945 системного интерфейса переносимых операционных систем (Portable Operating System Interface — POSIX) определяют расширенные регулярные выражения POSIX, аналогичные оригинальным регулярным выражениям Unix с небольшими исключениями, такими как мнемонические представления классов символов. Многие языки сценариев, такие как Perl, Python и Tcl, используют регулярные выражения, хотя зачастую и с нестандартными расширениями.

Привычная модель конечного автомата, как и его минимизация (алгоритм 3.39), появилась благодаря работам Хаффмана (Huffman) [6] и Мура (Moore) [14]. Недетерминированные конечные автоматы были предложены Рабином (Rabin) и Скоттом (Scott) [15]; они же предложили метод построения подмножеств (алгоритм 3.20) и показали эквивалентность детерминированных и недетерминированных конечных автоматов.

Мак-Нотон (McNaughton) и Ямада (Yamada) [13] описали алгоритм преобразования регулярных выражений непосредственно в детерминированные конечные автоматы. Алгоритм 3.36, описанный в разделе 3.9, был впервые использован Ахо (Aho) при создании утилиты `egrep` в Unix. Этот алгоритм применялся также и в подпрограммах проверки соответствия строк регулярным выражениям в утилите `awk` [3]. Подход с использованием промежуточного недетерминированного конечного автомата предложен Томпсоном (Thompson) [17]. Эта же статья содержит и алгоритм непосредственного моделирования недетерминированного конечного автомата (алгоритм 3.22), использованный Томпсоном в текстовом редакторе `QED`.

Первая версия `Lex` была разработана Леском (Lesk), после чего во второй версии `Lex` Леска и Шмидта (Schmidt) был применен алгоритм 3.36 [10]. Впоследствии было реализовано множество различных вариантов `Lex`. GNU-версию, `Flex` вместе с документацией можно найти на Web-сайте [4]. Популярные версии `Lex` для Java включают `JFlex` [7] и `JLex` [8].

Алгоритм КМП, описанный в разделе 3.4 перед упражнением 3.4.3, разработан в [11]. Его обобщение для многих ключевых слов описано в [2] и использовано Ахо в его первой реализации утилиты Unix `fgrep`.

Теория конечных автоматов и регулярных выражений излагается в [5], а хороший обзор методов поиска подстрок можно найти в [1].

1. Aho, A. V., "Algorithms for finding patterns in strings", in *Handbook of Theoretical Computer Science* (J. van Leeuwen, ed.), Vol. A, Ch. 5, MIT Press, Cambridge, 1990.

2. Aho, A. V. and M. J. Corasick, "Efficient string matching: an aid to bibliographic search", *Comm. ACM* **18**:6 (1975), pp. 333–340.

3. Aho, A. V., B. W. Kernighan, and P. J. Weinberger, *The AWK Programming Language*, Addison-Wesley, Boston, MA, 1988.

4. Начальная страница Flex `http://www.gnu.org/software/flex/`, Free Software Foundation.

5. Hopcroft, J. E., R. Motwani, and J. D. Ullman, *Introduction to Automata Theory, Languages, and Computation*, Addison-Wesley, Boston MA, 2006.

6. Huffman, D. A., "The synthesis of sequential machines", *J. Franklin Inst.* **257** (1954), pp. 3–4, 161, 190, 275–303.

7. Начальная страница JFlex `http://jflex.de/`.

8. `http://www.cs.princeton.edu/~appel/modern/java/JLex`.

9. Kleene, S. C., "Representation of events in nerve nets", in [16], pp. 3–40.

10. Lesk, M. E., "Lex — a lexical analyzer generator", Computing Science Tech. Report 39, Bell Laboratories, Murray Hill, NJ, 1975. Аналогичный документ с тем же названием , но с участием E. Schmidt в качестве соавтора, имеется в Vol. 2 of the *Unix Programmer's Manual*, Bell laboratories, Murray Hill NJ, 1975; см. `http://dinosaur.compilertools.net/lex/index.html`.

11. Knuth, D. E., J. H. Morris, and V. R. Pratt, "Fast pattern matching in strings", *SIAM J. Computing* **6**:2 (1977), pp. 323–350.

12. McCullough, W. S. and W. Pitts, "A logical calculus of the ideas immanent in nervous activity", *Bull. Math. Biophysics* **5** (1943), pp. 115–133.

13. McNaughton, R. and H. Yamada, "Regular expressions and state graphs for automata", *IRE Trans. on Electronic Computers* **EC-9**:1 (1960), pp. 38–47.

14. Moore, E. F., "Gedanken experiments on sequential machines", in [16], pp. 129–153.

15. Rabin, M. O. and D. Scott, "Finite automata and their decision problems", *IBM J. Res. and Devel.* **3**:2 (1959), pp. 114–125.

16. Shannon, C. and J. McCarthy (eds.), *Automata Studies*, Princeton Univ. Press, 1956.

17. Thompson, K., "Regular expression search algorithm", *Comm. ACM* **11**:6 (1968), pp. 419–422.

Глава 4

Синтаксический анализ

Эта глава посвящена методам синтаксического анализа, обычно используемым в компиляторах. Сначала здесь будут представлены базовые концепции, затем — методы, подходящие для реализации вручную, и наконец — алгоритмы, используемые в автоматизированном инструментарии. Поскольку программы могут содержать синтаксические ошибки, здесь же будут рассмотрены и методы синтаксического анализа для восстановления после распространенных ошибок.

В соответствии с дизайном каждый язык программирования имеет точные правила, которые предписывают синтаксическую структуру корректных программ. В C, например, программа создается из функций, функция — из объявлений и инструкций, инструкции — из выражений и т.д. Синтаксис конструкций языка программирования может быть описан с помощью контекстно-свободных грамматик или записи БНФ (Backus–Naur Form — форма Бэкуса–Наура), о которой шла речь в разделе 2.2. Грамматики обеспечивают значительные преимущества разработчикам языков программирования и создателям компиляторов.

- Грамматика дает точную и при этом простую для понимания синтаксическую спецификацию языка программирования.

- Для некоторых классов грамматик мы можем автоматически построить эффективный синтаксический анализатор, который определяет синтаксическую структуру исходной программы. Дополнительным преимуществом автоматического создания анализатора является возможность обнаружения синтаксических неоднозначностей и других сложностей, которые иначе могли бы остаться незамеченными на начальных фазах создания языка и его компилятора.

- Правильно построенная грамматика придает языку программирования структуру, которая способствует облегчению трансляции исходной программы в корректный объектный код и выявлению ошибок.

- Грамматика позволяет языку итеративно эволюционировать, обогащаясь новыми конструкциями для решения новых задач. Добавление этих новых конструкций в язык оказывается более простой задачей, если существующая реализация языка основана на его грамматическом описании.

4.1 Введение

В этом разделе будет рассмотрено место синтаксического анализатора в типичном компиляторе. После этого мы обратимся к обычной грамматике для арифметических выражений, которой вполне достаточно для иллюстрации сути синтаксического анализа, поскольку методы синтаксического анализа выражений переносятся на большинство программных конструкций. Заканчивается этот раздел рассмотрением обработки ошибок, поскольку синтаксический анализатор должен с честью выходить из ситуаций, когда его входные данные не могут быть сгенерированы соответствующей грамматикой.

4.1.1 Роль синтаксического анализатора

В нашей модели компилятора синтаксический анализатор получает строку токенов от лексического анализатора, как показано на рис. 4.1, и проверяет, может ли эта строка имен токенов порождаться грамматикой исходного языка. Мы также ожидаем от синтаксического анализатора сообщений обо всех выявленных ошибках, причем достаточно внятных и полных, а кроме того, умения обрабатывать обычные, часто встречающиеся ошибки и продолжать работу с оставшейся частью программы. Концептуально в случае корректной программы синтаксический анализатор строит дерево разбора и передает его следующей части компилятора для дальнейшей обработки. В действительности явное построение дерева разбора не требуется, поскольку проверки и действия трансляции, как мы уже видели, могут выполняться в процессе синтаксического анализа. Таким образом, синтаксический анализатор и прочие части начальной стадии компилятора могут быть реализованы в виде единого модуля.

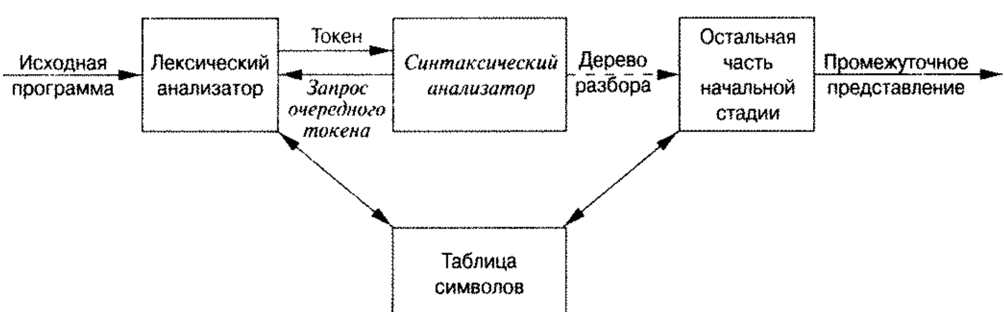


Рис. 4.1. Место синтаксического анализатора в модели компилятора

Имеется три основных типа синтаксических анализаторов грамматик: универсальные, восходящие и нисходящие. Универсальные методы разбора, такие как алгоритмы Кока–Янгера–Касами (Cocke–Younger–Kasami) и Эрли (Earley), могут работать с любой грамматикой (см. список литературы к главе 4). Однако эти

обобщенные методы слишком неэффективны для использования в промышленных компиляторах.

Методы, обычно применяемые в компиляторах, можно классифицировать как нисходящие (сверху вниз — top-down) или восходящие (снизу вверх — bottom-up). Как явствует из названий, нисходящие синтаксические анализаторы строят дерево разбора сверху (от корня) вниз (к листьям), тогда как восходящие начинают с листьев и идут к корню. В обоих случаях входной поток синтаксического анализатора сканируется посимвольно слева направо.

Наиболее эффективные нисходящие и восходящие методы работают только с подклассами грамматик, однако некоторые из этих классов, такие как LL- и LR-грамматики, достаточно выразительны для описания большинства синтаксических конструкций языков программирования. Реализованные вручную синтаксические анализаторы чаще работают с LL-грамматиками; например, предиктивный подход, описанный в разделе 2.4.2, работает с LL-грамматиками. Синтаксические анализаторы для большего класса LR-грамматик обычно создаются с помощью автоматизированных инструментов.

В этой главе мы полагаем, что выход синтаксического анализатора является некоторым представлением дерева разбора потока токенов от лексического анализатора. На практике имеется множество задач, которые могут сопровождать процесс разбора, такие как сбор информации о различных токенах в таблицу символов, выполнение проверки типов и других видов семантического анализа, а также генерация промежуточного кода. Все эти задачи представлены одним блоком "Остальная часть начальной стадии" на рис. 4.1. Детальнее они будут рассмотрены в последующих главах.

4.1.2 Образцы грамматик

Ниже представлены некоторые из грамматик, рассматриваемых в этой главе, что должно упростить дальнейшие ссылки на них. Конструкции, начинающиеся с ключевых слов наподобие **begin** и **int**, проанализировать относительно просто, поскольку ключевое слово определяет выбор продукции грамматики, которая должна быть применена ко входному потоку. Поэтому мы сконцентрируемся на выражениях, работать с которыми сложнее из-за ассоциативности и приоритета операторов.

Ассоциативность и приоритет операторов присутствуют в следующей грамматике, аналогичной грамматикам из главы 2, использовавшимся для описания выражений, слагаемых и сомножителей. E представляет выражение, состоящее из слагаемых, разделенных знаками $+$, T представляет слагаемые, которые со-

стоят из сомножителей, разделенных знаками *, а F представляет сомножители, которые могут быть либо выражениями в скобках, либо идентификаторами:

$$\begin{array}{lcl} E & \to & E + T \mid T \\ T & \to & T * F \mid F \\ F & \to & (E) \mid \mathbf{id} \end{array} \tag{4.1}$$

Грамматика выражений (4.1) принадлежит классу LR-грамматик, которые подходят для восходящего синтаксического анализа. Эта грамматика может быть адаптирована для работы с дополнительными операторами и дополнительными уровнями приоритетов. Однако она не может быть использована для нисходящего синтаксического анализа в силу ее леворекурсивности.

Для нисходящего синтаксического анализа можно использовать следующий вариант грамматики (4.1), в котором устранена левая рекурсия:

$$\begin{array}{lcl} E & \to & T\,E' \\ E' & \to & +T\,E' \mid \epsilon \\ T & \to & F\,T' \\ T' & \to & *F\,T' \mid \epsilon \\ F & \to & (E) \mid \mathbf{id} \end{array} \tag{4.2}$$

Приведенная далее грамматика рассматривает * и + как идентичные, так что она пригодится для иллюстрации методов обработки неоднозначностей в процессе синтаксического анализа:

$$E \;\to\; E + E \mid E * E \mid (E) \mid \mathbf{id} \tag{4.3}$$

Здесь E представляет выражения всех типов. Грамматика (4.3) допускает более чем одно дерево разбора для выражений наподобие $a + b * c$.

4.1.3 Обработка синтаксических ошибок

В оставшейся части этого раздела мы рассмотрим природу синтаксических ошибок и общие стратегии восстановления после ошибок. Две из стратегий, называющиеся "в режиме паники" и "на уровне фразы", будут рассмотрены более детально в связи с конкретными методами синтаксического анализа.

Если компилятор будет иметь дело исключительно с корректными программами, его разработка и реализация существенно упрощаются. Однако от компилятора ожидается, что он будет помогать программисту обнаруживать и устранять ошибки, которые неизбежно содержатся в программах несмотря на огромные усилия даже самых квалифицированных программистов. Примечательно, что хотя ошибки — явление чрезвычайно распространенное, лишь в нескольких языках

вопрос обработки ошибок рассматривался еще на фазе проектирования языка. Наша цивилизация была бы совсем другой, если бы в естественных языках были такие же требования к синтаксической точности, как и в языках программирования. Большинство спецификаций языков программирования, тем не менее, не определяет реакции компилятора на ошибки — этот вопрос отдается на откуп разработчикам компилятора. Однако планирование системы обработки ошибок с самого начала работы над компилятором может как упростить его структуру, так и улучшить его реакцию на ошибки.

Ошибки в программе могут быть на самых разных уровнях.

- *Лексические* ошибки включают неверно записанные идентификаторы, ключевые слова или операторы, например использование идентификатора `elipseSize` вместо `ellipseSize` или отсутствие кавычек вокруг текста, являющегося строкой.

- *Синтаксические* ошибки включают неверно поставленные точки с запятой или лишние или недостающие фигурные скобки `{` или `}`. В качестве еще одного примера в C или Java синтаксической ошибкой является конструкция `case` без охватывающего `switch` (однако эта ситуация часто пропускается синтаксическим анализатором и перехватывается позже, на уровне генерации кода).

- *Семантические* ошибки включают несоответствие типов операторов и их операндов. В качестве примера можно привести оператор `return` со значением в методе Java, возвращающем тип `void`.

- *Логические* ошибки могут быть любыми — от неверных решений программиста до использования в программе на языке C оператора присваивания = вместо оператора сравнения ==. Программа, содержащая оператор =, может быть корректной, но делать совсем не то, чего хотел от нее программист.

Точность современных методов разбора позволяет очень эффективно выявлять синтаксические ошибки в программе. Некоторые методы синтаксического анализа, такие как LL и LR, обнаруживают ошибки на самых ранних стадиях, т.е. когда разбор потока токенов от лексического анализатора в соответствии с грамматикой языка становится невозможен. Говоря более точно, они обладают *свойством корректного префикса* (viable-prefix property), т.е. они обнаруживают ошибку, как только встречают префикс, который не может быть префиксом ни одной корректной строки данного языка.

Еще одна причина, по которой делается упор на восстановление после ошибки в процессе синтаксического анализа, — многие ошибки, чем бы они ни были вызваны, оказываются синтаксическими и выявляются при дальнейшей невозможности синтаксического анализа. Эффективно обнаруживаются и некоторые

семантические ошибки, такие как несоответствие типов. Однако в общем случае точное определение семантических и логических ошибок во время компиляции является крайне трудной задачей.

Обработчик ошибок синтаксического анализатора имеет очень просто формулируемые, но очень сложно реализуемые цели:

- он должен ясно и точно сообщать о наличии ошибок;
- он должен обеспечивать быстрое восстановление после ошибки, чтобы продолжить поиск других ошибок;
- он не должен существенно замедлять обработку корректной программы.

К счастью, обычные ошибки достаточно просты, и для их обработки часто достаточно относительно простых механизмов обработки ошибок.

Каким образом обработчик ошибок должен сообщить об обнаруженных неприятностях? Самое меньшее, что он должен сделать, — указать место в исходной программе, где выявлена ошибка, поскольку, скорее всего, реальная ошибка находится несколькими токенами ранее. Распространенная стратегия заключается в том, чтобы вывести проблемную строку с указателем позиции, в которой обнаружена ошибка.

4.1.4 Стратегии восстановления после ошибок

После того как ошибка обнаружена, каким образом должно выполниться восстановление синтаксического анализатора? Хотя универсальной стратегии не существует, все же несколько методов применяются чаще других. Простейший подход состоит в том, что синтаксический анализатор завершает работу при первой же обнаруженной ошибке, выводя о ней информативное сообщение. Если же синтаксический анализатор может восстановить некоторым образом свое состояние до такого, когда работа может быть продолжена, то имеется определенная надежда на то, что будут обнаружены и другие ошибки, а также что информация о них окажется достаточно корректной. Если же восстановление не совсем корректное и количество обнаруженных ошибок растет, как снежный ком, то будет лучше, если компилятор после некоторого предельного количества ошибок прекратит вывод раздражающих сообщений об этих "фальшивых" ошибках.

Этот раздел посвящен следующим стратегиям восстановления: режим паники, уровень фразы, продукции ошибок и глобальная коррекция.

Восстановление в режиме паники

В этом случае при обнаружении ошибки синтаксический анализатор пропускает входные символы по одному, пока не будет найден один из специально

определенного множества *синхронизирующих токенов*. Обычно такими токенами являются разделители, такие как ; или }, роль которых в исходной программе совершенно очевидна и недвусмысленна. Разработчик компилятора, естественно, должен выбирать синхронизирующие токены, исходя из особенностей исходного языка. Хотя в ходе такой коррекции часто значительное количество символов пропускается без проверки на наличие дополнительных ошибок, ее преимуществом является простота; кроме того, в отличие от некоторых методов, рассматриваемых ниже, она гарантирует отсутствие зацикливания.

Восстановление на уровне фразы

При обнаружении ошибки синтаксический анализатор может выполнить локальную коррекцию оставшегося входного потока, т.е. он может заменить префикс остальной части потока некоторой строкой, которая позволит синтаксическому анализатору продолжить работу. Типичная локальная коррекция может состоять в замене запятой точкой с запятой, удалении лишней точки с запятой или вставке недостающей. Выбор способа локальной коррекции — прерогатива разработчика компилятора. Естественно, следует быть предельно осторожным при выборе способа коррекции, чтобы он не привел, например, к бесконечному циклу (что возможно при вставке некоторого символа перед текущим сканируемым).

Замена на уровне фразы используется в некоторых компиляторах с восстановлением после ошибки и может скорректировать любую входную строку. Главным недостатком этого метода являются сложности, с которыми приходится сталкиваться в ситуациях, когда реальная ошибка произошла до точки ее обнаружения.

Продукции ошибок

Знание наиболее распространенных ошибок позволяет расширить грамматику языка продукциями, порождающими ошибочные конструкции. Синтаксический анализатор, построенный на основе такой расширенной продукциями ошибок грамматики языка, обнаруживает ошибку при использовании "ошибочной" продукции. Таким образом, он может сгенерировать корректное диагностическое сообщение для распознанной во входном потоке ошибки.

Глобальная коррекция

В идеальном случае при обработке некорректной входной строки компилятор должен вносить минимально возможное количество изменений. Существует ряд алгоритмов, которые позволяют выбрать минимальную последовательность вносимых изменений для проведения коррекции. По заданной некорректной строке x и грамматике G такие алгоритмы находят дерево разбора для строки y, такой, что количество вставок, удалений и изменений токенов, требуемых для преобразования x в y, минимально возможное. К сожалению эти методы в общем случае слишком дорогостоящи для применения в компиляторах в силу высоких требо-

ваний к памяти и времени работы, а потому представляют в настоящее время, в основном, теоретический интерес.

Следует заметить, что ближайшая к исходной исправленная программа может оказаться вовсе не тем, что хотел получить программист. Тем не менее понятие минимальной коррекции дает нам критерий оценки качества различных технологий восстановления после ошибок и используется для поиска оптимальной замены строки при восстановлении на уровне фразы.

4.2 Контекстно-свободные грамматики

Грамматики были введены в разделе 2.2 для систематического описания конструкций языка программирования наподобие выражений или инструкций. Используя синтаксическую переменную *stmt* для обозначения инструкций и переменную *expr* для обозначения выражений, продукция

$$stmt \rightarrow \textbf{if}\ (expr)\ stmt\ \textbf{else}\ stmt \tag{4.4}$$

определяет структуру условной инструкции данного вида. Другие продукции затем точно определяют, что могут представлять собой *expr* и *stmt*.

В этом разделе рассматривается определение контекстно-свободных грамматик и вводится терминология, используемая при рассмотрении синтаксического анализа. В частности, понятие порождения очень полезно при рассмотрении порядка применения продукций в процессе разбора.

4.2.1 Формальное определение контекстно-свободной грамматики

Из раздела 2.2 мы знаем, что контекстно-свободная грамматика (далее для краткости — просто грамматика) состоит из терминалов, нетерминалов, стартового символа и продукций.

1. *Терминалы* представляют собой базовые символы, из которых формируются строки. Термин "имя токена" является синонимом слова "терминал", и мы часто будем использовать слово "токен" для обозначения терминала там, где очевидно, что мы говорим об имени токена. Мы считаем также, что терминалы являются первыми компонентами токенов, возвращаемых лексическим анализатором, когда мы говорим о грамматиках языков программирования. В (4.4) терминалами являются ключевые слова **if** и **else**, а также символы (и).

2. *Нетерминалы* представляют собой синтаксические переменные, которые обозначают множества строк. В (4.4) *stmt* и *expr* являются нетерминалами.

Эти множества строк, обозначаемые нетерминалами, помогают определить язык, порождаемый грамматикой. Нетерминалы также налагают на язык иерархическую структуру, облегчающую синтаксический анализ и трансляцию.

3. Один из нетерминалов грамматики считается *стартовым символом*, и множество строк, которые он обозначает, является языком, определяемым грамматикой. По соглашению продукции стартового символа указываются первыми.

4. Продукции грамматики определяют способ, которым терминалы и нетерминалы могут объединяться для создания строк. Каждая *продукция* состоит из следующих частей.

   а) Нетерминал, именуемый *заголовком* или *левой частью* продукции; эта продукция определяет некоторые из строк, обозначаемые заголовком.

   б) Символ $\rightarrow$. Иногда вместо стрелки используется ::=.

   в) *Тело*, или *правая часть*, состоящая из нуля или некоторого количества терминалов и нетерминалов. Эти компоненты тела описывают один из способов, которым могут быть построены строки нетерминала в заголовке.

Пример 4.1. Грамматика на рис. 4.2 определяет простые арифметические выражения. В этой грамматике терминальными символами являются

$$\mathbf{id} \; + \; - \; * \; / \; (\;)$$

Нетерминальными символами являются *expression*, *term* и *factor*, причем *expression* — стартовый символ грамматики. □

$$\begin{array}{rcl}
expression & \rightarrow & expression + term \\
expression & \rightarrow & expression - term \\
expression & \rightarrow & term \\
term & \rightarrow & term * factor \\
term & \rightarrow & term \; / \; factor \\
term & \rightarrow & factor \\
factor & \rightarrow & (\; expression \;) \\
factor & \rightarrow & \mathbf{id}
\end{array}$$

Рис. 4.2. Грамматика для простых арифметических выражений

4.2.2 Соглашения об обозначениях

Для того чтобы избежать постоянного упоминания "это — терминалы", "это — нетерминалы" и т.п., будем использовать следующие соглашения по обозначениям при записи грамматик в оставшейся части книги.

1. Следующие символы являются терминалами.

   а) Строчные буквы из начала алфавита, такие как a, b, c.

   б) Символы операторов, такие как +, * и т.п.

   в) Символы пунктуации, такие как запятые, скобки и т.п.

   г) Цифры `0`, `1`, ..., `9`.

   д) Строки, выделенные полужирным шрифтом, такие как **id** и **if**, каждая из которых представляет терминальный символ.

2. Следующие символы являются нетерминалами.

   а) Прописные буквы из начала алфавита, такие как A, B, C.

   б) Буква S, которая обычно означает стартовый символ.

   в) Имена из строчных букв, выделенные курсивом, такие как *stmt* и *expr*.

   г) При рассмотрении программных конструкций прописные буквы могут использоваться для представления нетерминалов конструкций. Например, нетерминалы для выражений, слагаемых и сомножителей часто представлены буквами E, T и F соответственно.

3. Прописные буквы из конца алфавита, такие как X, Y, Z, представляют *грамматические символы*, т.е. либо терминалы, либо нетерминалы.

4. Строчные буквы из конца алфавита, такие как $u, v, \dots, z$, обозначают строки терминалов.

5. Строчные греческие буквы, такие как α, β, γ, представляют (возможно, пустые) строки грамматических символов. Таким образом, в общем виде продукция может быть записана как $A \to \alpha$, где A — заголовок продукции, а α — ее тело.

6. Множество продукций $A \to \alpha_1$, $A \to \alpha_2$, ..., $A \to \alpha_k$ с общим заголовком A (называющихся A-продукциями) может быть записано как $A \to \alpha_1 \mid \alpha_2 \mid \cdots \mid \alpha_k$. $\alpha_1, \alpha_2, \dots, \alpha_k$ называются *альтернативами* (alternatives) A.

7. Если иное не указано явно, заголовок первой продукции является стартовым символом.

Пример 4.2. Используя указанные соглашения, грамматика из примера 4.1 может быть сокращенно записана как

$$\begin{array}{lcl} E & \rightarrow & E + T \mid E - T \mid T \\ T & \rightarrow & T * F \mid T / F \mid F \\ F & \rightarrow & (E) \mid \mathbf{id} \end{array}$$

Соглашения по обозначениям говорят нам, что E, T и F — нетерминалы, причем E является стартовым символом. Остальные символы являются терминалами. □

4.2.3 Порождения

Построение дерева разбора можно сделать совершенно точным при помощи порождений, рассматривая каждую продукцию как правило для переписывания. Начиная со стартового символа, на каждом шаге переписывания нетерминал замещается телом одной из его продукций. Описанные действия соответствуют нисходящему построению дерева разбора, но точность, достигаемая таким образом, оказывается полезной при рассмотрении восходящего синтаксического анализа. Как мы увидим, восходящий синтаксический анализ связан с классом порождений, известных как правые порождения, когда на каждом шаге переписывается крайний справа нетерминал.

Рассмотрим, например, следующую грамматику с единственным нетерминалом E, в которой к грамматике (4.3) добавлена продукция $E \rightarrow -E$:

$$E \rightarrow E + E \mid E * E \mid -E \mid (E) \mid \mathbf{id} \quad (4.5)$$

Продукция $E \rightarrow -E$ означает, что если E обозначает выражение, то $-E$ также должно обозначать выражение. Замена одного E на $-E$ может быть описана записью

$$E \Rightarrow -E$$

Она читается как "E порождает $-E$". Продукция $E \rightarrow -E$ может быть применена для замены любого экземпляра E в любой строке символов грамматики на (E), например $E * E \Rightarrow (E) * E$ или $E * E \Rightarrow E * (E)$. Можно взять один нетерминал E и многократно применять продукции в произвольном порядке для получения последовательности замещений, например

$$E \Rightarrow -E \Rightarrow -(E) \rightarrow -(\mathbf{id})$$

Такая последовательность замен называется *выводом* (derivation) или *порождением*[1] $-(\mathbf{id})$ из E. Это порождение доказывает, что строка $-(\mathbf{id})$ является конкретным примером выражения.

[1]В оригинале — derivations (выводы). В связи с неоднозначностью общепринятого в русскоязычной литературе термина "вывод" далее используется термин "порождение". — *Прим. пер.*

Чтобы дать общее определение порождения, рассмотрим нетерминал A в середине последовательности грамматических символов, как в случае $\alpha A\beta$, где α и β — произвольные строки грамматических символов. Предположим, что $A \to \gamma$ является продукцией. Тогда мы записываем $\alpha A\beta \Rightarrow \alpha\gamma\beta$. Символ $\Rightarrow$ означает "порождение за один шаг". Если последовательность порождений $\alpha_1 \Rightarrow \alpha_2 \Rightarrow \cdots \Rightarrow \alpha_n$ переписывает α_1, заменяя его α_n, мы говорим, что α_1 *порождает* α_n. Часто нам надо сказать "порождает за нуль или более шагов". Для этой цели используется символ $\overset{*}{\Rightarrow}$. Таким образом,

1) $\alpha \overset{*}{\Rightarrow} \alpha$ для любой строки α;

2) если $\alpha \overset{*}{\Rightarrow} \beta$ и $\beta \Rightarrow \gamma$, то $\alpha \overset{*}{\Rightarrow} \gamma$.

Аналогично символ $\overset{+}{\Rightarrow}$ используется для обозначения "порождает за один или более шагов".

Если $S \overset{*}{\Rightarrow} \alpha$, где S — стартовый символ грамматики G, то мы говорим, что α является *сентенциальной формой*[2] G. Заметим, что сентенциальная форма может содержать как терминалы, так и нетерминалы, а также быть пустой. *Предложение* G представляет собой сентенциальную форму без нетерминалов. *Язык, генерируемый* грамматикой, представляет собой множество ее предложений. Таким образом, строка терминалов w принадлежит генерируемому грамматикой G языку $L(G)$ тогда и только тогда, когда w является предложением G (или $S \overset{*}{\Rightarrow} w$). Язык, который может быть сгенерирован грамматикой, называется *контекстно-свободным*языком. Если две грамматики генерируют один и тот же язык, то эти грамматики называются *эквивалентными*.

Строка $-(\mathbf{id} + \mathbf{id})$ является предложением грамматики (4.5) ввиду наличия порождения

$$E \Rightarrow -E \Rightarrow -(E) \Rightarrow -(E+E) \Rightarrow -(\mathbf{id}+E) \Rightarrow -(\mathbf{id}+\mathbf{id}) \tag{4.6}$$

Строки E, $-E$, $-(E)$, $\ldots$, $-(\mathbf{id}+\mathbf{id})$ представляют собой сентенциальные формы данной грамматики. Для указания того, что $-(\mathbf{id}+\mathbf{id})$ может быть порождено из E, мы записываем $E \overset{*}{\Rightarrow} -(\mathbf{id}+\mathbf{id})$.

На каждом шаге порождения осуществляется два выбора — мы должны выбрать заменяемый нетерминал, а затем продукцию, у которой данный нетерминал является заголовком. Например, приведенное ниже альтернативное порождение для $-(\mathbf{id}+\mathbf{id})$ отличается от порождения (4.6) последними двумя шагами:

$$E \Rightarrow -E \Rightarrow -(E) \Rightarrow -(E+E) \Rightarrow -(E+\mathbf{id}) \Rightarrow -(\mathbf{id}+\mathbf{id}) \tag{4.7}$$

Каждый нетерминал замещается тем же телом, что и ранее, но в другом порядке.

[2]В русскоязычной литературе также используется термин "цепочка": цепочка терминалов и нетерминалов. — *Прим. ред.*

Чтобы понять, как работают синтаксические анализаторы, рассмотрим порождения, в которых замещаемый нетерминал на каждом шаге выбирается следующим образом.

1. В *левых* (leftmost) порождениях в каждом предложении всегда выбирается крайний слева нетерминал. Если $\alpha \Rightarrow \beta$ является шагом, на котором замещается крайний слева нетерминал α, то это записывается как $\alpha \underset{lm}{\Rightarrow} \beta$.

2. В *правых* (rightmost) порождениях в каждом предложении всегда выбирается крайний справа нетерминал; в этом случае мы пишем $\alpha \underset{rm}{\Rightarrow} \beta$.

Порождение (4.6) левое, поскольку его можно переписать как

$$E \underset{lm}{\Rightarrow} -E \underset{lm}{\Rightarrow} -(E) \underset{lm}{\Rightarrow} -(E+E) \underset{lm}{\Rightarrow} -(\mathbf{id}+E) \underset{lm}{\Rightarrow} -(\mathbf{id}+\mathbf{id})$$

Заметим, что порождение (4.7) — правое.

С использованием наших соглашений по обозначениям каждый левый шаг может быть записан как $wA\gamma \underset{lm}{\Rightarrow} w\delta\gamma$, где w состоит только из терминалов, $A \to \delta$ — примененная продукция, а γ — строка грамматических символов. Чтобы подчеркнуть, что α порождает β путем левого порождения, мы записываем $\alpha \overset{*}{\underset{lm}{\Rightarrow}} \beta$. Если $S \overset{*}{\underset{lm}{\Rightarrow}} \alpha$, то мы говорим, что α — *левосентенциальная форма* рассматриваемой грамматики.

Аналогичные определения выполняются и для правых порождений. Правые порождения иногда называются *каноническими*.

4.2.4 Деревья разбора и порождения

Дерево разбора может рассматриваться как графическое представление порождения, из которого удалена информация о порядке замещения нетерминалов. Каждый внутренний узел дерева разбора представляет применение продукции. Внутренний узел дерева помечен нетерминалом A из заголовка соответствующей продукции, а дочерние узлы слева направо — символами из тела продукции, использованной в порождении для замены A.

Например, дерево разбора для $-(\mathbf{id}+\mathbf{id})$, показанное на рис. 4.3, получается как в результате порождения (4.6), так и в результате порождения (4.7).

Листья дерева разбора помечены нетерминалами или терминалами и, будучи прочитаны слева направо, образуют сентенциальную форму, называемую *кроной* (yield) или *границей* (frontier) дерева.

Для того чтобы увидеть взаимосвязь между порождением и деревьями разбора, рассмотрим произвольное приведение $\alpha_1 \Rightarrow \alpha_2 \Rightarrow \cdots \Rightarrow \alpha_n$, где α_1 — отдельный нетерминал A. Для каждой сентенциальной формы α_i в приведении

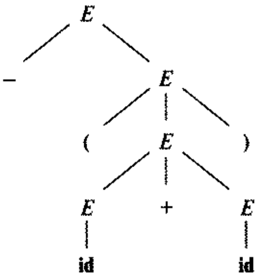


Рис. 4.3. Дерево разбора для $-(\mathbf{id}+\mathbf{id})$

можно построить дерево разбора, кроной которого является α_i. Этот процесс представляет собой индукцию по i.

БАЗИС: дерево для $\alpha_1 = A$ представляет собой единственный узел, помеченный A.

ИНДУКЦИЯ: предположим, что мы уже построили дерево разбора, имеющее крону $\alpha_{i-1} = X_1 X_2 \dots X_k$ (заметим, что в соответствии с нашими соглашениями об обозначениях X_i может означать нетерминал или терминал). Предположим, что α_{i-1} порождает α_i заменой нетерминала X_j на $\beta = Y_1 Y_2 \dots Y_m$. Иначе говоря, на i-м шаге порождения к α_{i-1} применяется продукция $X_j \to \beta$, порождая $\alpha_i = X_1 X_2 \dots X_{j-1} \beta X_{j+1} \dots X_k$.

Для моделирования этого шага порождения находим j-й слева не-ϵ-лист в текущем дереве разбора, который помечен X_j. Мы даем этому листу m дочерних узлов, помеченных $Y_1, Y_2, \dots, Y_m$ слева направо. В частном случае, если $m = 0$, то $\beta = \epsilon$, и у j-го листа появляется один дочерний узел, помеченный ϵ.

Пример 4.3. Последовательность деревьев разбора, построенная на основе порождения (4.6), показана на рис. 4.4. Первый шаг этого порождения — $E \Rightarrow -E$. Для моделирования этого шага мы добавляем к корню E начального дерева два дочерних узла, помеченных - и E.

Вторым шагом порождения является $-E \Rightarrow -(E)$. Соответственно, мы добавляем три дочерних узла, помеченных (, E и), к листу E во втором дереве для получения третьего дерева с кроной $-(E)$. Продолжая построения описанным способом, мы получим в качестве полного дерева разбора шестое дерево последовательности. □

Поскольку дерево разбора игнорирует порядок, в котором производилось замещение символов в сентенциальной форме, между порождениями и деревьями разбора возникает соотношение "многие к одному". Например, и порождение (4.6), и порождение (4.7) связаны с одним и тем же окончательным деревом разбора, показанным на рис. 4.4.

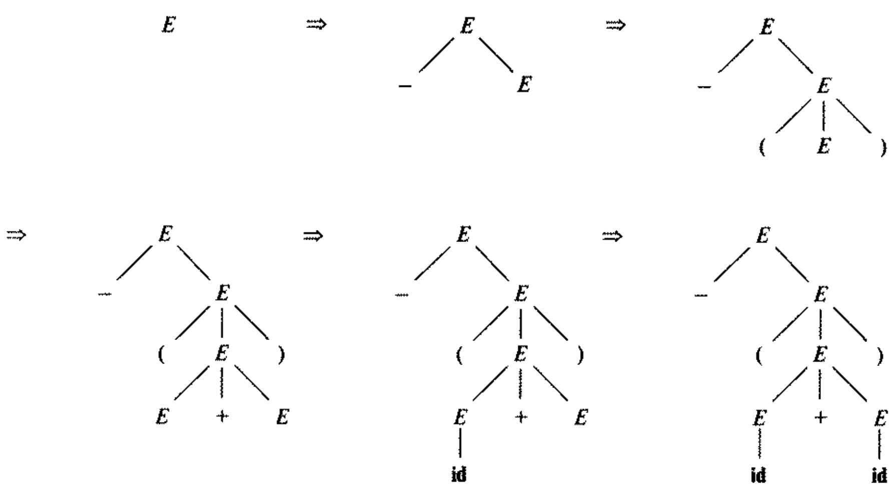


Рис. 4.4. Последовательность деревьев разбора для порождения (4.6)

Далее мы зачастую будем выполнять синтаксический анализ при помощи левых или правых порождений, поскольку между деревьями разбора и левыми (как и правыми) порождениями существует взаимно однозначное соответствие. И левое, и правое порождения выбирают определенный порядок замещения символов в сентенциальной форме, так что все варианты с иным порядком отсеиваются. Нетрудно показать, что каждое дерево разбора связано с единственным левым и единственным правым порождением.

4.2.5 Неоднозначность

Из раздела 2.2.4 мы знаем, что грамматика, которая дает более одного дерева разбора для некоторого предложения, называется *неоднозначной* (ambiguous). Иначе говоря, неоднозначная грамматика — это грамматика, которая для одного и того же предложения дает не менее двух левых или правых порождений.

Пример 4.4. Грамматика для арифметических выражений (4.3) допускает два разных левых порождения для предложения $\mathbf{id} + \mathbf{id} * \mathbf{id}$:

$$\begin{array}{lllllll} E & \Rightarrow & E + E & \quad & E & \Rightarrow & E * E \\ & \Rightarrow & \mathbf{id} + E & & & \Rightarrow & E + E * E \\ & \Rightarrow & \mathbf{id} + E * E & & & \Rightarrow & \mathbf{id} + E * E \\ & \Rightarrow & \mathbf{id} + \mathbf{id} * E & & & \Rightarrow & \mathbf{id} + \mathbf{id} * E \\ & \Rightarrow & \mathbf{id} + \mathbf{id} * \mathbf{id} & & & \Rightarrow & \mathbf{id} + \mathbf{id} * \mathbf{id} \end{array}$$

Соответствующие деревья разбора показаны на рис. 4.5.

Обратите внимание, что дерево на рис. 4.5, *а* отражает обычно принимаемые приоритеты операторов + и *, в то время как дерево на рис. 4.5, *б* отражает обратное соотношение приоритетов. Иными словами, обычно считается, что приоритет оператора * выше, чем приоритет оператора +, так что обычно выражение наподобие $a + b * c$ вычисляется как $a + (b * c)$, а не как $(a + b) * c$. □

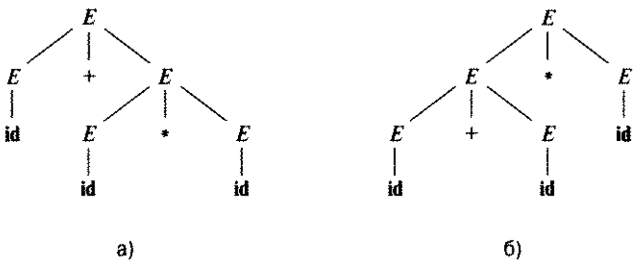


Рис. 4.5. Два дерева разбора для предложения **id** + **id** * **id**

Для большинства синтаксических анализаторов грамматика должна быть однозначной, поскольку, если это не так, мы не в состоянии определить дерево разбора для предложения единственным образом. В некоторых случаях удобно использовать тщательно отобранную неоднозначную грамматику совместно с *правилами устранения неоднозначностей* (disambiguating rules), которые "отбрасывают" нежелательные деревья разбора, оставляя для каждого предложения по одному дереву.

4.2.6 Проверка языка, сгенерированного грамматикой

Хотя разработчики компиляторов редко делают это для полной грамматики языка программирования, очень полезно иметь возможность убедиться, что данное множество продукций генерирует определенный язык. Проблемные конструкции можно изучить, написав сокращенный, абстрактный вариант грамматики и изучив генерируемый ею язык. Ниже мы построим такую грамматику для условных инструкций.

Для доказательства того, что грамматика G порождает язык L, мы должны показать, что любая строка, генерируемая G, принадлежит L и, наоборот, что каждая строка из L может быть порождена грамматикой G.

Пример 4.5. Рассмотрим грамматику

$$S \rightarrow (S)\ S \mid \epsilon \tag{4.8}$$

Хотя на первый взгляд это не очевидно, данная простая грамматика порождает все строки из сбалансированных скобок, и только такие строки. Чтобы убедиться

в этом, вначале покажем, что каждое предложение, выводимое из S, сбалансировано, а затем — что каждая сбалансированная строка может быть выведена из S. Для того чтобы показать, что каждое предложение, выводимое из S, сбалансировано, воспользуемся индукцией по количеству шагов n в порождении.

БАЗИС: $n = 1$. Единственная строка терминалов, порождаемая из S за один шаг, — это пустая строка, которая, само собой, является сбалансированной.

ИНДУКЦИЯ: теперь предположим, что все порождения из менее чем n шагов приводят к сбалансированным предложениям, и рассмотрим левое порождение, состоящее ровно из n шагов. Такое порождение должно иметь вид

$$S \underset{lm}{\Rightarrow} (S)\ S \overset{*}{\underset{lm}{\Rightarrow}} (x)\ S \overset{*}{\underset{lm}{\Rightarrow}} (x)\,y$$

Порождения x и y из S занимают менее n шагов и согласно гипотезе индукции x и y сбалансированы. Следовательно, строка $(x)\,y$ должна быть сбалансированной, т.е. иметь одинаковое количество левых и правых скобок, причем любой префикс строки содержит как минимум столько же левых скобок, сколько и правых.

Таким образом, любая строка, выводимая из S, сбалансирована. Теперь нужно показать, что любая сбалансированная строка может быть получена из S. Для этого воспользуемся индукцией по длине строки.

БАЗИС: Если строка имеет длину 0, это пустая строка ϵ, являющаяся сбалансированной.

ИНДУКЦИЯ: Сначала заметим, что каждая сбалансированная строка имеет четную длину. Предположим, что любая сбалансированная строка длиной менее $2n$ порождаема из S, и рассмотрим сбалансированную строку w длиной $2n$, где $n \geqslant 1$. Несомненно, что w начинается с левой скобки. Пусть (x) — кратчайший префикс w, имеющий одинаковое количество левых и правых скобок. Тогда w можно записать как $(x)\,y$, где и x, и y сбалансированы. Поскольку и x, и y имеют длину менее $2n$, они порождаемы из S согласно гипотезе индукции. Таким образом, можно найти порождение вида

$$S \rightarrow (S)\ S \overset{*}{\Rightarrow} (x)\ S \overset{*}{\Rightarrow} (x)\,y$$

которое доказывает, что $w = (x)\,y$ также порождаемо из S. □

4.2.7 Контекстно-свободные грамматики и регулярные выражения

Перед тем как завершить этот раздел, посвященный грамматикам и их свойствам, мы убедимся, что грамматики представляют собой более мощные обозначения по сравнению с регулярными выражениями. Каждая конструкция, которая

может быть описана регулярным выражением, может быть описана и грамматикой, но не наоборот. Говоря иначе, каждый регулярный язык является контекстно-свободным, но не наоборот.

Например, регулярное выражение (**a** | **b**)* **abb** и грамматика

$$\begin{array}{lcl} A_0 & \rightarrow & aA_0 \mid bA_0 \mid aA_1 \\ A_1 & \rightarrow & bA_2 \\ A_2 & \rightarrow & bA_3 \\ A_3 & \rightarrow & \epsilon \end{array}$$

описывают один и тот же язык, множество строк из a и b, заканчивающихся на abb.

Можно механически построить грамматику для распознавания того же языка, что и распознаваемый недетерминированным конечным автоматом (НКА). Приведенная выше грамматика построена на основе НКА на рис. 3.24 следующим образом.

1. Для каждого состояния i в НКА создается нетерминал A_i.

2. Если состояние i имеет переход в состояние j по символу a, добавляем в грамматику продукцию $A_i \rightarrow aA_j$. Если состояние i переходит в состояние j для пустой строки ϵ, добавляем продукцию $A_i \rightarrow A_j$.

3. Если i — принимающее состояние, добавляем продукцию $A_i \rightarrow \epsilon$.

4. Если i — начальное состояние, делаем A_i стартовым символом грамматики.

С другой стороны, язык $L = \{a^n b^n \mid n \geqslant 1\}$ с одинаковым количеством a и b в строках является прототипичным примером языка, который может быть описан грамматикой, но не регулярным выражением. Чтобы убедиться в этом, предположим, что L — язык, определенный некоторым регулярным выражением. Мы можем построить ДКА D с конечным числом состояний, скажем, k, который принимает L. Поскольку D имеет только k состояний, при входной строке, начинающейся более чем с k символов a, автомат D должен войти в некоторое состояние дважды, скажем, в состояние s_i, как показано на рис. 4.6. Предположим, что путь из s_i в себя же помечен последовательностью a^{j-i}. Поскольку строка $a^i b^i$ принадлежит языку, должен существовать путь, помеченный b^i, из s_i в принимающее состояние f. Однако в таком случае имеется путь из начального состояния s_0 в принимающее состояние f, проходящий через s_i и помеченный $a^j b^i$, как показано на рис. 4.6. Таким образом, конечный автомат D принимает также строку $a^j b^i$, не принадлежащую языку, что противоречит предположению о том, что L — язык, принимаемый D.

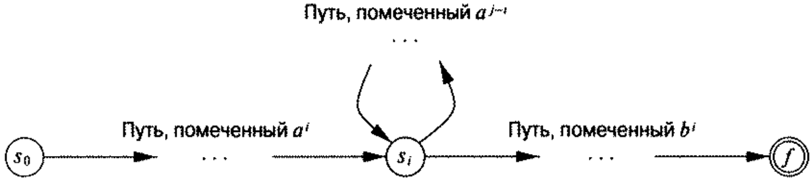


Рис. 4.6. ДКА D, принимающий как a^ib^i, так и a^jb^i

Говоря простым языком, "конечный автомат не умеет считать", а это означает, что конечный автомат не может принимать язык наподобие $\{a^nb^n \mid n \geqslant 1\}$, в котором требуется запоминать количество символов a перед просмотром символов b. Аналогично "грамматика может считать две вещи, но не три", как мы увидим при рассмотрении конструкций не контекстно-свободного языка в разделе 4.3.5.

4.2.8 Упражнения к разделу 4.2

Упражнение 4.2.1. Рассмотрим контекстно-свободную грамматику

$$S \to S\ S + \mid S\ S\ * \mid a$$

и строку *aa+a**.

а) Приведите левое порождение строки.

б) Приведите правое порождение строки.

в) Приведите дерево разбора строки.

! г) Является ли данная грамматика однозначной? Обоснуйте свой ответ.

! д) Опишите язык, генерируемый этой грамматикой.

Упражнение 4.2.2. Повторите упражнение 4.2.1 для следующих грамматик и строк.

а) Грамматика $S \to 0\ S\ 1 \mid 0\ 1$, строка 000111.

б) Грамматика $S \to +S\ S \mid *\ S\ S \mid a$, строка +**aaa*.

! в) Грамматика $S \to S\ (S)\ S \mid \epsilon$, строка (( ) ( )).

! г) Грамматика $S \to S + S \mid S\ S \mid (S) \mid S\ * \mid a$, строка $(a + a) * a$.

! д) Грамматика $S \to (L) \mid a$ и $L \to L, S \mid S$, строка $((a, a), a, (a))$.

!! е) Грамматика $S \to a\ S\ b\ S \mid b\ S\ a\ S \mid \epsilon$, строка *aabbab*.

! ж) Грамматика для булевых выражений:

$$\begin{array}{rcl} bexpr & \rightarrow & bexpr \ \mathbf{or} \ bterm \mid bterm \\ bterm & \rightarrow & bterm \ \mathbf{and} \ bfactor \mid bfactor \\ bfactor & \rightarrow & \mathbf{not} \ bfactor \mid (bexpr) \mid \mathbf{true} \mid \mathbf{false} \end{array}$$

Упражнение 4.2.3. Разработайте грамматику для следующих языков.

а) Множество всех строк из 0 и 1, таких, что за каждым 0 следует, по меньшей мере, одна 1.

! б) Множество всех строк из 0 и 1, являющихся *палиндромами*, т.е. строками, которые одинаковы при чтении как слева направо, так и справа налево.

! в) Множество всех строк из 0 и 1 с одинаковым количеством 0 и 1.

!! г) Множество всех строк из 0 и 1 с количеством 0, не равным количеству 1.

! д) Множество всех строк из 0 и 1, в которых не встречается 011 в качестве подстроки.

!! е) Множество всех строк из 0 и 1 вида xy, где $x \neq y$ и x и y имеют одинаковую длину.

Упражнение 4.2.4. Существуют достаточно распространенные расширения описания грамматик. В их число входят являющиеся метасимволами (как $\rightarrow$ или $|$) квадратные и фигурные скобки в телах продукций, имеющие следующие значения.

i) Квадратные скобки, в которые заключен грамматический символ или символы, означают необязательность данной конструкции. Таким образом, продукция $A \rightarrow X\ [Y]\ Z$ имеет то же действие, что и две продукции $A \rightarrow X\ Y\ Z$ и $A \rightarrow X\ Z$.

ii) Фигурные скобки вокруг грамматического символа или символов говорят о том, что эти символы могут повторяться любое число раз, включая нуль. Таким образом, продукция $A \rightarrow X\ \{Y\ Z\}$ эквивалентна бесконечной последовательности продукций $A \rightarrow X$, $A \rightarrow X\ Y\ Z$, $A \rightarrow X\ Y\ Z\ Y\ Z$ и т.д.

Покажите, что эти два расширения не повышают мощность грамматик, т.е. что любой язык, который может быть сгенерирован грамматикой с данными расширениями, может быть сгенерирован и грамматикой без расширений.

Упражнение 4.2.5. Воспользуйтесь фигурными скобками, описанными в упражнении 4.2.4, чтобы упростить приведенную ниже грамматику блоков и условных конструкций:

$$\begin{array}{lcl} \mathit{stmt} & \to & \textbf{if}\ \mathit{expr}\ \textbf{then}\ \mathit{stmt}\ \textbf{else}\ \mathit{stmt} \\ & | & \textbf{if}\ \mathit{stmt}\ \textbf{then}\ \mathit{stmt} \\ & | & \textbf{begin}\ \mathit{stmtList}\ \textbf{end} \\ \mathit{stmtList} & \to & \mathit{stmt}\ ;\ \mathit{stmtList} \mid \mathit{stmt} \end{array}$$

! **Упражнение 4.2.6.** Расширьте идею, приведенную в упражнении 4.2.4, чтобы позволить использование в теле продукций произвольных регулярных выражений грамматических символов. Покажите, что такое расширение не позволяет грамматикам определить ни один новый язык.

! **Упражнение 4.2.7.** Грамматический символ X (терминал или нетерминал) *бесполезен* (useless), если не существует порождения вида $S \overset{*}{\Rightarrow} wXy \overset{*}{\Rightarrow} wxy$, т.е. X никогда не появляется в порождении некоторого предложения.

а) Разработайте алгоритм для удаления из грамматики всех продукций, содержащих бесполезные символы.

б) Примените свой алгоритм к следующей грамматике:

$$\begin{array}{lcl} S & \to & 0 \mid A \\ A & \to & AB \\ B & \to & 1 \end{array}$$

Упражнение 4.2.8. Грамматика на рис. 4.7 генерирует объявления для отдельных числовых идентификаторов; эти объявления включают четыре различных независимых свойства чисел.

а) Обобщите грамматику на рис. 4.7, чтобы она позволяла иметь n атрибутов A_i для некоторого фиксированного n и для $i = 1, 2, \ldots, n$, где A_i может быть либо a_i, либо b_i. Ваша грамматика должна использовать только $O(n)$ грамматических символов и иметь общую длину продукций, равную $O(n)$.

! б) Грамматика на рис. 4.7 и ее обобщение в части *а* упражнения допускают противоречащие и/или избыточные объявления, такие как

```
declare foo real fixed real floating
```

Можно потребовать, чтобы синтаксис языка запрещал такие объявления, т.е. чтобы каждое объявление, генерируемое грамматикой, имело только

одно значение для каждого из n атрибутов. Если сделать это, то для любого фиксированного n будет существовать только конечное количество корректных объявлений. Язык корректных объявлений (как и любой конечный язык) имеет грамматику (а также регулярное выражение). Очевидная грамматика, в которой стартовый символ имеет продукцию для каждого корректного объявления, содержит $n!$ продукций общей длиной $O(n \times n!)$. Вы должны справиться лучше: общая длина продукций вашей грамматики должна составлять $O(n2^n)$.

!! в) Покажите, что любая грамматика из части *б* упражнения должна иметь общую длину продукций не менее 2^n.

г) Что говорит часть *в* упражнения об обеспечении требования неизбыточности и непротиворечивости атрибутов объявлений посредством синтаксиса языка программирования?

stmt	→	**declare id** *optionList*
optionList	→	*optionList option* | ϵ
option	→	*mode* | *scale* | *precision* | *base*
mode	→	**real** | **complex**
scale	→	**fixed** | **floating**
precision	→	**single** | **double**
base	→	**binary** | **decimal**

Рис. 4.7. Грамматика для многоатрибутных объявлений

4.3 Разработка грамматики

Грамматикой можно описать большую часть (но не весь) синтаксиса языков программирования. Например, требование объявления идентификаторов до их использования не может быть описано контекстно-свободной грамматикой. Значит, последовательности токенов, принимаемые синтаксическим анализатором, образуют надмножество языка программирования; последующие фазы компилятора должны анализировать выход синтаксического анализатора, чтобы обеспечить его соответствие правилам, которые не проверяются синтаксическим анализатором.

Этот раздел начинается с рассмотрения разделения работы между лексическим и синтаксическим анализаторами. Затем будут рассмотрены преобразования, которые могут быть применены для получения грамматики, в большей степени приспособленной для синтаксического анализа. Один метод предназначен

для устранения неоднозначности грамматики, другие (устранение левой рекурсии и левая факторизация) — для переписывания грамматики в виде, пригодном для нисходящего синтаксического анализа. Завершится этот раздел рассмотрением некоторых конструкций языков программирования, которые не могут быть описаны ни одной грамматикой.

4.3.1 Лексический и синтаксический анализ

Как говорилось в разделе 4.2.7, все, что может быть описано при помощи регулярного выражения, может быть также описано и грамматикой. Закономерен вопрос — почему же для определения лексического синтаксиса языка используются регулярные выражения? На то есть несколько причин.

1. Разделение синтаксической структуры языка на лексическую и не лексическую части представляет собой удобный способ разбиения начальной стадии компиляции на два модуля меньшего размера, что упрощает работу с ними.

2. Лексические правила языка часто очень просты, и для их описания не требуется такое мощное средство, как грамматика.

3. Регулярные выражения в общем случае предоставляют по сравнению с грамматикой более краткую и простую для понимания запись токенов.

4. Автоматически создаваемые на основе регулярных выражений лексические анализаторы более эффективны, чем лексические анализаторы, построенные на основе грамматики.

Четких правил, что именно следует отнести к лексическим правилам, а что — к синтаксическим, нет. Регулярные выражения в большей степени подходят для описания структуры таких конструкций, как идентификаторы, константы, ключевые слова и пробельные символы. Грамматики же более приспособлены для описания вложенных структур, таких как сбалансированные скобки, пары **begin-end**, соответствующие друг другу **if-then-else**, и т.п. Эти вложенные структуры не могут быть описаны регулярными выражениями.

4.3.2 Устранение неоднозначности

Иногда для устранения неоднозначности грамматика может быть переписана. В качестве примера устраним неоднозначность из следующей грамматики с "висящим else":

$$\begin{array}{lcl} stmt & \rightarrow & \textbf{if}\ expr\ \textbf{then}\ stmt \\ & | & \textbf{if}\ expr\ \textbf{then}\ stmt\ \textbf{else}\ stmt \\ & | & \textbf{other} \end{array} \tag{4.9}$$

Здесь **other** означает любую другую инструкцию. Согласно этой грамматике составная условная инструкция

$$\textbf{if } E_1 \textbf{ then } S_1 \textbf{ else if } E_2 \textbf{ then } S_2 \textbf{ else } S_3$$

имеет дерево разбора, показанное на рис. 4.8[3]. Грамматика (4.9) неоднозначна, поскольку строка

$$\textbf{if } E_1 \textbf{ then if } E_2 \textbf{ then } S_1 \textbf{ else } S_2 \qquad (4.10)$$

имеет два дерева разбора, показанных на рис. 4.9.

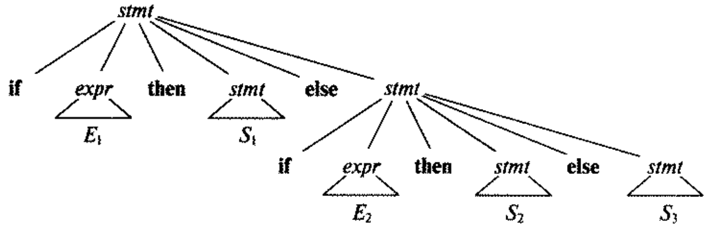


Рис. 4.8. Дерево разбора для условной инструкции

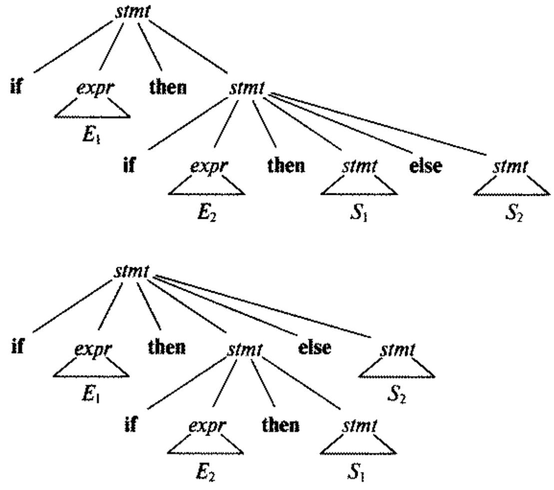


Рис. 4.9. Два дерева разбора неоднозначного предложения

Во всех языках программирования с условными инструкциями такого вида предпочтительно первое дерево разбора. Общее правило гласит: "сопоставить

[3]Нижние индексы у E и S не означают разные нетерминалы; они предназначены только для того, чтобы отличать разные вхождения одного и того же нетерминала.

каждое **else** ближайшему незанятому **then**"[4]. Это правило устранения неоднозначности теоретически может быть внедрено непосредственно в грамматику, но на практике его редко реализуют в виде продукций.

Пример 4.6. *Мы можем переписать грамматику (4.9) как однозначную. Идея за*ключается в том, что инструкция, появляющаяся между **then** и **else**, должна быть "сбалансированная" (matched), т.е. не должна оканчиваться открытым или не соответствующим некоторому **else** ключевым словом **then**. Такая "сбалансированная" инструкция может либо представлять собой полную инструкцию **if-then-else**, не содержащую открытых инструкций, либо быть любой инструкцией, отличающейся от условной. Таким образом, мы получаем грамматику, показанную на рис. 4.10. Эта грамматика генерирует те же строки, что и грамматика с висящим **else** (4.9), но для строки (4.10) дает только одно дерево разбора, а именно — то, которое связывает каждое **else** с ближайшим незанятым **then**. □

$$
\begin{array}{rcl}
stmt & \rightarrow & matched_stmt \\
 & | & open_stmt \\
matched_stmt & \rightarrow & \textbf{if}\ expr\ \textbf{then}\ matched_stmt\ \textbf{else}\ matched_stmt \\
 & | & \textbf{other} \\
open_stmt & \rightarrow & \textbf{if}\ expr\ \textbf{then}\ stmt \\
 & | & \textbf{if}\ expr\ \textbf{then}\ matched_stmt\ \textbf{else}\ open_stmt
\end{array}
$$

Рис. 4.10. Однозначная грамматика для инструкций if-then-else

4.3.3 Устранение левой рекурсии

Грамматика является *леворекурсивной* (left recursive), если в ней имеется нетерминал A, такой, что существует порождение $A \stackrel{+}{\Rightarrow} A\alpha$ для некоторой строки α. Методы нисходящего разбора не в состоянии работать с леворекурсивными грамматиками, поэтому требуется преобразование грамматики, которое устранило бы из нее левую рекурсию. В разделе 2.4.5 мы рассматривали *непосредственную левую рекурсию* (immediate left recursion), при которой существует продукция вида $A \rightarrow A\alpha$. Сейчас же будет рассмотрен общий случай. В разделе 2.4.5 показано, как леворекурсивная пара продукций $A \rightarrow A\alpha \mid \beta$ может быть заменена нелеворекурсивными продукциями

$$
\begin{array}{rcl}
A & \rightarrow & \beta A' \\
A' & \rightarrow & \alpha A' \mid \epsilon
\end{array}
$$

[4]Следует заметить, что C и производные от него языки программирования включены в этот класс. Несмотря на то, что в семействе C-образных языков не используется ключевое слово **then**, его роль играет закрывающая скобка условия, следующего за **if**.

без изменения строк, порождаемых из A. Одного этого правила достаточно для большого количества грамматик.

Пример 4.7. Повторенная здесь нелеворекурсивная грамматика для выражений (4.2)

$$\begin{array}{lcl} E & \to & T\,E' \\ E' & \to & +T\,E' \mid \epsilon \\ T & \to & F\,T' \\ T' & \to & *\,F\,T' \mid \epsilon \\ F & \to & (E) \mid \textbf{id} \end{array}$$

получается путем устранения непосредственной левой рекурсии из грамматики выражений (4.1). Леворекурсивная пара продукций $E \to E + T \mid T$ заменяется двумя продукциями, $E \to T\ E'$ и $E' \to +T\ E' \mid \epsilon$. Новые продукции для T и T' получаются аналогично при устранении левой рекурсии из продукции $T \to T\ *\ F \mid F$. □

Непосредственная левая рекурсия может быть устранена при помощи следующего метода, который работает для любого количества A-продукций. Вначале сгруппируем A-продукции следующим образом

$$A \to A\alpha_1 \mid A\alpha_2 \mid \cdots \mid A\alpha_m \mid \beta_1 \mid \beta_2 \mid \cdots \mid \beta_n$$

Здесь ни одно β_i не начинается с A. Затем заменим эти A-продукции следующим образом:

$$\begin{array}{lcl} A & \to & \beta_1 A' \mid \beta_2 A' \mid \cdots \mid \beta_n A' \\ A' & \to & \alpha_1 A' \mid \alpha_2 A' \mid \cdots \mid \alpha_m A' \mid \epsilon \end{array}$$

Нетерминал A порождает те же строки, что и ранее, но без левой рекурсии. Эта процедура устраняет все левые рекурсии из продукций для A и A' (при условии, что ни одна строка α_i не является ϵ), но не устраняет левую рекурсию, вызванную двумя или более шагами порождения. Рассмотрим, например, грамматику

$$\begin{array}{lcl} S & \to & A\,a \mid b \\ A & \to & A\,c \mid S\,d \mid \epsilon \end{array} \tag{4.11}$$

Нетерминал S леворекурсивен, поскольку $S \Rightarrow Aa \Rightarrow Sda$, но эта рекурсия не является непосредственной.

Алгоритм 4.8, приведенный ниже, систематически удаляет из грамматики левую рекурсию. Он гарантированно работает с грамматиками, не имеющими циклов (порождений типа $A \overset{+}{\Rightarrow} A$) и ϵ-продукций (продукций вида $A \to \epsilon$). Циклы, как и ϵ-продукции, также могут быть удалены из грамматики систематическим образом (см. упражнения 4.4.6 и 4.4.7).

Алгоритм 4.8. Устранение левой рекурсии

Вход: грамматика G без циклов и ϵ-продукций.

Выход: эквивалентная грамматика без левой рекурсии.

Метод: применить алгоритм, приведенный на рис. 4.11. Обратите внимание, что получающаяся грамматика без левых рекурсий может иметь ϵ-продукции. □

```
1)  Расположить нетерминалы в некотором порядке A_1, A_2, ..., A_n.
2)  for ( каждое i от 1 до n ) {
3)      for ( каждое j от 1 до i - 1 ) {
4)          заменить каждую продукцию вида A_i → A_j продукциями
                A_i → δ_1γ | δ_2γ | ··· | δ_kγ, где A_j → δ_1 | δ_2 | ··· | δ_k — все
                текущие A_j-продукции
5)      }
6)      устранить непосредственную левую рекурсию среди A_i-продукций
7)  }
```

Рис. 4.11. Алгоритм для устранения левой рекурсии из грамматики

Процедура на рис. 4.11 работает следующим образом. В первой итерации для $i = 1$ внешний цикл в строках 2–7 устраняет непосредственную левую рекурсию из A_1-продукций. Все остальные A_1-продукции вида $A_1 \to A_l\alpha$ должны, таким образом, иметь $l > 1$. После $i-1$-й итерации внешнего цикла все нетерминалы A_k, где $k < i$, оказываются "чистыми", т.е. все продукции $A_k \to A_l\alpha$ должны иметь $l > k$. В результате на i-й итерации внутренний цикл в строках 3–5 постепенно поднимает нижнюю границу всех продукций $A_i \to A_m\alpha$, пока мы не получим $m \geqslant i$. Затем устранение непосредственной левой рекурсии из A_i-продукций в строке 6 делает m больше i.

Пример 4.9. Применим алгоритм 4.8 к грамматике (4.11). Технически из-за наличия ϵ-продукции алгоритм может не работать, но в данном случае продукция $A \to \epsilon$ не мешает работе.

Мы располагаем нетерминалы в порядке S, A. Непосредственной левой рекурсии среди S-продукций нет, так что в процессе работы внешнего цикла при $i = 1$ ничего не происходит. При $i = 2$ мы подставляем S-продукцию в $A \to S\ d$ для получения следующих A-продукций:

$$A \to A\ c \mid A\ a\ d \mid b\ d \mid \epsilon$$

Устранение непосредственной левой рекурсии среди A-продукций дает грамматику

$$\begin{array}{lcl} S & \to & A\,a \mid b \\ A & \to & b\,d\,A' \mid A' \\ A' & \to & c\,A' \mid a\,d\,A' \mid \epsilon \end{array}$$

□

4.3.4 Левая факторизация

Левая факторизация (left factoring) представляет собой преобразование грамматики в пригодную для предиктивного, или нисходящего, синтаксического анализа. Когда не ясно, какая из двух альтернативных продукций должна использоваться для нетерминала A, A-продукции можно переписать так, чтобы отложить принятие решения до тех пор, пока из входного потока не будет прочитано достаточно символов для правильного выбора.

Например, если есть продукции

$$\begin{array}{lcl} \mathit{stmt} & \to & \textbf{if}\ \mathit{expr}\ \textbf{then}\ \mathit{stmt}\ \textbf{else}\ \mathit{stmt} \\ & \mid & \textbf{if}\ \mathit{expr}\ \textbf{then}\ \mathit{stmt} \end{array}$$

то, обнаружив во входном потоке **if**, мы не в состоянии тут же выбрать ни одну из них. В общем случае, если $A \to \alpha\beta_1 \mid \alpha\beta_2$ представляют собой две A-продукции и входной поток начинается с непустой строки, порождаемой α, то мы не знаем, будет ли использоваться первая или вторая продукция. Однако можно отложить решение, расширив A до $\alpha A'$. В этом случае, после того как рассмотрен входной поток, порождаемый α, мы расширяем A' до β_1 или β_2. Таким образом, будучи левофакторизованными, исходные продукции превращаются в

$$\begin{array}{lcl} A & \to & \alpha A' \\ A' & \to & \beta_1 \mid \beta_2 \end{array}$$

Алгоритм 4.10. Левая факторизация грамматики

Вход: грамматика G.

Выход: эквивалентная левофакторизованная грамматика.

Метод: для каждого нетерминала A находим самый длинный префикс α, общий для двух или большего числа альтернатив. Если $\alpha \neq \epsilon$, т.е. имеется нетривиальный общий префикс, заменим все продукции $A \to \alpha\beta_1 \mid \alpha\beta_2 \mid \cdots \mid \alpha\beta_n \mid \gamma$, где γ представляет все альтернативы, не начинающиеся с α, продукциями

$$\begin{array}{lcl} A & \to & \alpha A' \mid \gamma \\ A' & \to & \beta_1 \mid \beta_2 \mid \cdots \mid \beta_n \end{array}$$

Здесь A' — новый нетерминал. Выполняем это преобразование до тех пор, пока никакие две альтернативы нетерминала не будут иметь общий префикс. □

Пример 4.11. Следующая грамматика абстрагирует проблему "висящего **else**":

$$\begin{array}{lcl} S & \rightarrow & i\,E\,t\,S \mid i\,E\,t\,S\,e\,S \mid a \\ E & \rightarrow & b \end{array} \tag{4.12}$$

Здесь i, t и e означают **if**, **then** и **else**, E и S соответствуют "условному выражению" и "инструкции". Будучи левофакторизованной, эта грамматика принимает следующий вид:

$$\begin{array}{lcl} S & \rightarrow & i\,E\,t\,S\,S' \mid a \\ S' & \rightarrow & e\,S \mid \epsilon \\ E & \rightarrow & b \end{array} \tag{4.13}$$

Таким образом, можно расширить S до $iEtSS'$ для входного i и подождать, пока из входного потока не будет считано $iEtS$, чтобы затем решить, расширять ли S' до eS или до ϵ. Конечно, обе эти грамматики неоднозначны и при обнаружении во входном потоке e будет неясно, какая из альтернатив для S' должна быть выбрана. В примере 4.19 обсуждается путь решения этой дилеммы. □

4.3.5 Не контекстно-свободные языковые конструкции

Некоторые синтаксические конструкции, имеющиеся в типичных языках программирования, не могут быть определены только лишь одной грамматикой. Здесь мы рассмотрим две такие конструкции с использованием для иллюстрации возникающих сложностей простых абстрактных языков.

Пример 4.12. Язык в этом примере абстрагирует проблему проверки того, что идентификаторы объявлены до их использования в программе. Язык состоит из строк вида wcw, где первое w представляет объявление идентификатора w, а второе w — его использование.

Упомянутый абстрактный язык — $L_1 = \{wcw \mid w \in (\mathbf{a} \mid \mathbf{b})^*\}$. L_1 состоит из всех слов, образованных повторяющимися строками из a и b, разделенными символом c, например $aabcaab$. Хотя доказательство того, что L_1 не является контекстно-свободным языком, выходит за рамки данной книги, из этого факта непосредственно следует, что такие языки программирования, как C и Java, требующие объявления идентификаторов до их использования и допускающие существование идентификаторов произвольной длины, не являются контекстно-свободными.

По этой причине грамматики C или Java не различают идентификаторы, являющиеся различными строками символов. Все идентификаторы представлены в их грамматиках токеном, таким как **id**. В компиляторах этих языков проверка

того, что идентификаторы объявлены до их использования, производится на фазе семантического анализа. □

Пример 4.13. Не контекстно-свободный язык в этом примере абстрагирует проблему проверки соответствия количества фактических параметров при вызове функции количеству формальных параметров в ее объявлении. Язык состоит из строк вида $a^n b^m c^n d^m$ (вспомните, что a^n означает a, записанное n раз). Здесь a^n и b^m могут представлять списки формальных параметров двух функций, объявленных с n и m аргументами соответственно, в то время как c^n и d^m представляют списки фактических параметров в вызовах этих двух функций.

Упомянутый абстрактный язык — $L_2 = \{a^n b^m c^n d^m \mid n \geqslant 1 \text{ и } m \geqslant 1\}$, т.е. L_2 состоит из строк, генерируемых регулярным выражением $\mathbf{a^*b^*c^*d^*}$, причем количество a в строке равно количеству c, а количество b — количеству d. Этот язык не является контекстно-свободным.

Опять же, типичный синтаксис объявления и использования функции сам по себе не рассматривает количество ее параметров. Например, вызов функции в С-подобном языке может быть определен как

$$\begin{array}{rcl} stmt & \rightarrow & \mathbf{id}\ (expr_list) \\ expr_list & \rightarrow & expr_list\ ,\ expr \\ & | & expr \end{array}$$

с соответствующими продукциями для *expr*. Проверка того, что число действительных параметров в вызове корректно, обычно выполняется во время семантического анализа. □

4.3.6 Упражнения к разделу 4.3

Упражнение 4.3.1. Ниже приведена грамматика для регулярных выражений над символами a и b (с использованием + вместо | для обозначения объединения, чтобы избежать конфликта с использованием вертикальной черты как метасимвола грамматики):

$$\begin{array}{lcl} rexpr & \rightarrow & rexpr + rterm \mid rterm \\ rterm & \rightarrow & rterm\ rfactor \mid rfactor \\ rfactor & \rightarrow & rfactor\ * \mid rprimary \\ rprimary & \rightarrow & \mathbf{a} \mid \mathbf{b} \end{array}$$

а) Выполните левую факторизацию данной грамматики.

б) Делает ли левая факторизация данную грамматику пригодной для нисходящего синтаксического анализа?

в) В дополнение к левой факторизации устраните из исходной грамматики левую рекурсию.

г) Является ли полученная грамматика пригодной для нисходящего синтаксического анализа?

Упражнение 4.3.2. Повторите упражнение 4.3.1 для следующих грамматик.

а) Грамматика из упражнения 4.2.1.

б) Грамматика из упражнения 4.2.2, *а*.

в) Грамматика из упражнения 4.2.2, *в*.

г) Грамматика из упражнения 4.2.2, *д*.

д) Грамматика из упражнения 4.2.2, *ж*.

! **Упражнение 4.3.3.** Приведенная ниже грамматика предложена для устранения "неоднозначности висящего **else**", рассматривавшегося в разделе 4.3.2:

$$\begin{array}{lcl} stmt & \rightarrow & \textbf{if}\ expr\ \textbf{then}\ stmt \\ & | & matchedStmt \\ matchedStmt & \rightarrow & \textbf{if}\ expr\ \textbf{then}\ matchedStmt\ \textbf{else}\ stmt \\ & | & \textbf{other} \end{array}$$

Покажите, что эта грамматика остается неоднозначной.

4.4 Нисходящий синтаксический анализ

Нисходящий синтаксический анализ можно рассматривать как задачу построения дерева разбора для входной строки, начиная с корня и создавая узлы дерева разбора в прямом порядке обхода (обход в глубину, рассматривавшийся в разделе 2.3.4). Или, что то же самое, нисходящий синтаксический анализ можно рассматривать как поиск левого порождения входной строки.

Пример 4.14. На рис. 4.12 приведена последовательность деревьев разбора для входной строки **id** + **id** * **id**, представляющая собой нисходящий синтаксический анализ в соответствии с грамматикой (4.2), повторенной далее:

$$\begin{array}{lcl} E & \rightarrow & T\ E' \\ E' & \rightarrow & +\ T\ E' \mid \epsilon \\ T & \rightarrow & F\ T' \\ T' & \rightarrow & *\ F\ T' \mid \epsilon \\ F & \rightarrow & (E) \mid \textbf{id} \end{array} \qquad (4.14)$$

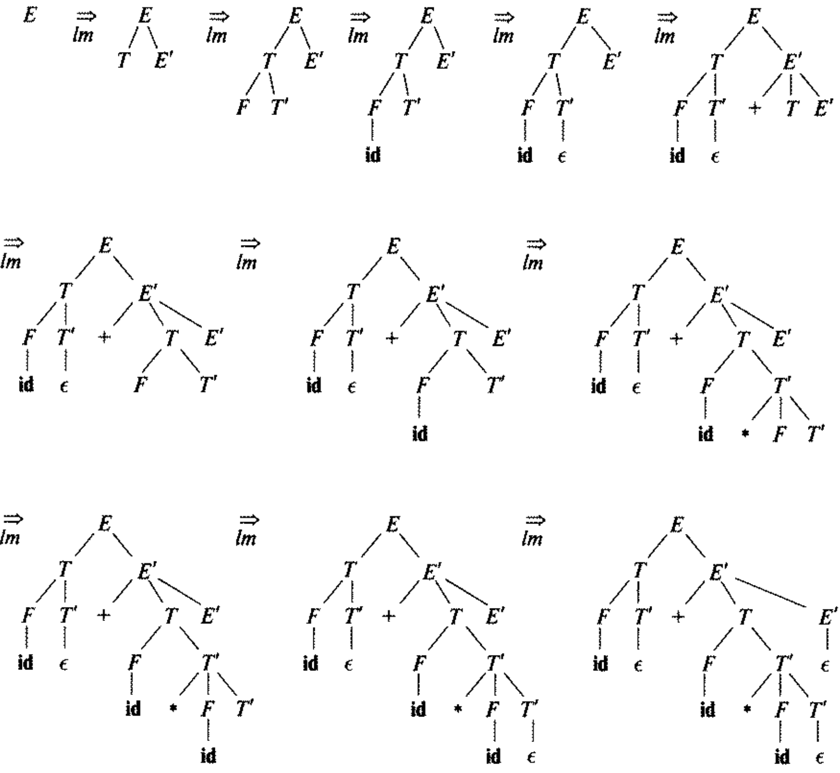


Рис. 4.12. Нисходящий синтаксический анализ для входной строки **id**+**id*****id**

Эта последовательность деревьев соответствует левому порождению входной строки. □

На каждом шаге нисходящего синтаксического анализа ключевой проблемой является определение продукции, применимой для нетерминала, скажем, A. Когда A-продукция выбрана, остальная часть процесса синтаксического анализа состоит из проверки "соответствий" терминальных символов в теле продукции входной строке.

Этот раздел начинается с общего вида нисходящего разбора, называющегося синтаксическим анализом методом рекурсивного спуска, который может потребовать возврата (отката — backtracking) для поиска корректной A-продукции, которая должна быть применена. В разделе 2.4.2 рассказывалось о предиктивном синтаксическом анализе — частном случае синтаксического анализа методом рекурсивного спуска, не требующем возврата. Предиктивный синтаксический анализ выбирает корректную A-продукцию путем предпросмотра фиксированного количества символов входной строки; типичной является ситуация, когда достаточно просмотреть только один (очередной) входной символ.

Рассмотрим, например, нисходящий синтаксический анализ на рис. 4.12, который строит дерево с двумя узлами, помеченными E'. В первом (в прямом порядке обхода) узле E' выбирается продукция $E' \to +T\ E'$; во втором узле E' выбирается продукция $E' \to \epsilon$. Предиктивный синтаксический анализатор может выбрать нужную E'-продукцию, просматривая очередной входной символ.

Класс грамматик, для которых можно построить предиктивный синтаксический анализатор, просматривающий k символов во входном потоке, иногда называется классом LL(k). Класс LL(1) будет рассматриваться в разделе 4.4.3, но некоторые необходимые при рассмотрении вычисления, FIRST и FOLLOW, будут рассмотрены в разделе 4.4.2. Из множеств FIRST и FOLLOW грамматики можно построить "таблицы предиктивного анализа", которые делают явным выбор продукции при нисходящем синтаксическом анализе. Эти таблицы применяются также и при восходящем синтаксическом анализе.

В разделе 4.4.4 будет приведен нерекурсивный алгоритм синтаксического анализа, использующий стек явно, а не посредством рекурсивных вызовов. Наконец, в разделе 4.4.5 будет рассмотрен вопрос восстановления после ошибок в процессе нисходящего разбора.

4.4.1 Синтаксический анализ методом рекурсивного спуска

Программа синтаксического анализа методом рекурсивного спуска (recursive-descent parsing) состоит из набора процедур, по одной для каждого нетерминала. Работа программы начинается с вызова процедуры для стартового символа и успешно заканчивается в случае сканирования всей входной строки. Псевдокод для типичного нетерминала показан на рис. 4.13. Обратите внимание на то, что этот псевдокод недетерминированный, поскольку он начинается с выбора A-продукции для применения не указанным способом.

```
   void A() {
1)     Выбираем A-продукцию A → X1X2...Xk;
2)     for ( i от 1 до k ) {
3)         if ( Xi – нетерминал )
4)             Вызов процедуры Xi ();
5)         else if ( Xi равно текущему входному символу a )
6)             Переходим к следующему входному символу;
7)         else /* Обнаружена ошибка */;
       }
   }
```

Рис. 4.13. Типичная процедура для нетерминала в нисходящем анализаторе

Рекурсивный спуск в общем случае может потребовать выполнения возврата, т.е. повторения сканирования входного потока. Однако при анализе синтаксических конструкций языков программирования возврат требуется редко, так что встреча с синтаксическим анализатором с возвратом — явление не частое. Даже в ситуациях наподобие синтаксического анализа естественного языка возврат не слишком эффективен, и предпочтительными являются табличные методы наподобие динамического программирования из упражнения 4.4.9 или метода Эрли (Earley) (см. список литературы к главе 4).

Чтобы разрешить возврат, код на рис. 4.13 должен быть немного модифицирован. Во-первых, невозможно выбрать единственную A-продукцию в строке 1, так что требуется испытывать каждую из нескольких продукций в некотором порядке. Во-вторых, ошибка в строке 7 не является окончательной и предполагает возврат к строке 1 и испытание другой A-продукции. Объявлять о найденной во входной строке ошибке можно только в том случае, если больше не имеется непроверенных A-продукций. Чтобы быть в состоянии проверить новую A-продукцию, нужно иметь возможность сбросить указатель входного потока в состояние, в котором он находился при первом достижении строки 1. Таким образом, для хранения этого указателя входного потока требуется локальная переменная.

Пример 4.15. Рассмотрим грамматику

$$\begin{array}{lcl} S & \rightarrow & c\,A\,d \\ A & \rightarrow & a\,b \mid a \end{array}$$

Чтобы построить дерево разбора для входной строки $w = cad$, начнем с дерева, состоящего из единственного узла с меткой S, и указателя входного потока, указывающего на c, первый символ w. S имеет единственную продукцию, так что мы используем ее для разворачивания S и получения дерева, показанного на рис. 4.14, *а*. Крайний слева лист, помеченный c, соответствует первому символу входного потока w, так что мы перемещаем указатель входного потока к a, второму символу w, и рассматриваем следующий лист, помеченный A.

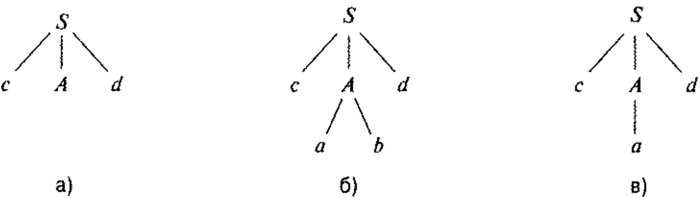


Рис. 4.14. Шаги нисходящего синтаксического анализа

Теперь мы разворачиваем A с использованием первой альтернативы, $A \rightarrow a\,b$, и получаем таким образом дерево, показанное на рис. 4.14, *б*. У нас имеется совпадение второго входного символа, a, так что мы переходим к третьему символу,

d, и сравниваем его с очередным листом b. Поскольку b не соответствует d, мы сообщаем об ошибке и возвращаемся к A, чтобы выяснить, нет ли альтернативной продукции, которая не была проверена до этого момента.

Вернувшись к A, мы должны сбросить указатель входного потока так, чтобы он указывал на позицию 2, в которой мы находились, когда впервые столкнулись с A. Это означает, что процедура для A должна хранить указатель на входной поток в локальной переменной.

Вторая альтернатива для A дает дерево разбора, показанное на рис. 4.14, *в*. Лист a соответствует второму символу w, а лист d — третьему символу. Поскольку нами построено дерево разбора для входной строки w, мы завершаем работу и сообщаем об успешном выполнении синтаксического анализа и построении дерева разбора. □

Леворекурсивная грамматика может привести синтаксический анализатор, работающий методом рекурсивного спуска, к бесконечному циклу, т.е. когда мы попытаемся развернуть нетерминал A, то в конечном счете можем найти этот же нетерминал и прийти к попытке развернуть A, так и не взяв ни одного символа из входного потока.

4.4.2 FIRST и FOLLOW

При построении как нисходящего, так и восходящего синтаксического анализатора нам помогут две функции — FIRST и FOLLOW, — связанные с грамматикой G. В процессе нисходящего синтаксического анализа FIRST и FOLLOW позволяют выбрать применяемую продукцию на основании очередного символа входного потока. Множества токенов, порождаемые функцией FOLLOW, могут также использоваться как синхронизирующие токены в процессе восстановления после ошибки в "режиме паники".

Определим $\text{FIRST}(\alpha)$, где α — произвольная строка символов грамматики, как множество терминалов, с которых начинаются строки, порождаемые α. Если $\alpha \stackrel{*}{\Rightarrow} \epsilon$, то $\epsilon \in \text{FIRST}(\alpha)$. Например, на рис. 4.15 $A \stackrel{*}{\Rightarrow} c\gamma$, так что $c \in \text{FIRST}(A)$.

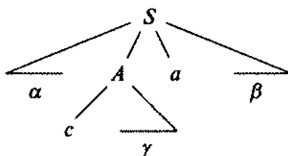


Рис. 4.15. Терминал $c \in \text{FIRST}(A)$,
а терминал $a \in \text{FOLLOW}(A)$

Чтобы понять, как FIRST может использоваться в процессе предиктивного синтаксического анализа, рассмотрим две A-продукции $A \to \alpha \mid \beta$, где $\text{FIRST}(\alpha)$

и FIRST (β) представляют собой непересекающиеся множества. Выбрать необходимую A-продукцию можно путем просмотра очередного входного символа a, поскольку a может находиться не более чем в одном из множеств FIRST (α) и FIRST (β), но ни в коем случае не в обоих одновременно. Например, если $a \in$ FIRST (β), то следует выбрать продукцию $A \to \beta$. Эта идея будет использоваться в разделе 4.4.3 для LL(1)-грамматик.

Определим FOLLOW (A) для нетерминала A как множество терминалов a, которые могут располагаться непосредственно справа от A в некоторой сентенциальной форме, т.е. множество терминалов a, таких, что существует порождение вида $S \stackrel{*}{\Rightarrow} \alpha A a \beta$ для некоторых α и β, как на рис. 4.15. Заметим, что в процессе порождения между A и a могут появиться символы, но если это так, то они порождают ϵ и исчезают. Кроме того, если A может оказаться крайним справа символом некоторой сентенциальной формы, то $\$ \in$ FOLLOW (A) (вспомните, что $ — это специальный символ "маркера конца", который не является символом грамматики).

Чтобы вычислить FIRST (X) для всех символов грамматики X, будем применять следующие правила до тех пор, пока ни к одному из множеств FIRST не смогут быть добавлены ни терминалы, ни ϵ.

1. Если X — терминал, то FIRST $(X) = \{X\}$.

2. Если X — нетерминал и имеется продукция $X \to Y_1 Y_2 \ldots Y_k$ для некоторого $k \geqslant 1$, то поместим a в FIRST (X), если для некоторого i $a \in$ FIRST (Y_i) и ϵ входит во все множества FIRST (Y_1), $\ldots$, FIRST (Y_{i-1}), т.е. $Y_1 \cdots Y_{i-1} \stackrel{*}{\Rightarrow} \epsilon$. Если ϵ входит в FIRST (Y_j) для всех $j = 1, 2, \ldots, k$, то добавляем ϵ к FIRST (X). Например, все, что находится в множестве FIRST (Y_1), есть и в множестве FIRST (X). Если Y_1 не порождает ϵ, то больше мы ничего не добавляем к FIRST (X), но если $Y_1 \stackrel{*}{\Rightarrow} \epsilon$, то к FIRST (X) добавляется FIRST (Y_2) и т.д.

3. Если имеется продукция $X \to \epsilon$, добавим ϵ к FIRST (X).

Теперь можно вычислить FIRST для любой строки $X_1 X_2 \ldots X_n$ следующим образом. Добавим к FIRST $(X_1 X_2 \ldots X_n)$ все не-ϵ-символы из FIRST (X_1). Добавим также все не-ϵ-символы из FIRST (X_2), если $\epsilon \in$ FIRST (X_1), все не-ϵ-символы из FIRST (Y_3), если ϵ имеется как в FIRST (X_1), так и в FIRST (X_2), и т.д. И наконец, добавим ϵ к FIRST $(X_1 X_2 \ldots X_n)$, если для всех i FIRST (X_i) содержит ϵ.

Чтобы вычислить FOLLOW (A) для всех нетерминалов A, будем применять следующие правила до тех пор, пока ни к одному множеству FOLLOW нельзя будет добавить ни одного символа.

1. Поместим $ в FOLLOW (S), где S — стартовый символ, а $ — правый ограничитель входного потока.

2. Если имеется продукция $A \to \alpha B \beta$, то все элементы множества FIRST (β), кроме ϵ, помещаются в множество FOLLOW (B).

3. Если имеется продукция $A \to \alpha B$ или $A \to \alpha B \beta$, где FIRST (β) содержит ϵ, то все элементы из множества FOLLOW (A) помещаются в множество FOLLOW (B).

Пример 4.16. Вновь обратимся к нелеворекурсивной грамматике (4.14). В этом случае мы найдем следующие значения FIRST и FOLLOW.

1. FIRST (F) = FIRST (T) = FIRST (E) = $\{(, \textbf{id}\}$. Что бы понять, почему это так, заметим, что две продукции для F имеют тела, начинающиеся с указанных терминальных символов — **id** и левой скобки. T имеет только одну продукцию, тело которой начинается с F. Поскольку F не порождает ϵ, FIRST (T) должно быть тем же, что и FIRST (F). Те же рассуждения применимы и к FIRST (E).

2. FIRST (E') = $\{+, \epsilon\}$. Причина этого в том, что одна из двух продукций для E' имеет тело, начинающееся с терминала +, а тело второй — ϵ. Когда нетерминал порождает ϵ, мы помещаем ϵ в множество FIRST этого нетерминала.

3. FIRST (T') = $\{*, \epsilon\}$. Аргументация в данном случае аналогична аргументации для FIRST (E').

4. FOLLOW (E) = FOLLOW (E') = $\{), \$\}$. Поскольку E является стартовым символом, множество FOLLOW (E) должно содержать \$. Тело продукции (E) объясняет, почему в FOLLOW (E) входит правая скобка. В случае E' заметим, что этот нетерминал появляется только в конце тел E-продукций. Таким образом, множество FOLLOW (E') должно быть тем же, что и FOLLOW (E).

5. FOLLOW (T) = FOLLOW (T') = $\{+,), \$\}$. Заметим, что в продукциях грамматики за T всегда следует E'. Таким образом, все элементы множества FIRST (E'), кроме ϵ, должны находиться в FOLLOW (T). Это объясняет наличие символа +. Однако, поскольку FIRST (E') содержит ϵ (т.е. $E' \overset{*}{\Rightarrow} \epsilon$), а E' представляет собой целую строку, следующую за T в телах E-продукций, все, что находится в FOLLOW (E), должно также находиться и в FOLLOW (T). Это объясняет наличие символа \$ и правой скобки. Что касается T', то, поскольку этот нетерминал появляется только в конце T-продукций, должно выполняться равенство FOLLOW (T') = FOLLOW (T).

6. FOLLOW (F) = $\{+, *,), \$\}$. Аргументация аналогична аргументации для T в п. 5. □

4.4.3 LL(1)-грамматики

Предиктивные синтаксические анализаторы, т.е. синтаксические анализаторы, работающие методом рекурсивного спуска без возврата, могут быть построены для класса грамматик, называющегося LL(1). Первое "L" в LL(1) означает сканирование входного потока слева направо, второе "L" — получение левого порождения, а "1" — использование на каждом шаге предпросмотра одного символа для принятия решения о действиях синтаксического анализатора.

Класс грамматик LL(1) достаточно богат для того, чтобы охватить большинство программных конструкций, хотя при написании грамматики для исходного языка требуется аккуратность. Например, в LL(1)-грамматике не может быть ни левой рекурсии, ни неоднозначности.

Грамматика G принадлежит классу LL(1) тогда и только тогда, когда для любых двух различных продукций G $A \to \alpha \mid \beta$ выполняются следующие условия.

1. Не существует такого терминала a, для которого и α, и β порождают строку, начинающуюся с a.
2. Пустую строку может порождать не более чем одна из продукций α или β.
3. Если $\beta \overset{*}{\Rightarrow} \epsilon$, то α не порождает ни одну строку, начинающуюся с терминала из FOLLOW(A). Аналогично, если $\alpha \overset{*}{\Rightarrow} \epsilon$, то β не порождает ни одну строку, начинающуюся с терминала из FOLLOW(A).

Первые два условия эквивалентны утверждению, что FIRST(α) и FIRST(β) представляют собой непересекающиеся множества. Третье условие эквивалентно утверждению, что если $\epsilon \in$ FIRST(β), то FIRST(α) и FOLLOW(A) — непересекающиеся множества; аналогичное утверждение справедливо и в случае, если $\epsilon \in$ FIRST(α).

Для LL(1)-грамматик могут быть построены предиктивные синтаксические анализаторы, поскольку корректная продукция для применения к нетерминалу может быть выбрана путем просмотра только текущего входного символа. Конструкции управления потоком с их определяющими ключевыми словами обычно удовлетворяют ограничениям LL(1). Например, пусть имеются продукции

$$\begin{array}{lcl} \mathit{stmt} & \to & \mathbf{if}\ (\mathit{expr})\ \mathit{stmt}\ \mathbf{else}\ \mathit{stmt} \\ & | & \mathbf{while}\ (\mathit{expr})\ \mathit{stmt} \\ & | & \{\mathit{stmt_list}\} \end{array}$$

Тогда ключевые слова **if**, **while** и символ { говорят, какая из альтернатив должна быть выбрана, когда мы встречаемся с инструкцией.

Приведенный далее алгоритм собирает информацию из множеств FIRST и FOLLOW в таблицу предиктивного синтаксического анализа $M[A, a]$ (представляющую собой двумерный массив), где A — нетерминал, а a — терминал или

Диаграммы переходов для предиктивных синтаксических анализаторов

Диаграммы переходов полезны для визуализации предиктивных синтаксических анализаторов. Например, на рис. 4.16, *а* показаны диаграммы для нетерминалов E и E' грамматики (4.14). Для построения диаграммы переходов на основе грамматики сначала требуется удалить из нее левую рекурсию, а затем выполнить левую факторизацию грамматики. Затем необходимо для каждого нетерминала A

1) создать начальное и конечное состояния;

2) для каждой продукции $A \to X_1 X_2 \ldots X_k$ создать путь из начального в конечное состояние с дугами, помеченными $X_1, X_2, \ldots, X_k$; если $A \to \epsilon$, путь представляет собой ребро, помеченное ϵ.

Диаграммы переходов для предиктивных синтаксических анализаторов отличаются от диаграмм переходов для лексических анализаторов. Синтаксические анализаторы имеют по диаграмме для каждого нетерминала. Метки ребер могут быть токенами или нетерминалами. Переход для токена (или терминала) означает, что этот переход будет выполнен, если этот токен будет очередным входным символом. Переход для нетерминала A представляет собой вызов процедуры для A.

В случае LL(1)-грамматики неоднозначность, связанная с решением о том, использовать ли ϵ-переход, может быть разрешена, если считать ϵ-переход выбором по умолчанию.

Диаграммы переходов могут быть упрощены при сохранении грамматических символов вдоль путей. Можно также вместо дуг для нетерминалов подставить диаграммы переходов для этих нетерминалов. Диаграммы на рис. 4.16, *а* и *б* эквивалентны: если мы проследим путь от E до принимающего состояния и выполним подстановку для E', то заметим, что в обоих множествах диаграмм символы вдоль путей образуют строки вида $T + T + \cdots + T$. Диаграмма на рис. 4.16, *б* может быть получена из диаграммы *а* путем преобразований, подобных описанным в разделе 2.5.4, где использовались устранение оконечной рекурсии и подстановка тел процедур для оптимизации процедуры нетерминала.

символ \$ (маркер конца входного потока). Алгоритм основан на следующей идее: если очередной входной символ a находится в множестве FIRST (α), выбирается продукция $A \to \alpha$. Единственная сложность возникает при $\alpha = \epsilon$ или, в общем

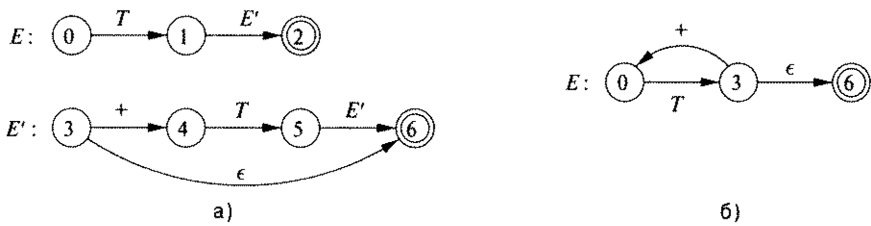


Рис. 4.16. Диаграммы переходов для нетерминалов E и E' грамматики (4.14)

случае, когда $\alpha \stackrel{*}{\Rightarrow} \epsilon$. В этом случае мы снова должны выбрать $A \to \alpha$, если текущий входной символ имеется в FOLLOW (A) или если из входного потока получен \$, который при этом входит в FOLLOW (A).

Алгоритм 4.17. Построение таблицы предиктивного синтаксического анализа

Вход: грамматика G.

Выход: таблица синтаксического анализа M.

Метод: для каждой продукции грамматики $A \to \alpha$ выполняем следующие действия.

1. Для каждого терминала a из FIRST (α) добавляем $A \to \alpha$ в ячейку $M[A, a]$.
2. Если $\epsilon \in$ FIRST (α), то для каждого терминала b из FOLLOW (A) добавляем $A \to \alpha$ в $M[A, b]$. Если $\epsilon \in$ FIRST (α) и $\$ \in$ FOLLOW (A), то добавляем $A \to \alpha$ также и в $M[A, \$]$.

Если после выполнения этих действий ячейка $M[A, a]$ осталась без продукции, устанавливаем ее значение равным **error** (это значение обычно представляется пустой записью таблицы). □

Пример 4.18. Для грамматики выражений (4.14) алгоритм 4.17 дает таблицу синтаксического анализа, приведенную на рис. 4.17. Пустые места в таблице означают записи ошибок; непустые указывают продукции, при помощи которых выполняется разворачивание нетерминала.

Рассмотрим продукцию $E \to T\ E'$. Поскольку

$$\text{FIRST}\left(T\ E'\right) = \text{FIRST}\left(T\right) = \{(, \textbf{id}\},$$

эта продукция добавляется к $M[E, (]$ и $M[E, \textbf{id}]$. Продукция $E' \to +T\ E'$ добавляется к $M[E', +]$, поскольку FIRST $(+T\ E') = \{+\}$. Так как FOLLOW $(E') = \{), \$\}$, продукция $E' \to \epsilon$ добавляется к $M[E',)]$ и $M[E', \$]$. □

Алгоритм 4.17 может быть применен для получения таблицы M к любой грамматике G. Для любой LL(1)-грамматики каждая запись в таблице синтаксического

Нетерминал	Входной символ					
	id	+	*	(	)	$
E	$E \to T\,E'$			$E \to T\,E'$		
E'		$E' \to +T\,E'$			$E' \to \epsilon$	$E' \to \epsilon$
T	$T \to F\,T'$			$T \to F\,T'$		
T'		$T' \to \epsilon$	$T' \to *\,F\,T'$		$T' \to \epsilon$	$T' \to \epsilon$
F	$F \to$ **id**			$F \to (E)$		

Рис. 4.17. Таблица синтаксического анализа M к примеру 4.18

анализа единственным образом определяет продукцию либо сообщает об ошибке. Однако для некоторых грамматик таблица M может иметь записи с несколькими продукциями. Например, если G — леворекурсивная или неоднозначная грамматика, таблица M будет содержать как минимум одну запись с несколькими продукциями. Хотя устранение левой рекурсии и левая факторизация — легко выполняемые задачи, существуют такие грамматики, никакие изменения которых не приведут к LL(1)-грамматике.

Язык в следующем примере не является LL(1)-грамматикой.

Пример 4.19. Следующая грамматика, абстрагирующая проблему "висящего **else**", взята из примера 4.11:

$$\begin{array}{lcl} S & \to & i\,E\,t\,S\,S' \mid a \\ S' & \to & e\,S \mid \epsilon \\ E & \to & b \end{array}$$

Таблица синтаксического анализа для этой грамматики показана на рис. 4.18. Запись $M\,[S', e]$ в ней содержит как $S' \to e\,S$, так и $S' \to \epsilon$.

Нетерминал	Входной символ					
	a	b	e	i	t	$
S	$S \to a$			$S \to i\,E\,t\,S\,S'$		
S'			$S' \to \epsilon$ $S' \to e\,S$			$S' \to \epsilon$
E		$E \to b$				

Рис. 4.18. Таблица синтаксического анализа M к примеру 4.19

Грамматика неоднозначна, и эта неоднозначность проявляется в выборе используемой продукции при встрече во входном потоке e (**else**). Эту неоднознач-

ность можно разрешить путем выбора продукции $S' \to e\ S$. Данный выбор соответствует связыванию **else** с ближайшим предыдущим **then**. Заметим, что выбор $S' \to \epsilon$ приводит к тому, что e не помещается в стек и не удаляется из входного потока, что, очевидно, неверно. □

4.4.4 Нерекурсивный предиктивный синтаксический анализ

Нерекурсивный предиктивный синтаксический анализатор можно построить с помощью явного использования стека (вместо неявного при рекурсивных вызовах). Синтаксический анализатор имитирует левое порождение. Если w — входная строка, соответствие которой проверено до текущего момента, то в стеке хранится последовательность грамматических символов α, такая, что

$$S \underset{lm}{\overset{*}{\Rightarrow}} w\alpha$$

Синтаксический анализатор на рис. 4.19, управляемый таблицей синтаксического анализа, имеет входной буфер, стек, содержащий последовательность грамматических символов, таблицу синтаксического анализа, построенную при помощи алгоритма 4.17, и выходной поток. Входной буфер содержит анализируемую строку, за которой следует маркер конца строки $. Мы также используем символ $ для указания дна стека, который изначально содержит над символом $ стартовый символ грамматики.

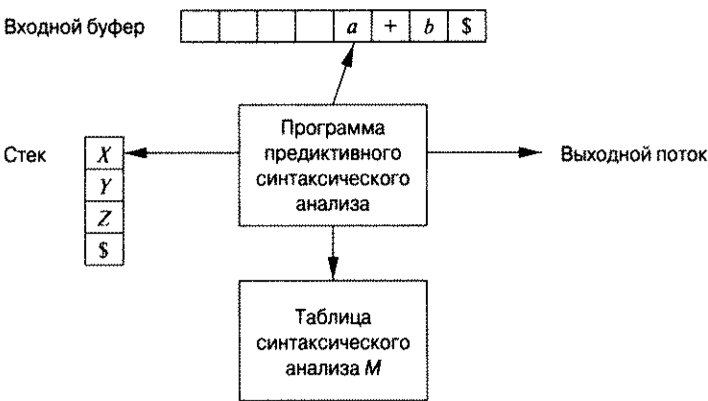


Рис. 4.19. Модель синтаксического анализатора, управляемого таблицей синтаксического анализа

Синтаксический анализатор управляется программой, которая рассматривает символ на вершине стека X и текущий входной символ a. Если X является нетерминалом, синтаксический анализатор выбирает X-продукцию в соответствии с записью $M[X, a]$ таблицы синтаксического анализа M. (Здесь может выполняться

дополнительный код, например, для построения узла дерева разбора.) В противном случае проверяется соответствие между терминалом X и текущим входным символом a.

Поведение синтаксического анализатора может быть описано в терминах его *конфигураций* (configuration), которые дают содержимое стека и оставшийся входной поток. Приведенный далее алгоритм описывает работу с конфигурациями.

Алгоритм 4.20. Предиктивный синтаксический анализ, управляемый таблицей

Вход: строка w и таблица синтаксического анализа M для грамматики G.

Выход: если $w \in L(G)$ — левое порождение w; в противном случае — сообщение об ошибке.

Метод: изначально синтаксический анализатор находится в конфигурации с $w\$$ во входном буфере и стартовым символом S грамматики G на вершине стека, над \$. Программа на рис. 4.20 использует таблицу предиктивного синтаксического анализа M для анализа входной строки. □

```
Устанавливаем указатель входного потока ip так,
чтобы он указывал на первый символ строки w;
Устанавливаем X равным символу на вершине стека;
while ( X ≠ $ ) { /* Стек не пуст */
      Устанавливаем a равным символу, на который
            в настоящий момент указывает ip
      if ( X равен a ) {
            Снимаем символ со стека и перемещаем ip
                  к следующему символу строки;
      }
      else if ( X – терминал ) error();
      else if ( M [X, a] – запись об ошибке ) error();
      else if ( M [X, a] = X → Y1Y2 ... Yk) {
            Выводим продукцию X → Y1Y2 ... Yk;
            Снимаем символ со стека;
            Помещаем в стек Yk, Yk-1, ..., Y1; Y1
                  помещается на вершину стека;
      }
      Устанавливаем X равным символу на вершине стека;
}
```

Рис. 4.20. Алгоритм предиктивного синтаксического анализа

Пример 4.21. Рассмотрим грамматику (4.14); ее таблица синтаксического анализа уже была приведена на рис. 4.17. Для входной строки **id**+**id*****id** нерекурсивный

предиктивный синтаксический анализатор алгоритма 4.20 делает последовательность шагов, представленную на рис. 4.21. Эти шаги соответствуют левому порождению

$$E \underset{lm}{\Rightarrow} T\ E' \underset{lm}{\Rightarrow} F\ T'\ E' \underset{lm}{\Rightarrow} \mathbf{id}\ T'\ E' \underset{lm}{\Rightarrow} \mathbf{id}\ E' \underset{lm}{\Rightarrow} \mathbf{id} + T\ E' \underset{lm}{\Rightarrow} \cdots$$

Полностью оно показано на рис. 4.12.

Соответствие	Стек	Входная строка	Действие
	$E\$$	$\mathbf{id} + \mathbf{id} * \mathbf{id}\$$	
	$T\ E'\$$	$\mathbf{id} + \mathbf{id} * \mathbf{id}\$$	Вывод $E \to T\ E'$
	$F\ T'\ E'\$$	$\mathbf{id} + \mathbf{id} * \mathbf{id}\$$	Вывод $T \to F\ T'$
	$\mathbf{id}\ T'\ E'\$$	$\mathbf{id} + \mathbf{id} * \mathbf{id}\$$	Вывод $F \to \mathbf{id}$
$\mathbf{id}$	$T'\ E'\$$	$+\mathbf{id} * \mathbf{id}\$$	Соответствие $\mathbf{id}$
$\mathbf{id}$	$E'\$$	$+\mathbf{id} * \mathbf{id}\$$	Вывод $T' \to \epsilon$
$\mathbf{id}$	$+\ T\ E'\$$	$+\mathbf{id} * \mathbf{id}\$$	Вывод $E' \to +\ T\ E'$
$\mathbf{id}+$	$T\ E'\$$	$\mathbf{id} * \mathbf{id}\$$	Соответствие +
$\mathbf{id}+$	$F\ T'\ E'\$$	$\mathbf{id} * \mathbf{id}\$$	Вывод $T \to F\ T'$
$\mathbf{id}+$	$\mathbf{id}\ T'\ E'\$$	$\mathbf{id} * \mathbf{id}\$$	Вывод $F \to \mathbf{id}$
$\mathbf{id} + \mathbf{id}$	$T'\ E'\$$	$*\mathbf{id}\$$	Соответствие $\mathbf{id}$
$\mathbf{id} + \mathbf{id}$	$*\ F\ T'\ E'\$$	$*\mathbf{id}\$$	Вывод $T' \to *\ F\ T'$
$\mathbf{id} + \mathbf{id}*$	$F\ T'\ E'\$$	$\mathbf{id}\$$	Соответствие *
$\mathbf{id} + \mathbf{id}*$	$\mathbf{id}\ T'\ E'\$$	$\mathbf{id}\$$	Вывод $F \to \mathbf{id}$
$\mathbf{id} + \mathbf{id} * \mathbf{id}$	$T'\ E'\$$	$\$$	Соответствие $\mathbf{id}$
$\mathbf{id} + \mathbf{id} * \mathbf{id}$	$E'\$$	$\$$	Вывод $T' \to \epsilon$
$\mathbf{id} + \mathbf{id} * \mathbf{id}$	$\$$	$\$$	Вывод $E' \to \epsilon$

Рис. 4.21. Шаги, выполняемые предиктивным синтаксическим анализатором для входной строки $\mathbf{id} + \mathbf{id} * \mathbf{id}$

Обратите внимание, что сентенциальные формы в этом порождении представляют собой части входного потока, которые уже были проверены на соответствие (приведены в столбце "Соответствие") и за которыми следует содержимое стека. Часть потока, проверенная на соответствие, показана исключительно для того, чтобы подчеркнуть указанную связь. По той же причине вершина стека располагается слева; при рассмотрении восходящего синтаксического анализа более естественно располагать вершину стека справа. Указатель входного потока указывает на крайний слева символ строки в столбце "Входная строка". □

4.4.5 Восстановление после ошибок в предиктивном синтаксическом анализе

Данное рассмотрение восстановления после ошибок ссылается на стек предиктивного синтаксического анализатора, управляемого таблицей анализа, поскольку он явно указывает терминалы и нетерминалы, соответствие которым синтаксический анализатор намеревается найти в оставшейся части входного потока. Эти методы могут использоваться и при синтаксическом анализе методом рекурсивного спуска.

Ошибка в процессе предиктивного синтаксического анализа обнаруживается в тот момент, когда терминал на вершине стека не соответствует очередному входному символу или когда на вершине стека находится нетерминал A, очередной входной символ — a, а ячейка таблицы синтаксического анализа $M[A, a]$ содержит **error** (т.е. данная запись таблицы пуста).

Режим паники

Восстановление после ошибки "в режиме паники" основано на пропуске символов из входного потока до тех пор, пока не будет обнаружен токен из предопределенного множества синхронизирующих токенов. Эффективность этого метода зависит от выбора синхронизирующего множества. Эти множества должны выбираться так, чтобы синтаксический анализатор быстро восстанавливался после часто встречающихся на практике ошибок. Вот некоторые эвристические правила.

1. В качестве первого приближения можно поместить в синхронизирующее множество для нетерминала A все символы из множества $\text{FOLLOW}(A)$. Если пропустить все токены до элемента из $\text{FOLLOW}(A)$ и снять A со стека, вероятно, удастся продолжить синтаксический анализ.

2. В качестве синхронизирующего множества для A недостаточно использовать $\text{FOLLOW}(A)$. Например, если инструкция завершается точкой с запятой, как в языке программирования C, то ключевое слово, с которого начинается инструкция, может не оказаться в множестве FOLLOW нетерминалов, представляющих выражения. Отсутствующая точка с запятой после присвоения может, таким образом, привести к пропуску ключевого слова, с которого начинается новая инструкция. Зачастую в языке имеется иерархическая структура конструкций; например, выражения появляются в инструкциях, которые находятся в блоках, и т.д. В таком случае к синхронизирующему множеству конструкции нижнего уровня можно добавить символы, с которых начинаются конструкции более высокого уровня. Например, можно добавить ключевые слова, с которых начинаются инструкции, в синхронизирующие множества нетерминалов, порождающих выражения.

3. Если добавить символы из FIRST (A) в синхронизирующее множество для нетерминала A, станет возможным продолжение анализа в соответствии с A, когда во входном потоке появится символ из множества FIRST (A).

4. Если нетерминал может порождать пустую строку, то в качестве продукции по умолчанию может использоваться продукция, порождающая ϵ. Это может отсрочить обнаружение ошибки, зато не может вызвать ее пропуск и сокращает число нетерминалов, которые должны быть рассмотрены в процессе восстановления после ошибки.

5. Если терминал на вершине стека не может быть сопоставлен с входным символом, то простейший метод состоит в снятии терминала со стека, генерации сообщения о вставке терминала в программу и продолжении синтаксического анализа. По сути, при этом подходе синхронизирующее множество токена состоит из всех остальных токенов.

Пример 4.22. Использование символов из множеств FOLLOW и FIRST в качестве синхронизирующих достаточно неплохо работает при синтаксическом анализе грамматики (4.14). Таблица синтаксического анализа для этой грамматики, приведенная на рис. 4.17, повторена на рис. 4.22, но здесь значение *synch* в ячейках таблицы указывает синхронизирующие токены, полученные из множества FOLLOW соответствующего нетерминала. Множества FOLLOW для нетерминалов получены из примера 4.16.

Нетерминал	Входной символ					
	id	+	*	(	)	$
E	$E \to T\,E'$			$E \to T\,E'$	*synch*	*synch*
E'		$E' \to +\,T\,E'$			$E' \to \epsilon$	$E' \to \epsilon$
T	$T \to F\,T'$	*synch*		$T \to F\,T'$	*synch*	*synch*
T'		$T' \to \epsilon$	$T' \to *\,F\,T'$		$T' \to \epsilon$	$T' \to \epsilon$
F	$F \to \mathbf{id}$	*synch*	*synch*	$F \to (E)$	*synch*	*synch*

Рис. 4.22. Синхронизирующие токены, добавленные к таблице синтаксического анализа из рис. 4.17

Рис. 4.22 используется следующим образом. Если синтаксический анализатор просматривает ячейку $M[A, a]$ и обнаруживает, что она пуста, то входной символ a пропускается. Если в этой ячейке находится запись *synch*, то нетерминал снимается с вершины стека в попытке продолжить синтаксический анализ. Если токен на вершине стека не соответствует входному символу, то, как упоминалось выше, мы снимаем его со стека.

В случае ошибочного ввода)**id** $*$ +**id** синтаксический анализатор и механизм восстановления после ошибок, представленные на рис. 4.22, ведут себя так, как показано на рис. 4.23. □

Стек	Входная строка	Примечание
$E\$$	)**id** $*$ +**id**$	Ошибка, пропускаем)
$E\$$	**id** $*$ +**id**$	**id** $\in$ FIRST (E)
$T\ E'\$$	**id** $*$ +**id**$	
$F\ T'\ E'\$$	**id** $*$ +**id**$	
id $T'\ E'\$$	**id** $*$ +**id**$	
$T'\ E'\$$	$*$ + **id**$	
$*\ F\ T'\ E'\$$	$*$ + **id**$	
$F\ T'\ E'\$$	+**id**$	Ошибка, $M\ [F, +] = synch$
$T'\ E'\$$	+**id**$	F снимается со стека
$E'\$$	+**id**$	
$+\ T\ E'\$$	+**id**$	
$T\ E'\$$	**id**$	
$F\ T'\ E'\$$	**id**$	
id $T'\ E'\$$	**id**$	
$T'\ E'\$$	$	
$E'\$$	$	
$	$	

Рис. 4.23. Шаги синтаксического анализа и восстановления после ошибок, выполняемые предиктивным анализатором

В приведенном обсуждении метода восстановления после ошибки "в режиме паники" мы не касались важного вопроса — сообщений об ошибках. Разработчик компилятора должен обеспечить вывод информативных сообщений об обнаруженных ошибках, которые не только описывают ошибку, но и указывают, где именно она обнаружена.

Восстановление на уровне фразы

Восстановление на уровне фразы реализуется заполнением пустых ячеек в таблице предиктивного синтаксического анализа указателями на подпрограммы обработки ошибок. Эти подпрограммы могут изменять, вставлять или удалять символы входного потока и выводить соответствующие сообщения об ошибках. Кроме того, они могут снимать элементы со стека. При таком способе восстановления

по ряду причин возникают вопросы, следует ли разрешить изменение символов в стеке и размещение в стеке новых символов. Во-первых, после этого шаги, выполняемые синтаксическим анализатором, могут не соответствовать порождению ни одного слова языка вообще. Во-вторых, следует гарантировать невозможность бесконечного цикла. Проверка того, что каждое действие по восстановлению после ошибки в конечном счете приводит к потреблению очередного входного символа (или к снятию элементов со стека, если достигнут конец входного потока), — хороший способ защиты от зацикливания.

4.4.6 Упражнения к разделу 4.4

Упражнение 4.4.1. Разработайте для каждой из приведенных грамматик предиктивный синтаксический анализатор и покажите, какой вид имеют таблицы синтаксического анализа. Перед разработкой синтаксического анализатора при необходимости можно выполнить левую факторизацию и/или устранение левой рекурсии.

а) Грамматика из упражнения 4.2.2, *а*.

б) Грамматика из упражнения 4.2.2, *б*.

в) Грамматика из упражнения 4.2.2, *в*.

г) Грамматика из упражнения 4.2.2, *г*.

д) Грамматика из упражнения 4.2.2, *д*.

е) Грамматика из упражнения 4.2.2, *ж*.

!! **Упражнение 4.4.2.** Возможно ли путем некоторой модификации грамматики построить предиктивный синтаксический анализатор для языка из упражнения 4.2.1 (постфиксные выражения с операндом a)?

Упражнение 4.4.3. Вычислите FIRST и FOLLOW для каждой из грамматик из упражнения 4.2.1.

Упражнение 4.4.4. Вычислите FIRST и FOLLOW для каждой из грамматик из упражнения 4.2.2.

Упражнение 4.4.5. Грамматика $S \to a\ S\ a \mid a\ a$ генерирует все строки четной длины из символов a. Можно разработать для этой грамматики синтаксический анализатор, работающий методом рекурсивного спуска с возвратом. Если выбирать для разворачивания первой продукцию $S \to a\ a$, то будет распознаваться только одна строка — aa. Следовательно, корректный синтаксический анализатор, работающий методом рекурсивного спуска, должен первой испытывать продукцию $S \to a\ S\ a$.

а) Покажите, что такой синтаксический анализатор будет распознавать строки *aa*, *aaaa* и *aaaaaaaa*, но не *aaaaaa*.

б) Какой язык распознает данный синтаксический анализатор?

Следующие упражнения являются шагами представления произвольной грамматики в *нормальной форме Хомски* (Chomsky normal form — CNF), определенной в упражнении 4.4.8.

! **Упражнение 4.4.6.** Грамматика называется *ϵ-свободной*, если в ней отсутствуют продукции, тело которых представляет собой ϵ (называющиеся *ϵ-продукциями*).

а) Разработайте алгоритм для преобразования произвольной грамматики в ϵ-свободную грамматику, генерирующую тот же язык (с возможным исключением в виде пустой строки — ни одна ϵ-свободная грамматика не может генерировать ϵ). *Указание*: сначала найдите все нетерминалы, которые в состоянии генерировать ϵ, пусть даже путем длинного порождения.

б) Примените ваш алгоритм к грамматике $S \to a\ S\ b\ S \mid b\ S\ a\ S \mid \epsilon$.

! **Упражнение 4.4.7.** *Единичной* (single) продукцией называется продукция, тело которой состоит из единственного нетерминала, т.е. продукция вида $A \to B$.

а) Разработайте алгоритм для преобразования произвольной грамматики в ϵ-свободную грамматику без единичных продукций, генерирующую тот же язык (с возможным исключением в виде пустой строки). *Указание*: сначала удалите ϵ-продукции, а затем найдите, для каких пар терминалов A и B выполняется соотношение $A \overset{*}{\Rightarrow} B$ путем последовательности единичных продукций.

б) Примените ваш алгоритм к грамматике (4.1) из раздела 4.1.2.

в) Покажите, что как следствие из п. *а* данного упражнения можно преобразовать грамматику в эквивалентную ей грамматику без циклов (порождений $A \overset{*}{\Rightarrow} A$ за один или несколько шагов для некоторого нетерминала A).

! **Упражнение 4.4.8.** Грамматика называется грамматикой в *нормальной форме Хомски* (Chomsky normal form — CNF), если все ее продукции имеют вид $A \to BC$ или $A \to a$, где A, B и C — нетерминалы, а a — терминал. Покажите, как преобразовать любую грамматику в грамматику в нормальной форме Хомски для того же языка (с возможным исключением в виде пустой строки — ни одна CNF-грамматика не может генерировать ϵ).

! **Упражнение 4.4.9.** Любой язык с контекстно-свободной грамматикой может быть распознан за время $O\left(n^3\right)$ для строки длиной n. Простой способ сделать это —

применить алгоритм Кока–Янгера–Касами (Cocke–Younger–Kasami — CYK), основанный на динамическом программировании. Для данной строки $a_1a_2 \ldots a_n$ строится таблица T размером $n \times n$, такая, что T_{ij} представляет собой множество нетерминалов, генерирующих подстроку $a_ia_{i+1} \ldots a_j$. Если в основе языка лежит CNF-грамматика (см. упражнение 4.4.8), то одна ячейка таблицы может быть заполнена за время $O(n)$, если заполнять ячейки в правильном порядке, начиная с наименьших значений $j - i$. Разработайте алгоритм, который корректно заполняет ячейки указанной таблицы, и покажите, что он имеет время работы $O(n^3)$. Если у вас имеется заполненная таблица, как в этом случае определить, принадлежит ли строка $a_1a_2 \ldots a_n$ языку?

! **Упражнение 4.4.10.** Покажите, как при наличии заполненной таблицы из упражнения 4.4.9 можно восстановить дерево разбора для $a_1a_2 \ldots a_n$ за время $O(n)$. *Указание*: модифицируйте таблицу таким образом, чтобы для каждого нетерминала A в каждой ячейке T_{ij} она записывала некоторую пару нетерминалов в других ячейках таблицы, которые обосновывают помещение A в T_{ij}.

! **Упражнение 4.4.11.** Модифицируйте ваш алгоритм из упражнения 4.4.9 таким образом, чтобы для любой строки он находил наименьшее количество вставок, удалений и изменений отдельных символов, необходимых для превращения указанной строки в корректную строку грамматики.

! **Упражнение 4.4.12.** На рис. 4.24 представлена грамматика для некоторых инструкций. В ней e и s можно рассматривать как терминалы, обозначающие соответственно условные выражения и "прочие инструкции". Если разрешить конфликт, связанный с разворачиванием необязательного **else** (нетерминал *stmtTail*), путем обязательной выборки из входного потока имеющегося там **else**, то для приведенной грамматики можно построить предиктивный синтаксический анализатор. Воспользуйтесь идеей синхронизирующих символов из раздела 4.4.5 и выполните следующее.

а) Постройте для данной грамматики таблицу исправляющего ошибки предиктивного синтаксического анализа.

б) Покажите, как поведет себя ваш синтаксический анализатор для следующих входных строк:

i) **if** e **then** s ; **if** e **then** s **end**

ii) **while** e **do begin** s ; **if** e **then** s ; **end**

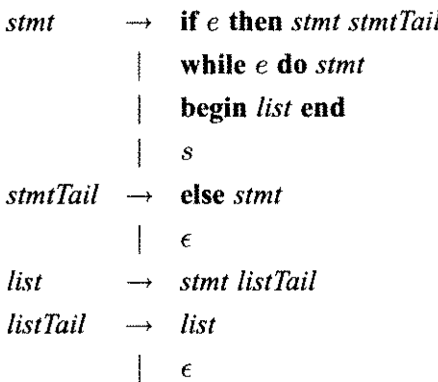


Рис. 4.24. Грамматика для некоторых инструкций

4.5 Восходящий синтаксический анализ

Восходящий синтаксический анализ соответствует построению дерева разбора для входной строки, начиная с листьев (снизу) и идя по направлению к корню (вверх). Удобно описывать синтаксический анализ как процесс построения дерева разбора, хотя начальная стадия компиляции может в действительности быть выполнена и без явного построения дерева. Последовательность "снимков" деревьев разбора на рис. 4.25 иллюстрирует восходящий синтаксический анализ для потока токенов **id** $*$ **id** в соответствии с грамматикой выражений (4.1).

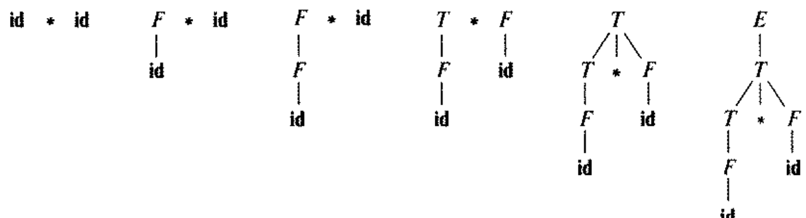


Рис. 4.25. Восходящий синтаксический анализ для строки **id** $*$ **id**

В этом разделе будет рассмотрен общий вид восходящего синтаксического анализа, известный как синтаксический анализ типа "перенос/свертка" (shift-reduce). В разделах 4.6 и 4.7 будет рассмотрен большой класс грамматик, для которых могут быть построены синтаксические анализаторы, работающие по принципу переноса/свертки — LR-грамматики. Хотя построение LR-анализатора вручную — задача очень трудоемкая, автоматические генераторы синтаксических анализаторов делают создание эффективных LR-анализаторов для соответствующих грамматик достаточно простым занятием. Концепции, рассматриваемые в данном разделе, полезны при написании грамматик для эффективного использования генератора

LR-анализаторов. Алгоритм для реализации генераторов синтаксических анализаторов приводится в разделе 4.7.

4.5.1 Свертки

Можно рассматривать восходящий синтаксический анализ как процесс "свертки" строки w к стартовому символу грамматики. На каждом шаге *свертки* (reduction) определенная подстрока, соответствующая телу продукции, заменяется нетерминалом из заголовка этой продукции.

Ключевые решения, принимаемые в процессе восходящего синтаксического анализа, — когда выполнять свертку и какую продукцию применять.

Пример 4.23. Снимки на рис. 4.25 иллюстрируют последовательность сверток; используемая грамматика — грамматика выражений (4.1). Свертки будут рассматриваться в терминах последовательности строк

$$\mathbf{id} * \mathbf{id}, F * \mathbf{id}, T * \mathbf{id}, T * F, T, E$$

Строки этой последовательности образованы корнями всех поддеревьев на снимке. Последовательность начинается со входной строки $\mathbf{id} * \mathbf{id}$. Первая свертка дает $F * \mathbf{id}$ путем свертывания крайнего слева $\mathbf{id}$ в F с использованием продукции $F \to \mathbf{id}$. Вторая свертка дает $T * \mathbf{id}$ при помощи свертывания F в T.

После этого у нас есть выбор между сверткой строки T, которая является телом продукции $E \to T$, и строки, состоящей из следующего $\mathbf{id}$, являющегося телом продукции $F \to \mathbf{id}$. Вместо того, чтобы выполнять свертку T в E, свернем второй $\mathbf{id}$ в F, получая строку $T * F$. Затем эта строка сворачивается в T. Синтаксический анализ завершается сверткой T в стартовый символ E. □

По определению свертка представляет собой шаг, обратный порождению (вспомните, что в порождении нетерминал в сентенциальной форме замещается телом одной из его продукций). Цель восходящего синтаксического анализа, таким образом, состоит в построении порождения в обратном порядке. Вот порождение, соответствующее синтаксическому анализу, показанному на рис. 4.25:

$$E \Rightarrow T \Rightarrow T * F \Rightarrow T * \mathbf{id} \Rightarrow F * \mathbf{id} \Rightarrow \mathbf{id} * \mathbf{id}$$

Данное порождение является правым.

4.5.2 Обрезка основ

Восходящий синтаксический анализ в процессе сканирования входного потока слева направо строит правое порождение в обратном порядке. Неформально говоря, основа, или дескриптор (handle), строки — это подстрока, которая соответствует телу продукции и свертка которой представляет собой один шаг правого порождения в обратном порядке.

Например, основы в процессе синтаксического анализа $\mathbf{id}_1 * \mathbf{id}_2$ (нижние индексы добавлены к токенам **id** для ясности) в соответствии с грамматикой (4.1) показаны на рис. 4.26. Хотя T — тело продукции $E \to T$, символ T не является основой в сентенциальной форме $T * \mathbf{id}_2$. Если заменить T на E, мы получим строку $E * \mathbf{id}_2$, которая не может быть порождена из стартового символа E. Таким образом, крайняя слева подстрока, которая соответствует телу некоторой продукции, не обязательно является основой.

Правая сентенциальная форма	Основа	Сворачивающая продукция
$\mathbf{id}_1 * \mathbf{id}_2$	$\mathbf{id}_1$	$F \to \mathbf{id}$
$F * \mathbf{id}_2$	F	$T \to F$
$T * \mathbf{id}_2$	$\mathbf{id}_2$	$F \to \mathbf{id}$
$T * F$	$T * F$	$E \to T * F$

Рис. 4.26. Основы в процессе синтаксического анализа строки $\mathbf{id}_1 * \mathbf{id}_2$

Формально, если $S \overset{*}{\underset{rm}{\Rightarrow}} \alpha A w \underset{rm}{\Rightarrow} \alpha\beta w$, как на рис. 4.27, то продукция $A \to \beta$ в позиции после α является *основой* (handle) $\alpha\beta w$. В качестве альтернативы основой правосентенциальной формы γ является продукция $A \to \beta$ и позиция γ, в которой может быть найдена строка β, такая, что замена β в данной позиции на A дает предшествующую правосентенциальную форму в правом порождении γ.

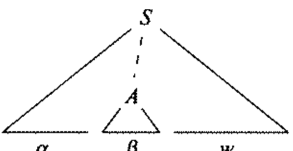


Рис. 4.27. Основа $A \to \beta$ в дереве разбора $\alpha\beta$w

Заметим, что строка w справа от основы должна содержать только терминальные символы. Для удобства мы будем говорить как об основе о теле продукции β, а не обо всей продукции $A \to \beta$ в целом. Следует также заметить, что грамматика может быть неоднозначной, с несколькими правыми порождениями $\alpha\beta w$. Если грамматика однозначна, то каждая правосентенциальная форма грамматики имеет ровно одну основу.

Обращенное правое порождение может быть получено посредством "обрезки основ". Мы начинаем процесс со строки терминаловw, которую хотим проанализировать. Если w — предложение рассматриваемой грамматики, то пусть $w = \gamma_n$, где γ_n — n-я правосентенциальная форма некоторого еще неизвестного правого порождения $S = \gamma_0 \underset{rm}{\Rightarrow} \gamma_1 \underset{rm}{\Rightarrow} \gamma_2 \underset{rm}{\Rightarrow} \cdots \underset{rm}{\Rightarrow} \gamma_{n-1} \underset{rm}{\Rightarrow} \gamma_n = w$. Для воссоздания этого

порождения в обратном порядке мы находим основу β_n в γ_n и заменяем ее левой частью продукции $A_n \to \beta_n$ для получения предыдущей правосентенциальной формы γ_{n-1}. Заметим, что пока мы не знаем, каким образом искать основы, но вскоре познакомимся с соответствующими методами.

Затем мы повторяем описанный процесс, т.е. находим в γ_{n-1} основу β_{n-1} и свертываем ее для получения правосентенциальной формы γ_{n-2}. Если после очередного шага правосентенциальная форма содержит только стартовый символ S, мы прекращаем процесс и сообщаем об успешном завершении анализа. Обращенная последовательность продукций, использованных в свертках, представляет собой правое порождение входной строки.

4.5.3 Синтаксический анализ "перенос/свертка"

Синтаксический анализ "перенос/свертка" (именуемый далее сокращенно ПС-анализом) представляет собой разновидность восходящего анализа, в которой для хранения символов грамматики используется стек, а для хранения остающейся непроанализированной части входной строки — входной буфер.

Мы используем символ \$ для маркирования дна стека и правого конца входной строки. При рассмотрении восходящего анализа удобно располагать вершину стека справа (а не слева, как это делалось при рассмотрении нисходящего синтаксического анализа). Изначально стек пуст, а во входном буфере находится строка w:

Стек	Вход
\$	$w\$$

В процессе сканирования входной строки слева направо синтаксический анализатор выполняет нуль или несколько переносов символов в стек, пока не будет готов выполнить свертку строки β символов грамматики на вершине стека. Затем он выполняет свертку β к заголовку соответствующей продукции. Синтаксический анализатор повторяет этот цикл до тех пор, пока не будет обнаружена ошибка или пока стек не будет содержать только стартовый символ, а входной буфер будет при этом пуст:

Стек	Вход
$\$S$	\$

Достигнув указанной конфигурации, синтаксический анализатор останавливается и сообщает об успешном завершении анализа. На рис. 4.28 пошагово показаны действия ПС-анализатора, выполняемые при синтаксическом анализе строки $\mathbf{id}_1 * \mathbf{id}_2$ согласно грамматике выражений (4.1).

Хотя основными операциями являются перенос и свертка, всего ПС-анализатор может выполнять четыре действия: 1) перенос, 2) свертка, 3) принятие и 4) ошибка.

Стек	Вход	Действие
$\$$	$\mathbf{id}_1 * \mathbf{id}_2\$$	Перенос
$\$\mathbf{id}_1$	$* \mathbf{id}_2\$$	Свертка по $F \to \mathbf{id}$
$\$F$	$* \mathbf{id}_2\$$	Свертка по $T \to F$
$\$T$	$* \mathbf{id}_2\$$	Перенос
$\$T *$	$\mathbf{id}_2\$$	Перенос
$\$T * \mathbf{id}_2$	$\$$	Свертка по $F \to \mathbf{id}$
$\$T * F$	$\$$	Свертка по $T \to T * F$
$\$T$	$\$$	Свертка по $E \to T$
$\$E$	$\$$	Принятие

Рис. 4.28. Конфигурации ПС-анализатора при входной строке $\mathbf{id}_1 * \mathbf{id}_2$

1. *Перенос* (shift). Перенос очередного входного символа на вершину стека.
2. *Свертка* (reduce). Правая часть сворачиваемой строки должна располагаться на вершине стека. Определяется левый конец строки в стеке и принимается решение о том, каким нетерминалом будет заменена строка.
3. *Принятие* (accept). Объявление об успешном завершении синтаксического анализа.
4. *Ошибка* (error). Обнаружение синтаксической ошибки и вызов подпрограммы восстановления после ошибки.

Использование стека в ПС-анализаторе объясняется тем важным фактом, что основа всегда находится на вершине стека и никогда – внутри него. Это можно показать путем рассмотрения возможных видов двух последовательных шагов в любом правом порождении. На рис. 4.29 показаны эти два возможных случая. В случае (1) A заменяется на βBy, после чего крайний справа нетерминал B в теле βBy заменяется на γ. В случае (2) A снова раскрывается первым, но на этот раз тело представляет собой строку y, состоящую из одних терминалов. Следующий крайний справа нетерминал B будет находиться где-то слева от y.

Другими словами, имеем следующее:

1) $S \underset{rm}{\overset{*}{\Rightarrow}} \alpha Az \underset{rm}{\Rightarrow} \alpha\beta Byz \underset{rm}{\Rightarrow} \alpha\beta\gamma yz$

2) $S \underset{rm}{\overset{*}{\Rightarrow}} \alpha BxAz \underset{rm}{\Rightarrow} \alpha Bxyz \underset{rm}{\Rightarrow} \alpha\gamma xyz$

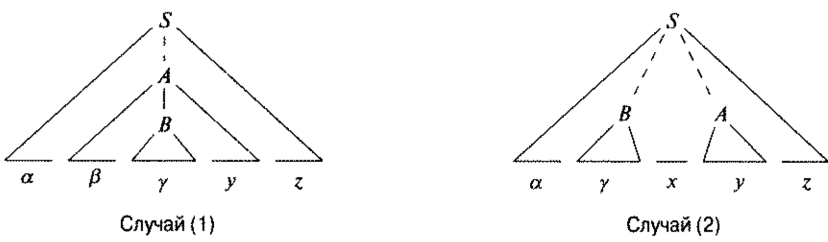


Рис. 4.29. Варианты двух последовательных шагов правого порождения

Рассмотрим случай (1) в обратном порядке, начиная с момента, когда ПС-анализатор достиг конфигурации

Стек	Вход
$\$\alpha\beta\gamma$	$yz\$$

Синтаксический анализатор сворачивает основу γ в B и достигает конфигурации

$\$\alpha\beta B$	$yz\$$

Теперь синтаксический анализатор может выполнить перенос строки y в стек при помощи нуля или нескольких шагов переноса и получить конфигурацию

$\$\alpha\beta By$	$z\$$

с основой βBy на вершине стека, после чего выполнить ее свертку в A.

Теперь рассмотрим случай (2). В конфигурации

$\$\alpha\gamma$	$xyz\$$

основа γ находится на вершине стека. После свертки основы γ в B синтаксический анализатор может перенести строку xy для получения очередной основы y на вершине стека, готового к свертке в A:

$\$\alpha Bxy$	$z\$$

В обоих случаях после выполнения свертки синтаксический анализатор должен один или несколько раз перенести символы для получения в стеке очередной основы. Он никогда не должен углубляться в ее поисках в стек.

4.5.4 Конфликты в процессе ПС-анализа

Имеются контекстно-свободные грамматики, для которых ПС-анализ неприменим. Любой ПС-анализатор для такой грамматики может достичь конфигурации,

в которой синтаксический анализатор, обладая информацией о содержимом стека и очередных k входных символах, не может принять решение о том, следует ли выполнить перенос или свертку (конфликт "перенос/свертка") либо какое именно из нескольких приведений должно быть выполнено (конфликт "свертка/свертка"). Ниже мы рассмотрим несколько примеров синтаксических конструкций, которые приводят к таким грамматикам. Технически эти грамматики не принадлежат классу LR(k)-грамматик, определенному в разделе 4.7; будем говорить о них как о не-LR-грамматиках. k в LR(k) указывает количество символов, которые предпросматриваются во входном потоке. Обычно используемые в компиляции грамматики принадлежат классу LR(1), т.е. выполняется предпросмотр не более одного символа.

Пример 4.24. Неоднозначная грамматика не может принадлежать классу LR. Рассмотрим, например, грамматику с "висящим **else**" (4.9) из раздела 4.3:

$$\begin{array}{lcl} stmt & \rightarrow & \textbf{if } expr \textbf{ then } stmt \\ & | & \textbf{if } expr \textbf{ then } stmt \textbf{ else } stmt \\ & | & \textbf{other} \end{array}$$

Если ПС-анализатор находится в конфигурации

Стек	Вход
$\cdots$ **if** *expr* **then** *stmt*	**else** $\cdots$ \$

то мы не можем сказать, является ли **if** *expr* **then** *stmt* основой, безотносительно к тому, что находится ниже его в стеке. Здесь мы сталкиваемся с конфликтом "перенос/свертка". В зависимости от того, что следует за **else** во входном потоке, верным решением может оказаться свертка **if** *expr* **then** *stmt* в *stmt* или перенос **else** и поиск еще одного *stmt* для завершения альтернативы **if** *expr* **then** *stmt* **else** *stmt*.

Следует отметить, однако, что ПС-анализ может быть адаптирован к разбору некоторых неоднозначных грамматик, таких как приведенная выше. При разрешении указанного конфликта "перенос/свертка" при обнаружении **else** в пользу переноса синтаксический анализатор будет работать так, как мы от него и ожидаем, связывая **else** с предыдущим **then**, которому еще не найдено соответствующее **else**. Синтаксические анализаторы для таких неоднозначных грамматик мы рассмотрим в разделе 4.8. □

Еще одна распространенная причина конфликта — когда у нас есть основа, но содержимого стека и очередного входного символа недостаточно для определения продукции, которая должна использоваться в свертке. Следующий пример иллюстрирует эту ситуацию.

Пример 4.25. Предположим, имеется лексический анализатор, который возвращает имя токена **id** для всех имен независимо от их типа. Предположим также, что наш язык вызывает процедуры по именам с параметрами, заключенными в скобки, и что тот же синтаксис используется и для работы с массивами. Поскольку трансляции индексов массива и параметров процедуры существенно отличаются друг от друга, мы должны использовать различные продукции для порождения списка фактических параметров и индексов. Следовательно, наша грамматика может иметь (среди прочих) продукции наподобие приведенных на рис. 4.30.

(1)	*stmt*	→	**id** (*parameter_list*)
(2)	*stmt*	→	*expr* := *expr*
(3)	*parameter_list*	→	*parameter_list* , *parameter*
(4)	*parameter_list*	→	*parameter*
(5)	*parameter*	→	**id**
(6)	*expr*	→	**id** (*expr_list*)
(7)	*expr*	→	**id**
(8)	*expr_list*	→	*expr_list* , *expr*
(9)	*expr_list*	→	*expr*

Рис. 4.30. Продукции, включающие вызов процедуры и обращение к массиву

Инструкция, начинающаяся с `p(i,j)`, будет передана синтаксическому анализатору как поток токенов **id** (**id**, **id**). После переноса первых трех токенов в стек ПС-анализатор окажется в конфигурации

Стек	Вход
⋯ **id** (**id**	, **id**) ⋯

Очевидно, что токен **id** на вершине стека должен быть свернут, но какой продукцией? Правильный выбор — продукцией (5), если `p` — процедура, и продукцией (7), если `p` — массив. Содержимое стека не может подсказать, чем является `p`; для принятия решения мы должны использовать информацию из таблицы символов, которая была занесена в стек при объявлении `p`.

Одно из решений состоит в замене токена **id** в продукции (1) на **procid** и использовании более интеллектуального лексического анализатора, который возвращает **procid** при распознавании лексемы, представляющей собой имя процедуры. Такой способ требует от лексического анализатора обращения к таблице символов перед тем, как вернуть токен.

Если мы внесем эти изменения, то при обработке `p(i,j)` синтаксический анализатор может оказаться в конфигурации

Стек	Вход
$\cdots$ **procid** (**id**	, **id**) $\cdots$

либо в конфигурации, приведенной ранее. В первом случае мы выбираем свертку с использованием продукции (5), в последнем — с использованием продукции (7). Обратите внимание, что выбор определяется третьим от вершины символом в стеке, который даже не участвует в свертке. Для управления разбором ПС-анализ может использовать информацию "из глубин" стека. □

4.5.5 Упражнения к разделу 4.5

Упражнение 4.5.1. Укажите основу каждой из приведенных ниже правосентенциальных форм для грамматики $S \to 0\ S\ 1 \mid 0\ 1$ из упражнения 4.2.2, *а*:

а) 000111;

б) $00S11$.

Упражнение 4.5.2. Повторите упражнение 4.5.1 для грамматики $S \to S\ S\ + + \mid S\ S\ * \mid a$ из упражнения 4.2.1 и следующих правосентенциальных форм:

а) $SSS + a * +$;

б) $SS + a * a+$;

в) $aaa * a + +$.

Упражнение 4.5.3. Проведите восходящий синтаксический анализ для следующих входных строк и грамматик:

а) входная строка 000111, соответствующая грамматике из упражнения 4.5.1;

б) входная строка $aaa * a + +$, соответствующая грамматике из упражнения 4.5.2.

4.6 Введение в LR-анализ: простой LR

Наиболее распространенный тип восходящих синтаксических анализаторов на сегодняшний день основан на концепции, называющейся LR(k)-анализом; L здесь означает сканирование входного потока слева направо, R — построение правого порождения в обратном порядке, а k — количество предпросматриваемых

символов входного потока, необходимое для принятия решения. Практический интерес представляют случаи $k = 0$ и $k = 1$, и здесь будут рассмотрены только LR-анализаторы с $k \leqslant 1$. Если (k) опущено, считается, что k равно 1.

В этом разделе рассматриваются базовые концепции LR-анализа и простейший метод построения ПС-анализаторов, называющийся простым LR (simple LR — SLR). Определенное знакомство с базовыми концепциями весьма полезно даже в том случае, когда для построения LR-анализатора используется автоматический генератор анализаторов. Мы начнем с "пунктов" и "состояний анализатора"; диагностические сообщения генератора LR-анализаторов обычно включают состояния синтаксического анализатора, которые могут использоваться для выяснения источника конфликтов синтаксического анализа.

В разделе 4.7 рассматриваются два более сложных метода — канонический LR и LALR — которые используются в большинстве LR-анализаторов.

4.6.1 Обоснование использования LR-анализаторов

LR-анализаторы управляются таблицами наподобие нерекурсивных LL-анализаторов из раздела 4.4.4. Грамматика, для которой можно построить таблицу синтаксического анализа с использованием одного из методов из этого и следующего разделов, называется *LR-грамматикой*. Интуитивно, чтобы грамматика была LR-грамматикой, достаточно, чтобы синтаксический анализатор, работающий слева направо методом переноса/свертки, был способен распознавать основы правосентенциальных форм при их появлении на вершине стека.

LR-анализ весьма привлекателен по множеству причин.

- LR-анализаторы могут быть созданы для распознавания практически всех конструкций языков программирования, для которых может быть написана контекстно-свободная грамматика. Контекстно-свободные грамматики, не являющиеся LR-грамматиками, существуют, однако для типичных конструкций языков программирования их вполне можно избежать.

- Метод LR-анализа — наиболее общий метод ПС-анализа без возврата, который, кроме того, не уступает в эффективности другим, более примитивным ПС-методам (см. список литературы к главе 4).

- LR-анализатор может обнаруживать синтаксические ошибки сразу же, как только это становится возможным при сканировании входного потока.

- Класс грамматик, которые могут быть проанализированы с использованием LR-методов, представляет собой истинное надмножество класса грамматик, которые могут быть проанализированы с использованием предиктивных или LL-методов синтаксического анализа. В случае грамматик, принадлежащих классу LR(k), мы должны быть способны распознать правую

часть продукции в порожденной ею правосентенциальной форме с дополнительным предпросмотром k входных символов. Это требование существенно мягче требования для LL(k)-грамматик, в которых мы должны быть способны распознать продукцию по первым k символам порождения ее тела. Таким образом, не должен казаться удивительным тот факт, что LR-грамматики могут описать существенно больше языков, чем LL-грамматики.

Основной недостаток LR-метода состоит в том, что построение LR-анализатора для грамматики типичного языка программирования вручную требует очень большого объема работы. Для решения этой задачи нужен специализированный инструмент — генератор LR-анализаторов. К счастью, имеется множество таких генераторов, и мы рассмотрим один из наиболее широко используемых — Yacc — в разделе 4.9. Такой генератор получает контекстно-свободную грамматику и автоматически строит ее синтаксический анализатор. Если грамматика содержит неоднозначности или другие конструкции, трудные для синтаксического анализа сканированием входного потока слева направо, генератор локализует их и предоставляет детальную диагностическую информацию.

4.6.2 Пункты и LR(0)-автомат

Каким образом ПС-анализатор выясняет, когда следует выполнять перенос, а когда — свертку? Например, когда стек на рис. 4.28 содержит $\$T$, а очередной входной символ — *, каким образом синтаксический анализатор узнает, что T на вершине стека — не основа, так что корректное действие — перенос, а не свертка T в E?

LR-анализатор принимает решение о выборе "перенос/свертка", поддерживая состояния, которые отслеживают, где именно в процессе синтаксического анализа мы находимся. Состояния представляют собой множества "пунктов". *LR(0)-пункт* (для краткости — просто *пункт*[5]) грамматики G — это продукция G с точкой в некоторой позиции правой части. Следовательно, продукция $A \to XYZ$ дает четыре пункта:

$$\begin{array}{lcl} A & \to & \cdot XYZ \\ A & \to & X \cdot YZ \\ A & \to & XY \cdot Z \\ A & \to & XYZ \cdot \end{array}$$

Продукция $A \to \epsilon$ генерирует единственный пункт $A \to \cdot$.

Интуитивно, пункт указывает, какую часть продукции мы уже просмотрели в данной точке в процессе синтаксического анализа. Например, пункт $A \to \cdot XYZ$

[5] В русскоязычной литературе иногда использовался термин "ситуация". — *Прим. пер.*

указывает, что во входном потоке мы ожидаем встретить строку, порождаемую XYZ. Пункт $A \to X \cdot YZ$ указывает, что нами уже просмотрена строка, порожденная X, и мы ожидаем получить из входного потока строку, порождаемую YZ. Пункт $A \to XYZ\cdot$ говорит о том, что уже обнаружено тело XYZ и что, возможно, пришло время свернуть XYZ в A.

Один набор множеств LR (0)-пунктов, именуемый *каноническим* набором LR (0), обеспечивает основу для построения детерминированного конечного автомата, который используется для принятия решений в процессе синтаксического анализа. Такой автомат называется LR (0)-*автоматом*[6]. В частности, каждое состояние LR (0)-автомата представляет множество пунктов в каноническом наборе LR (0). Автомат для грамматики выражений (4.1), показанный на рис. 4.31, будет служить в качестве примера при рассмотрении канонического LR (0)-набора грамматики.

Для построения канонического LR (0)-набора мы определяем расширенную грамматику и две функции, CLOSURE и GOTO. Если G — грамматика со стартовым символом S, то *расширенная грамматика* G' представляет собой G с новым стартовым символом S' и продукцией $S' \to S$. Назначение этой новой стартовой продукции — указать синтаксическому анализатору, когда следует прекратить анализ и сообщить о принятии входной строки; т.е. принятие осуществляется тогда и только тогда, когда синтаксический анализатор выполняет свертку с использованием продукции $S' \to S$.

Замыкание множеств пунктов

Если I — множество пунктов грамматики G, то CLOSURE (I) представляет собой множество пунктов, построенное из I согласно двум правилам.

1. Изначально в CLOSURE (I) добавляются все пункты из I.

2. Если $A \to \alpha \cdot B\beta$ входит в CLOSURE (I), а $B \to \gamma$ является продукцией, то в CLOSURE (I) добавляется пункт $B \to \cdot\gamma$, если его там еще нет. Это правило применяется до тех пор, пока не останется пунктов, которые могут быть добавлены в CLOSURE (I).

Интуитивно $A \to \alpha \cdot B\beta$ в CLOSURE (I) указывает, что в некоторой точке процесса синтаксического анализа мы полагаем, что далее во входной строке мы можем встретить подстроку, порождаемую из $B\beta$. Подстрока, порождаемая из $B\beta$, будет иметь префикс, порождаемый из B путем применения одной из B-продукций. Таким образом, мы добавляем пункты для всех B-продукций; т.е. если $B \to \gamma$ является продукцией, то мы включаем $B \to \cdot\gamma$ в CLOSURE (I).

---

[6]Технически автомат не является детерминированным по определению из раздела 3.6.4, поскольку не имеет тупикового состояния, соответствующего пустому множеству пунктов. В результате существует некоторое количество пар "состояние — входной символ", для которых отсутствует следующее состояние.

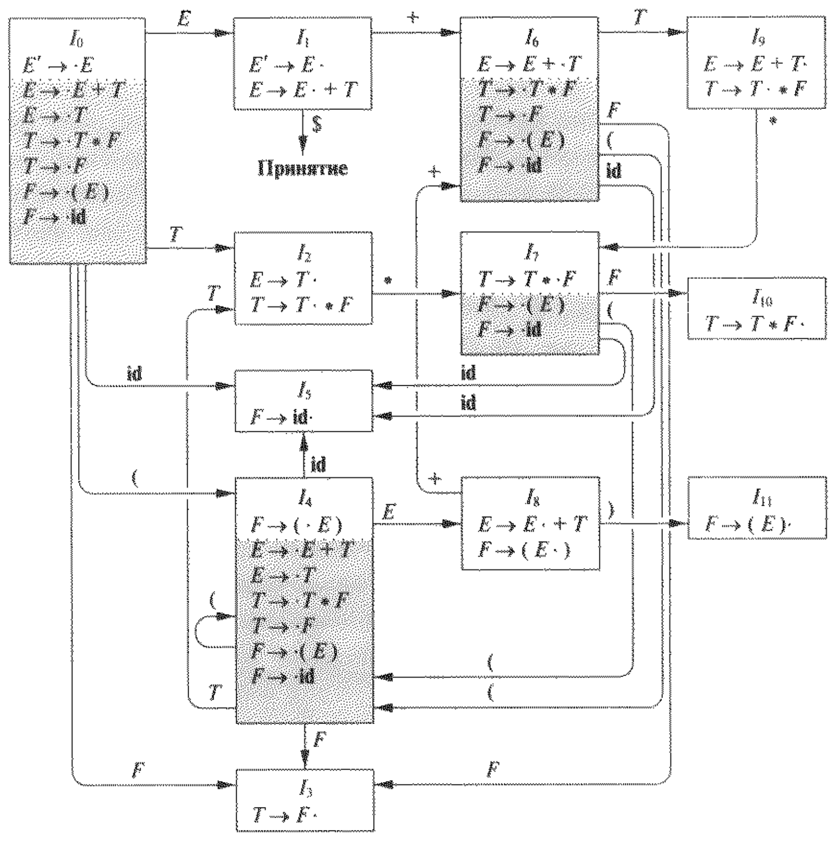


Рис. 4.31. LR (0)-автомат для грамматики выражений (4.1)

Представление множеств пунктов

Генератору синтаксических анализаторов, который создает восходящий анализатор, может потребоваться удобное представление пунктов и их множеств. Заметим, что пункт может быть представлен как пара целых чисел, первое из которых является номером одной из продукций грамматики, а второе — положением в ней точки. Множество пунктов может быть представлено списком этих пар. Однако, как мы вскоре увидим, требуемые множества пунктов часто включают "замыкающие" пункты, в которых точка находится в начале тела. Такие пункты всегда могут быть реконструированы из других пунктов множества, так что нам не требуется включать их в список.

Пример 4.26. Рассмотрим следующую расширенную грамматику выражений.

$$\begin{array}{lcc} E' & \to & E \\ E & \to & E + T \mid T \\ T & \to & T * F \mid F \\ F & \to & (E) \mid \mathbf{id} \end{array}$$

Если I — множество из одного пункта $\{[E' \to \cdot E]\}$, то CLOSURE(I) содержит множество пунктов I_0 на рис. 4.31.

Рассмотрим, как вычисляется замыкание. $E' \to \cdot E$ помещается в CLOSURE(I) согласно правилу (1). Поскольку непосредственно справа от точки находится E, мы добавляем E-продукции с точками на левом конце: $E \to \cdot E + T$ и $E \to \cdot T$. Теперь справа от точки в последней продукции находится T, так что следует добавить $T \to \cdot T * F$ и $T \to \cdot F$. Далее, F справа от точки заставляет добавить $F \to \cdot(E)$ и $F \to \cdot\mathbf{id}$, и больше никакие другие пункты не добавляются. □

Замыкание может быть вычислено так, как показано на рис. 4.32. Удобный способ реализации функции *closure* состоит в поддержании булева массива *added*, индексированного нетерминалами G, так что *added*$[B]$ устанавливается равным **true**, если и когда мы добавляем пункт $B \to \cdot\gamma$ для каждой B-продукции $B \to \gamma$.

```
SetOfItems CLOSURE(I) {
    J = I;
    repeat
        for ( каждый пункт A → α · Bβ из J )
            for ( каждая продукция B → γ из G )
                if ( B → ·γ не входит в J )
                    Добавить B → ·γ в J;
    until больше нет пунктов для добавления в J за один проход;
    return J;
}
```

Рис. 4.32. Вычисление CLOSURE

Заметим, что если одна B-продукция добавляется в замыкание I с точкой на левом конце, то в замыкание будут аналогичным образом добавлены все B-продукции. Следовательно, при некоторых условиях нет необходимости в перечислении пунктов $B \to \cdot\gamma$, добавленных в I при помощи функции CLOSURE; достаточно списка нетерминалов B, продукции которых были добавлены таким образом. Разделим множество интересующих нас пунктов на два класса.

1. *Базисные пункты*, или *пункты ядра* (kernel items): начальный пункт $S' \to \cdot S$ и все пункты, у которых точки расположены не у левого края.

2. *Небазисные* (nonkernel) пункты, у которых точки расположены слева, за исключением $S' \to \cdot S$.

Кроме того, каждое множество интересующих нас пунктов формируется как замыкание множества базисных пунктов; добавляемые в замыкание пункты не могут быть базисными. Таким образом, мы можем представить множества интересующих нас пунктов с использованием очень небольшого объема памяти, если отбросим все небазисные пункты, зная, что они могут быть восстановлены процессом замыкания. На рис. 4.31 небазисные пункты размещаются в заштрихованных частях прямоугольников состояний.

Функция GOTO

Второй полезной функцией является $\text{GOTO}(I, X)$, где I — множество пунктов, а X — грамматический символ. $\text{GOTO}(I, X)$ определяется как замыкание множества всех пунктов $[A \to \alpha X \cdot \beta]$, таких, что $[A \to \alpha \cdot X\beta]$ находится в I. Интуитивно функция GOTO используется для определения переходов в LR (0)-автомате грамматики. Состояния автомата соответствуют множествам пунктов, и $\text{GOTO}(I, X)$ указывает переход из состояния I при входном символе X.

Пример 4.27. Если I — множество из двух пунктов, $\{[E' \to E\cdot], [E \to E \cdot +T]\}$, то $\text{GOTO}(I, +)$ содержит пункты

$$\begin{array}{lcl} E & \to & E + \cdot T \\ T & \to & \cdot T * F \\ T & \to & \cdot F \\ F & \to & \cdot (E) \\ F & \to & \cdot \mathbf{id} \end{array}$$

$\text{GOTO}(I, +)$ вычисляется путем рассмотрения пунктов I, в которых + следует непосредственно за точкой. $E' \to E\cdot$ таким пунктом не является, но таковым является пункт $E \to E \cdot +T$. Поэтому мы переносим точку за +, получая пункт $E \to E + \cdot T$, а затем находим замыкание этого множества из одного элемента. □

Теперь мы готовы к построению канонического набора C множеств LR (0)-пунктов для расширенной грамматики G'. Соответствующий алгоритм показан на рис. 4.33.

Пример 4.28. Канонический набор множеств LR (0)-пунктов для грамматики (4.1) и функция GOTO показаны на рис. 4.31. Значения GOTO представлены на схеме переходами между состояниями. □

```
void items(G') {
    C = {CLOSURE({[S' → ·S]})};
    repeat
        for ( каждое множество пунктов I в C )
            for ( каждый грамматический символ X )
                if ( множество GOTO(I, X) не пустое и не входит в C )
                    Добавить GOTO(I, X) в C;
    until нет новых множеств пунктов для добавления в C за один проход;
}
```

Рис. 4.33. Вычисление канонического набора множеств LR (0)-пунктов

Использование LR(0)-автомата

Основная идея, лежащая в основе "простого LR", или SLR, синтаксического анализа заключается в построении LR (0)-автомата для заданной грамматики. Состояниями этого автомата являются множества пунктов из канонического набора LR (0), а переходы определяются функцией GOTO. LR (0)-автомат для грамматики выражений (4.1) показан на рис. 4.31.

Стартовое состояние LR (0)-автомата – CLOSURE $(\{[S' \to \cdot S]\})$, где S' – стартовый символ расширенной грамматики. Все состояния являются принимающими. Под состоянием j далее подразумевается состояние, соответствующее множеству пунктов I_j.

Каким образом LR (0)-автомат помогает в принятии решения "перенос/свертка"? Это решение может быть принято следующим образом. Предположим, что строка γ из символов грамматики переводит LR (0)-автомат из состояния 0 в некоторое состояние j. Тогда выполним перенос очередного входного символа a, если состояние j имеет переход для данного символа a. В противном случае выбирается свертка; пункт в состоянии j говорит нам, какую продукцию следует для этого использовать.

Алгоритм LR-анализа, приведенный в разделе 4.6.3, использует стек для отслеживания как состояний, так и символов грамматики; фактически символы грамматики могут быть восстановлены из состояний, так что стек хранит только состояния. Приведенный далее пример показывает, каким образом LR (0)-автомат и стек могут использоваться для принятия решения "перенос/свертка" в процессе синтаксического анализа.

Пример 4.29. На рис. 4.34 показаны действия синтаксического анализатора методом "перенос/свертка" для входной строки **id** $*$ **id** с использованием LR (0)-автомата из рис. 4.31. Стек используется для хранения состояний; для ясности в столбце "Символы" приведены символы грамматики, соответствующие состо-

яниям. В строке (1) в стеке находится стартовое состояние 0 автомата; соответствующий ему символ — маркер дна стека \$.

Строка	Стек	Символы	Вход	Действие
(1)	0	\$	**id** $*$ **id**\$	Перенос в 5
(2)	0 5	\$**id**	$*$**id**\$	Свертка по $F \to$ **id**
(3)	0 3	$\$F$	$*$**id**\$	Свертка по $T \to F$
(4)	0 2	$\$T$	$*$**id**\$	Перенос в 7
(5)	0 2 7	$\$T*$	**id**\$	Перенос в 5
(6)	0 2 7 5	$\$T*$ **id**	\$	Свертка по $F \to$ **id**
(7)	0 2 7 10	$\$T*F$	\$	Свертка по $T \to T*F$
(8)	0 2	$\$T$	\$	Свертка по $E \to T$
(9)	0 1	$\$E$	\$	Принятие

Рис. 4.34. Синтаксический анализ строки **id** $*$ **id**

Очередной входной символ — **id**, а состояние 0 имеет переход по **id** в состояние 5. Таким образом, выбирается перенос. В строке (2) состояние 5 (символ **id**) вносится в стек. Переходов из состояния 5 для входного символа * нет, так что выбирается свертка. Пункт $[F \to \mathbf{id}\cdot]$ в состоянии 5 указывает, что свертка выполняется с использованием продукции $F \to$ **id**.

Что касается символов, то свертка выполняется путем снятия тела продукции со стека (в строке (2) тело продукции — **id**) и размещения в нем заголовка продукции (в данном случае — F). В стеке мы снимаем состояние 5 для символа **id**, что приводит к поднятию состояния 0 на вершину стека, и ищем переходы для F, заголовка использованной для свертки продукции. На рис. 4.31 состояние 0 имеет переход по F в состояние 3, так что в стек помещается состояние 3 с соответствующим символом F (см. строку (3)).

В качестве еще одного примера рассмотрим строку (5) с состоянием 7 (символ *) на вершине стека. Это состояние имеет переход в состояние 5 для входного символа **id**, так что мы помещаем в стек состояние 5 (символ **id**). У состояния 5 нет переходов, поэтому выполняется свертка в соответствии с продукцией $F \to$ **id**. Когда со стека снимается состояние 5, соответствующее телу продукции **id**, на вершине стека оказывается состояние 7. Поскольку состояние 7 имеет переход по символу F в состояние 10, мы вносим в стек состояние 10 (символ F). □

4.6.3 Алгоритм LR-анализа

Схематически LR-анализатор показан на рис. 4.35. Он состоит из входного буфера, выхода, стека, программы-драйвера и таблицы синтаксического анализа,

состоящей из двух частей (ACTION и GOTO). Программа-драйвер одинакова для всех LR-анализаторов; от одного анализатора к другому меняются таблицы синтаксического анализа. Программа синтаксического анализа по одному считывает символы из входного буфера. Там, где ПС-анализатор должен перенести символ, LR-анализатор переносит *состояние*. Каждое состояние подытоживает информацию, содержащуюся в стеке ниже него.

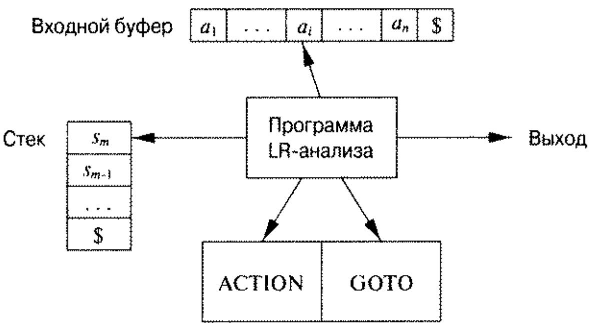


Рис. 4.35. Модель LR-анализатора

Стек хранит последовательность состояний $s_0 s_1 \ldots s_m$, где s_m находится на вершине стека. В случае метода SLR стек хранит состояния LR(0)-автомата; канонический метод LR и LALR-метод аналогичны. В соответствии с построением каждое состояние имеет соответствующий грамматический символ. Вспомним, что состояния соответствуют множествам пунктов и что существует переход из состояния i в состояние j, если $\text{GOTO}(I_i, X) = I_j$. Все переходы в состояние j должны соответствовать одному и тому же символу грамматики X. Таким образом, каждое состояние, за исключением стартового состояния 0, имеет единственный грамматический символ, связанный с ним.[7]

Структура таблицы LR-анализа

Таблица синтаксического анализа состоит из двух частей: функции действий синтаксического анализа ACTION и функции переходов GOTO.

1. Функция ACTION принимает в качестве аргумента состояние i и терминал a (или $, маркер конца входной строки). Значение $\text{ACTION}[i, a]$ может быть одного из следующих видов.

[7]Обратное не обязательно, т.е. один и тот же грамматический символ могут иметь несколько состояний. В качестве примера можно привести состояния 1 и 8 в LR(0)-автомате на рис. 4.31, вход в которые осуществляется переходом по E, или состояния 2 и 9, вход в которые осуществляется переходом по T.

а) Перенос j, где j — состояние. Действие, предпринимаемое синтаксическим анализатором, эффективно переносит входной символ a в стек, но для представления a использует состояние j.

б) Свертка $A \to \beta$. Действие синтаксического анализатора состоит в эффективной свертке β на вершине стека в заголовок A.

в) Принятие. Синтаксический анализатор принимает входную строку и завершает анализ.

г) Ошибка. Синтаксический анализатор обнаруживает ошибку во входной строке и предпринимает некоторое корректирующее действие. Более подробно о таких подпрограммах восстановления после ошибки будет говориться в разделах 4.8.3 и 4.9.4.

2. Функция GOTO, определенная на множествах пунктов, распространяется на состояния: если GOTO $[I_i, A] = I_j$, то GOTO отображает также состояние i и нетерминал A на состояние j.

Конфигурации LR-анализатора

Описать поведение LR-анализатора можно с помощью обозначений, представляющих полное состояние синтаксического анализатора: его стек и оставшуюся непроанализированной часть входной строки. *Конфигурация* LR-анализатора представляет собой пару

$$(s_0 s_1 \ldots s_m, a_i a_{i+1} \ldots a_n \$)$$

Здесь первый компонент — содержимое стека (вершина стека справа), а второй компонент — оставшаяся непроанализированной часть входной строки. Эта конфигурация представляет правосентенциальную форму

$$X_1 X_2 \ldots X_m a_i a_{i+1} \ldots a_n$$

по сути, тем же способом, что и ПС-анализатор; единственное отличие заключается в том, что вместо символов грамматики в стеке хранятся состояния, из которых могут быть восстановлены грамматические символы. Иначе говоря, X_i является грамматическим символом, представленным состоянием s_i. Заметим, что стартовое состояние синтаксического анализатора s_0 не представляет символ грамматики, а служит маркером дна стека и играет важную роль в процессе анализа.

Поведение LR-анализатора

Очередной шаг синтаксического анализатора из приведенной выше конфигурации определяется считанным текущим входным символом a_i и состоянием на вершине стека s_m путем обращения к записи ACTION $[s_m, a_i]$ в таблице действий

синтаксического анализа. В результате выполнения указанного в записи действия (одного из четырех возможных типов) получаются следующие конфигурации.

1. Если ACTION $[s_m, a_i]$ = перенос s, синтаксический анализатор выполняет перенос в стек очередного состояния s и его конфигурацией становится

   $$(s_0 s_1 \ldots s_m s, a_{i+1} \ldots a_n \$)$$

   Символ a_i хранить в стеке не нужно, поскольку при необходимости (что на практике никогда не требуется) он может быть восстановлен из s. Текущим входным символом становится a_{i+1}.

2. Если ACTION $[s_m, a_i]$ = свертка $A \to \beta$, синтаксический анализатор выполняет свертку и его конфигурацией становится

   $$(s_0 s_1 \ldots s_{m-r} s, a_i a_{i+1} \ldots a_n \$)$$

   Здесь r — длина β, а s = GOTO $[s_{m-r}, A]$. Синтаксический анализатор вначале снимает r символов состояний с вершины стека, что переносит на вершину стека состояние s_{m-r}, после чего на вершину стека помещается s, запись из GOTO $[s_{m-r}, A]$. При свертке текущий входной символ не изменяется. В LR-анализаторах, которые мы будем строить, последовательность символов грамматики $X_{m-r+1} \ldots X_m$ всегда соответствует β, правой части продукции свертки.

3. Если ACTION $[s_m, a_i]$ = принятие, синтаксический анализ завершается.

4. Если ACTION $[s_m, a_i]$ = ошибка, синтаксический анализатор обнаруживает ошибку и вызывает подпрограмму восстановления после ошибки.

Алгоритм LR-анализа приведен ниже. Все LR-анализаторы ведут себя подобным образом; единственное отличие одного LR-анализатора от другого заключается в информации в записях полей ACTION и GOTO таблицы синтаксического анализа.

Алгоритм 4.30. Алгоритм LR-анализа

ВХОД: входная строка w и таблица LR-анализа с функциями ACTION и GOTO для грамматики G.

ВЫХОД: если $w \in L(G)$, шаги сверток восходящего синтаксического анализа w; в противном случае — указание о происшедшей ошибке.

МЕТОД: изначально в стеке синтаксического анализатора находится начальное состояние s_0, а во входном буфере — $w\$$. Затем синтаксический анализатор выполняет программу, приведенную на рис. 4.36. □

```
Пусть a — первый символ w$.
while(1) { /* Бесконечный цикл */
        Пусть s — состояние на вершине стека.
        if ( ACTION[s, a] = перенос t ) {
                Внести t в стек.
                Присвоить a очередной входной символ.
        } else if ( ACTION[s, a] = свертка A → β ) {
                Снять |β| символов со стека.
                Пусть теперь на вершине стека находится состояние t.
                Внести в стек GOTO[t, A].
                Вывести продукцию A → β.
        } else if ( ACTION[s, a] = принятие ) {
                break; /* Синтаксический анализ завершен */
        } else Вызов подпрограммы восстановления после ошибки.
}
```

Рис. 4.36. Программа LR-анализа

Пример 4.31. На рис. 4.37 показаны функции ACTION и GOTO из таблицы LR-анализа грамматики выражений (4.1), повторенной ниже с пронумерованными продукциями:

$$
\begin{array}{llll}
(1) & E \to E + T & (4) & T \to F \\
(2) & E \to T & (5) & F \to (E) \\
(3) & T \to T * F & (6) & F \to \mathbf{id}
\end{array}
$$

Коды действий следующие.

1. si означает перенос и размещение в стеке состояния i.

2. rj означает свертку в соответствии с продукцией с номером j.

3. acc означает принятие.

4. Пустое поле означает ошибку.

Заметим, что значение GOTO$[s, a]$ для терминала a находится в поле ACTION, связанном с переносом для входного символа a и состояния s. Поле GOTO дает значения GOTO$[s, a]$ для нетерминалов A. Кроме того, следует иметь в виду, что мы пока что не пояснили, каким образом выбираются записи на рис. 4.37 (об этом поговорим немного позже).

Для входной строки $\mathbf{id} * \mathbf{id} + \mathbf{id}$ последовательность содержимого стека и входной строки показана на рис. 4.38. Для ясности показана также последовательность

Состояние	ACTION						GOTO		
	id	+	*	(	)	$	E	T	F
0	s5			s4			1	2	3
1		s6				acc			
2		r2	s7		r2	r2			
3		r4	r4		r4	r4			
4	s5			s4			8	2	3
5		r6	r6		r6	r6			
6	s5			s4				9	3
7	s5			s4					10
8		s6			s11				
9		r1	s7		r1	r1			
10		r3	r3		r3	r3			
11		r5	r5		r5	r5			

Рис. 4.37. Таблица синтаксического анализа для грамматики выражений

грамматических символов, соответствующая хранящимся в стеке состояниям. Например, в строке (1) LR-анализатор находится в состоянии 0, начальном состоянии без грамматических символов, и с **id** в качестве первого входного символа. Действие в строке 0 и столбце **id** поля ACTION на рис. 4.37 — s5; оно означает перенос и внесение в стек состояния 5. В строке (2) выполняется внесение в стек символа состояния 5 и удаление **id** из входного потока.

После этого текущим входным символом становится *; действие для состояния 5 и входного символа * — свертка согласно продукции $F \to$ **id**. Со стека при этом снимается один символ состояния, и на вершине стека появляется состояние 0. Поскольку GOTO $[0, F]$ равно s3, в стек вносится состояние 3. При этом получается конфигурация, показанная в строке (3). Остальные строки на рис. 4.38 получены аналогично. □

4.6.4 Построение таблиц SLR-анализа

SLR-метод построения таблиц синтаксического анализа — хорошее начало для изучения LR-анализа. Далее таблицы синтаксического анализа, построенные этим методом, будут именоваться SLR-таблицами, а LR-анализатор, использующий SLR-таблицы, — SLR-анализатором. Два других метода расширяют SLR-метод путем информации, получаемой предпросмотром входной строки.

	Стек	Символы	Вход	Действие
(1)	0		**id** $*$ **id** $+$ **id**\$	Перенос
(2)	0 5	**id**	$*$**id** $+$ **id**\$	Свертка по $F \to$ **id**
(3)	0 3	F	$*$**id** $+$ **id**\$	Свертка по $T \to F$
(4)	0 2	T	$*$**id** $+$ **id**\$	Перенос
(5)	0 2 7	$T*$	**id** $+$ **id**\$	Перенос
(6)	0 2 7 5	$T *$ **id**	$+$ **id**\$	Свертка по $F \to$ **id**
(7)	0 2 7 10	$T * F$	$+$ **id**\$	Свертка по $T \to T * F$
(8)	0 2	T	$+$ **id**\$	Свертка по $E \to T$
(9)	0 1	E	$+$ **id**\$	Перенос
(10)	0 1 6	$E+$	**id**\$	Перенос
(11)	0 1 6 5	$E +$ **id**	\$	Свертка по $F \to$ **id**
(12)	0 1 6 3	$E + F$	\$	Свертка по $T \to F$
(13)	0 1 6 9	$E + T$	\$	Свертка по $E \to E + T$
(14)	0 1	E	\$	Принятие

Рис. 4.38. Действия LR-анализатора для строки **id** $*$ **id** $+$ **id**

SLR-метод начинается с LR (0)-пунктов и LR (0)-автомата, с которыми вы познакомились в разделе 4.5, т.е. для данной грамматики G мы строим ее расширение G' с новым стартовым символом S'. Для G' строится канонический набор C множеств пунктов G' вместе с функцией GOTO.

Затем строятся записи ACTION и GOTO в таблице синтаксического анализа с использованием следующего алгоритма, который требует от нас знания FOLLOW (A) для каждого нетерминала A грамматики (см. раздел 4.4).

Алгоритм 4.32. Построение таблицы SLR-анализа

Вход: расширенная грамматика G'.

Выход: функции таблицы SLR-анализа ACTION и GOTO для грамматики G'.

Метод: выполняем следующие действия.

1. Строим $C = \{I_0, I_1, \ldots, I_n\}$ — набор множеств LR (0)-пунктов для G'.

2. Из I_i строим состояние i. Действие синтаксического анализа для состояния i определяем следующим образом.

   а) Если $[A \to \alpha \cdot a\beta] \in I_i$ и GOTO $(I_i, a) = I_j$, то устанавливаем ACTION $[i, a]$ равным "перенос j". Здесь a должно быть терминалом.

б) Если $[A \to \alpha\cdot] \in I_i$, то устанавливаем ACTION $[i, a]$ равным "свертка $A \to \alpha$" для всех a из FOLLOW (A); здесь A не должно быть равно S'.

в) Если $[S' \to S\cdot] \in I_i$, то устанавливаем ACTION $[i, \$]$ равным "принятие".

При наличии любого конфликта между действиями, возникшего в результате применения указанных правил, делается вывод о том, что грамматика не принадлежит классу SLR (1). В таком случае алгоритм не может построить синтаксический анализатор для данной грамматики.

3. Переходы для состояния i строим для всех нетерминалов A с использованием следующего правила: если GOTO $(I_i, A) = I_j$, то GOTO $[i, A] = j$.

4. Все записи, не определенные правилами 2 и 3, получают значение "ошибка".

5. Начальное состояние синтаксического анализатора строим из множества пунктов, содержащего $[S' \to \cdot S]$. □

Таблица синтаксического анализа, состоящая из функций ACTION и GOTO, определенных при помощи алгоритма 4.32, называется SLR (1)-*таблицейG*. LR-анализатор с использованием SLR (1)-таблицы для G называется SLR (1)-анализатором G, а грамматика, имеющая SLR (1)-таблицу, называется SLR (1)-*грамматикой*. Обычно в SLR (1) опускается (1), идущее после SLR, поскольку мы не работаем с синтаксическими анализаторами, предпросматривающими более одного символа.

Пример 4.33. Построим SLR-таблицу для расширенной грамматики выражений. Канонический набор множеств LR (0)-пунктов для этой грамматики был показан на рис. 4.31. Сначала рассмотрим множество пунктов I_0:

$$\begin{array}{lcl} E' & \to & \cdot E \\ E & \to & \cdot E + T \\ E & \to & \cdot T \\ T & \to & \cdot T * F \\ T & \to & \cdot F \\ F & \to & \cdot (E) \\ F & \to & \cdot \mathbf{id} \end{array}$$

Пункт $F \to \cdot(E)$ приводит к записи ACTION $[0, (] =$ перенос 4, а пункт $F \to \cdot\mathbf{id}$ – к записи ACTION $[0, \mathbf{id}] =$ перенос 5. Прочие пункты в I_0 действий не дают. Теперь рассмотрим пункты I_1:

$$\begin{array}{lcl} E' & \to & E\cdot \\ E & \to & E \cdot +T \end{array}$$

Первый пункт дает ACTION $[1, \$]$ = принятие, а второй — ACTION $[1, +]$ = перенос 6. Теперь очередь I_2:

$$\begin{array}{lcl} E & \to & T \cdot \\ T & \to & T \cdot *F \end{array}$$

Поскольку FOLLOW $(E) = \{\$, +,)\}$, первый пункт приводит к

$$\text{ACTION}[2, \$] = \text{ACTION}[2, +] = \text{ACTION}[2,)] = \text{свертка } E \to T.$$

Второй пункт дает ACTION $[2, *]$ = перенос 7. Продолжая работу таким образом, мы получим таблицы ACTION и GOTO, показанные на рис. 4.31. На этом рисунке номера продукций в свертках те же, что и порядок, в котором они находятся в исходной грамматике (4.1), т.е. продукция $E \to E + T$ имеет номер 1, $E \to T$ — номер 2 и т.д. □

Пример 4.34. Каждая SLR (1)-грамматика однозначна, однако имеется множество однозначных грамматик, не принадлежащих классу SLR (1). Рассмотрим грамматику с продукциями

$$\begin{array}{lcl} S & \to & L = R \mid R \\ L & \to & *R \mid \textbf{id} \\ R & \to & L \end{array} \tag{4.15}$$

L и R можно рассматривать как l- и r-значение соответственно, а * — как оператор “содержимое”.[8] Канонический набор множеств LR (0) пунктов для грамматики (4.15) показан на рис. 4.39.

Рассмотрим множество пунктов I_2. Первый пункт в этом множестве приводит к тому, что ACTION $[2, =]$ становится равным “перенос 6”. Поскольку FOLLOW (R) содержит = (чтобы увидеть, что это так, рассмотрите порождение $S \Rightarrow L \Rightarrow R \Rightarrow *R = R$), второй пункт устанавливает запись ACTION $[2, =]$ равной “свертка $R \to L$”. Поскольку в записи ACTION $[2, =]$ одновременно оказываются и перенос, и свертка, при входном символе = в состоянии 2 наблюдается конфликт “перенос/свертка”.

Грамматика (4.15) не является неоднозначной. Этот конфликт “перенос/свертка” возникает из того факта, что метод построения SLR-анализатора не достаточно мощный для запоминания достаточного левого контекста для принятия решения о том, какое действие должно быть предпринято синтаксическим анализатором для входного символа = при наличии строки, свертываемой в L. Канонический метод и LALR-метод, которые мы рассмотрим далее, успешно работают с бóльшим

[8]Как и в разделе 2.8.3, l-значение описывает ячейку памяти, а r-значение — значение, хранящееся в этой ячейке.

$$
\begin{array}{llll}
I_0: & S' \to \cdot S & I_5: & L \to \mathbf{id}\cdot \\
 & S \to \cdot L = R & & \\
 & S \to \cdot R & I_6: & S \to L = \cdot R \\
 & L \to \cdot * R & & R \to \cdot L \\
 & L \to \cdot \mathbf{id} & & L \to \cdot * R \\
 & R \to \cdot L & & L \to \cdot \mathbf{id} \\
 & & & \\
I_1: & S' \to S\cdot & I_7: & L \to *R\cdot \\
 & & & \\
I_2: & S \to L\cdot = R & I_8: & R \to L\cdot \\
 & R \to L\cdot & & \\
 & & I_9: & S \to L = R\cdot \\
I_3: & S \to R\cdot & & \\
 & & & \\
I_4: & L \to * \cdot R & & \\
 & R \to \cdot L & & \\
 & L \to \cdot * R & & \\
 & L \to \cdot \mathbf{id} & &
\end{array}
$$

Рис. 4.39. Канонический LR (0)-набор для грамматики (4.15)

набором грамматик, включая грамматику (4.15). Заметим, однако, что существуют такие однозначные грамматики, для которых любой метод построения LR-анализатора приводит к таблице действий с наличием конфликтов. К счастью, при разработке реальных языков программирования таких грамматик можно избежать. □

4.6.5 Активные префиксы

Почему LR (0)-автоматы могут использоваться при принятии решений "перенос/свертка"? LR (0)-автомат для грамматики характеризует строки грамматических символов, которые могут находиться в стеке ПС-анализатора грамматики. Содержимое стека должно быть префиксом правосентенциальной формы. Если в стеке хранится α, а оставшаяся часть входной строки — x, то последовательность сверток должна привести αx в S. В терминах порождений $S \overset{*}{\underset{rm}{\Rightarrow}} \alpha x$.

Однако в стеке могут находиться не все префиксы правосентенциальных форм, поскольку синтаксический анализатор не должен выполнять перенос после осно-

вы. Предположим, например,

$$E \underset{rm}{\overset{*}{\Rightarrow}} F * \mathbf{id} \underset{rm}{\Rightarrow} (E) * \mathbf{id}$$

Тогда в разные моменты времени в процессе синтаксического анализа в стеке хранятся (, $(E$ и (E), но в нем не должно находиться $(E)*$, поскольку (E) является основой, которую синтаксический анализатор должен свернуть в F до того, как выполнит перенос *.

Префиксы правосентенциальных форм, которые могут находиться в стеке ПС-анализатора, называются *активными префиксами* (viable prefixes). Они определяются следующим образом: активный префикс является префиксом правосентенциальной формы, не выходящим за пределы правого конца крайней справа основы сентенциальной формы. В соответствии с этим определением к концу активного префикса всегда можно добавить терминальные символы для получения правосентенциальной формы.

SLR-анализ основан на том факте, что LR (0)-автомат распознает активные префиксы. Мы говорим, что пункт $A \rightarrow \beta_1 \cdot \beta_2$ *допусти́м* (valid) для активного префикса $\alpha\beta_1$, если существует порождение $S' \underset{rm}{\Rightarrow}^* \alpha A w \underset{rm}{\Rightarrow} \alpha\beta_1\beta_2 w$. Вообще говоря, пункт может быть допустимым для многих активных префиксов.

Тот факт, что $A \rightarrow \beta_1 \cdot \beta_2$ допусти́м для $\alpha\beta_1$, многое говорит нам о том, что именно следует выбрать — перенос или свертку — при обнаружении $\alpha\beta_1$ в стеке. В частности, если $\beta_2 \neq \epsilon$, то это предполагает, что основа еще не полностью перенесена в стек и очередное действие анализатора — перенос. Если $\beta_2 = \epsilon$, то $A \rightarrow \beta_1$ — основа, и мы должны выполнить свертку в соответствии с этой продукцией. Конечно, два допустимых пункта могут указать на разные действия для одного и того же активного префикса. Одни из этих конфликтов могут быть разрешены путем просмотра очередного входного символа, а другие придется разрешать специальными методами, описанными в разделе 4.8. Однако не следует считать, что при применении LR-метода для построения таблицы синтаксического анализа произвольной грамматики могут быть разрешены все конфликты.

Можно легко вычислить множество допустимых пунктов для каждого активного префикса, который может появиться в стеке LR-анализатора. Основная теорема теории LR-анализа гласит, что множество допустимых пунктов для активного префикса γ в точности равно множеству пунктов, достижимых в LR (0)-автомате для данной грамматики из начального состояния по пути, помеченному γ. По сути, множество допустимых пунктов содержит в себе всю полезную информацию, которая может быть собрана из стека. Поскольку в этой книге мы не доказываем данную теорему, приведем соответствующий пример.

Пример 4.35. Рассмотрим расширенную грамматику выражений, множества пунктов и функция GOTO которой представлены на рис. 4.31. Очевидно, что строка $E + T*$ является активным префиксом этой грамматики. Автомат на рис. 4.31

Пункты как состояния НКА

Недетерминированный конечный автомат N для распознавания активных префиксов может быть построен путем рассмотрения пунктов в качестве состояний. Существует переход из $A \to \alpha \cdot X\beta$ в $A \to \alpha X \cdot \beta$, помеченный X, и переход из $A \to \alpha \cdot B\beta$ в $B \to \cdot\gamma$, помеченный ϵ. В таком случае CLOSURE (I) для множества пунктов (состояний N) I в точности представляет собой ϵ-замыкание множества состояний НКА, определенного в разделе 3.7.1. Таким образом, GOTO (I, X) дает переход из I для символа X в ДКА, построенном из N при помощи метода построения подмножеств. При таком подходе процедура *items* (G') на рис. 4.33 представляет собой процедуру построения подмножества, примененную к НКА N, состояниями которого являются пункты.

после чтения $E + T*$ будет находиться в состоянии 7. Это состояние содержит пункты

$$\begin{array}{lcl} T & \to & T * \cdot F \\ F & \to & \cdot (E) \\ F & \to & \cdot \mathbf{id} \end{array}$$

Они в точности являются допустимыми пунктами для $E + T*$. Чтобы увидеть, почему это так, рассмотрим следующие три правые порождения:

$$\begin{array}{lcl} E' & \underset{rm}{\Rightarrow} & E \\ & \underset{rm}{\Rightarrow} & E + T \\ & \underset{rm}{\Rightarrow} & E + T * F \end{array} \qquad \begin{array}{lcl} E' & \underset{rm}{\Rightarrow} & E \\ & \underset{rm}{\Rightarrow} & E + T \\ & \underset{rm}{\Rightarrow} & E + T * F \\ & \underset{rm}{\Rightarrow} & E + T * (E) \end{array} \qquad \begin{array}{lcl} E' & \underset{rm}{\Rightarrow} & E \\ & \underset{rm}{\Rightarrow} & E + T \\ & \underset{rm}{\Rightarrow} & E + T * F \\ & \underset{rm}{\Rightarrow} & E + T * \mathbf{id} \end{array}$$

Первое порождение показывает допустимость $T \to T * \cdot F$, второе — допустимость $F \to \cdot (E)$, а третье — допустимость $F \to \cdot\mathbf{id}$. Можно показать, что других допустимых пунктов для $E + T*$ не существует, хотя мы и не будем доказывать здесь этот факт. □

4.6.6 Упражнения к разделу 4.6

Упражнение 4.6.1. Опишите все активные префиксы следующих грамматик:

а) грамматика $S \to 0\ S\ 1 \mid 0\ 1$ из упражнения 4.2.2, *а*;

! б) грамматика $S \to S\ S+ \mid S\ S* \mid a$ из упражнения 4.2.1;

! в) грамматика $S \to S\ (S)\ |\ \epsilon$ из упражнения 4.2.2, *в*.

Упражнение 4.6.2. Постройте SLR-множества пунктов для (расширенной) грамматики из упражнения 4.2.1. Вычислите функцию GOTO для этих множеств пунктов. Приведите таблицу синтаксического анализа для этой грамматики. Принадлежит ли эта грамматика к классу SLR?

Упражнение 4.6.3. Приведите действия вашей таблицы синтаксического анализа из упражнения 4.6.2 для входной строки $aa * a+$.

Упражнение 4.6.4. Для каждой из (расширенных) грамматик из упражнений 4.2.2, *а–ж*:

а) постройте SLR-множества пунктов и их функции GOTO;

б) укажите конфликты действий в ваших множествах пунктов;

в) постройте таблицы SLR-анализа, если таковые существуют.

Упражнение 4.6.5. Покажите, что грамматика

$$\begin{array}{lcl} S & \to & A\,a\,A\,b\,|\,B\,b\,B\,a \\ A & \to & \epsilon \\ B & \to & \epsilon \end{array}$$

принадлежит классу LL (1), но не SLR (1).

Упражнение 4.6.6. Покажите, что грамматика

$$\begin{array}{lcl} S & \to & S\,A\,|\,A \\ A & \to & a \end{array}$$

принадлежит классу SLR (1), но не LL (1).

!! **Упражнение 4.6.7.** Рассмотрим семейство грамматик G_n, определяемое следующим образом:

$$\begin{array}{lcll} S & \to & A_i b_i & \text{для } 1 \leqslant i \leqslant n, \\ A_i & \to & a_j\,A_i\,|\,a_j & \text{для } 1 \leqslant i,j \leqslant n \text{ и } i \neq j. \end{array}$$

Покажите, что

а) G_n имеет $2n^2 - n$ продукций;

б) G_n имеет $2^n + n^2 + n$ множеств LR (0)-пунктов;

в) G_n принадлежит классу SLR (1).

Что говорит проведенный анализ о возможном размере LR-анализаторов?

! **Упражнение 4.6.8.** Допустим, что отдельные пункты можно рассматривать как состояния недетерминированного конечного автомата, в то время как множества допустимых пунктов являются состояниями детерминированного конечного автомата (см. врезку "Пункты как состояния НКА" в разделе 4.6.5). Для грамматики $S \to S\, S+ \mid S\, S* \mid a$ из упражнения 4.2.1 сделайте следующее.

а) Изобразите диаграмму переходов (НКА) для допустимых пунктов этой грамматики в соответствии с правилами из упомянутой врезки.

б) Примените метод построения подмножеств (алгоритм 3.20) к вашему НКА из части a. Каков размер получившегося ДКА по сравнению с множеством LR (0) пунктов грамматики?

!! в) Покажите, что во всех случаях применение метода построения подмножеств к НКА, полученному из допустимых пунктов грамматики, дает множества LR (0)-пунктов.

! **Упражнение 4.6.9.** Ниже приведена неоднозначная грамматика

$$\begin{array}{lcl} S & \to & A\, S \mid b \\ A & \to & S\, A \mid a \end{array}$$

Постройте для данной грамматики ее набор множеств LR (0)-пунктов. Если мы попытаемся построить таблицу LR-анализа для данной грамматики, то получим некоторые конфликты действий. Какие именно? Предположим, что мы пытаемся использовать таблицу синтаксического анализа недетерминированно, выбирая при конфликте возможное действие. Приведите все возможные последовательности действий для входной строки $abab$.

4.7 Более мощные LR-анализаторы

В этом разделе мы расширим рассмотренные методы LR-анализа использованием предпросмотра одного символа входного потока. Существует два различных метода.

1. Канонический LR, или просто LR-метод, использующий предпросмотр символа (или символов). Этот метод использует большое множество пунктов, именуемых LR (1)-пунктами.

2. LR с предпросмотром, или LALR (lookahead LR)-метод, основанный на множестве пунктов LR(0) и имеющий существенно меньше состояний, чем типичный анализатор, основанный на LR(1)-пунктах. Путем аккуратного добавления предпросмотров в LR(0)-пункты LALR позволяет работать с существенно большим количеством грамматик, чем SLR, и при этом строить таблицы синтаксического анализа, которые не больше SLR-таблиц. LALR — это метод, выбираемый в большинстве случаев.

После рассмотрения обоих этих методов мы завершим наше обсуждение вопросом о том, каким образом можно сжать LR-таблицы при работе в средах с ограниченной памятью.

4.7.1 Канонические LR(1)-пункты

Сейчас мы представим наиболее общий метод построения таблиц LR синтаксического анализа для грамматики. Вспомним, что в SLR-методе состояние i вызывает свертку в соответствии с продукцией $A \to \alpha$, если множество пунктов I_i содержит пункт $[A \to \alpha\cdot]$, а текущий входной символ a входит в FOLLOW(A). В некоторых ситуациях, однако, когда состояние i находится на вершине стека, допустимый префикс $\beta\alpha$ в стеке таков, что за βA ни в какой правосентенциальной форме не может следовать a. Таким образом, свертка в соответствии с продукцией $A \to \alpha$ оказывается некорректной при входном символе a.

Пример 4.36. Рассмотрим пример 4.34, в котором в состоянии 2 имелся пункт $R \to L\cdot$, соответствующий упомянутой выше продукции $A \to \alpha$, а a может быть символом = из множества FOLLOW(R). Таким образом, синтаксический анализатор SLR должен вызывать в состоянии 2 при очередном входном символе = свертку в соответствии с продукцией $R \to L$ (в связи с наличием в состоянии 2 пункта $S \to L\cdot = R$ может также использоваться перенос). Однако в грамматике из примера 4.34 не существует правосентенциальной формы, которая начинается с $R = \cdots$. Таким образом, в состоянии 2, соответствующем единственному допустимому префиксу L, не должна использоваться свертка этого L в R. □

Можно хранить в состоянии больший объем информации, который позволит отбрасывать такие некорректные свертки в соответствии с продукциями $A \to \alpha$. Разделяя при необходимости состояния, можно добиться того, что каждое состояние LR-анализатора будет точно указывать, какие входные символы могут следовать за основой α, для которой возможна свертка в A.

Дополнительная информация вносится в состояние путем такого переопределения пунктов, чтобы они включали в качестве второго компонента терминальный символ. Общим видом пункта становится $[A \to \alpha \cdot \beta, a]$, где $A \to \alpha\beta$ — продукция, а a — терминал или маркер конца входной строки \$. Такой объект называется LR(1)-*пунктом*. Здесь 1 означает длину второго компонента, именуемого

предпросмотром (lookahead) пункта.[9] Предпросмотр не влияет на пункт вида $[A \to \alpha \cdot \beta, a]$, где β не равно ϵ, но пункт $[A \to \alpha\cdot, a]$ приводит к свертке в соответствии с продукцией $A \to \alpha$, только если очередной входной символ равен a. Таким образом, свертка в соответствии с продукцией $A \to \alpha$ применяется только при входном символе a, для которого $[A \to \alpha\cdot, a]$ является LR(1)-пунктом из состояния на вершине стека. Множество таких a всегда является подмножеством FOLLOW(A), но может быть истинным подмножеством, как в примере 4.36.

Формально мы говорим, что LR(1)-пункт $[A \to \alpha \cdot \beta, a]$ *допустúм* (valid) для активного префикса γ, если существует порождение $S \overset{*}{\underset{rm}{\Rightarrow}} \delta A w \underset{rm}{\Rightarrow} \delta\alpha\beta w$, где

1) $\gamma = \delta\alpha$;

2) либо a является первым символом w, либо w — ϵ, а a — \$.

Пример 4.37. Рассмотрим грамматику

$$\begin{array}{lcl} S & \to & B\,B \\ B & \to & a\,B \mid b \end{array}$$

Существует правое порождение $S \overset{*}{\underset{rm}{\Rightarrow}} aaBab \underset{rm}{\Rightarrow} aaaBab$. Мы видим, что пункт $[B \to a \cdot B, a]$ допустúм для активного префикса $\gamma = aaa$, если положить в приведенном выше определении $\delta = aa$, $A = B$, $w = ab$, $\alpha = a$ и $\beta = B$. Существует также правое порождение $S \overset{*}{\underset{rm}{\Rightarrow}} BaB \underset{rm}{\Rightarrow} BaaB$. Из него видно, что пункт $[B \to a \cdot B, \$]$ является допустимым для активного префикса Baa. □

4.7.2 Построение множеств LR(1)-пунктов

Метод построения набора множеств допустимых LR(1)-пунктов, по сути, тот же, что и для построения канонического набора множеств LR(0)-пунктов. Нам надо только модифицировать две процедуры — CLOSURE и GOTO.

Чтобы разобраться в новом определении операции CLOSURE (в частности, почему b должно быть в FIRST(βa)), рассмотрим пункт вида $[A \to \alpha \cdot B\beta, a]$ в множестве пунктов, допустимых для некоторого активного префикса γ. Тогда существует правое порождение $S \overset{*}{\underset{rm}{\Rightarrow}} \delta A a x \underset{rm}{\Rightarrow} \delta\alpha B\beta a x$, где $\gamma = \delta\alpha$. Предположим, что $\beta a x$ порождает строку терминалов by. Тогда для каждой продукции вида $B \to \eta$ для некоторого η мы имеем порождение $S \overset{*}{\underset{rm}{\Rightarrow}} \gamma B b y \underset{rm}{\Rightarrow} \gamma\eta b y$. Таким образом, $[B \to \cdot\eta, b]$ является допустимым для γ. Заметим, что b может быть первым терминалом, порожденным из β, либо возможно, что β порождает ϵ в порождении $\beta a x \overset{*}{\underset{lm}{\Rightarrow}} by$, и, следовательно, b может представлять собой a. Подытоживая обе возможности, мы говорим, что b может быть любым терминалом в FIRST$(\beta a x)$, где

[9] Возможны предпросмотры длиной более 1, однако здесь мы их не рассматриваем.

```
SetOfItems CLOSURE(I) {
    repeat
        for ( каждый пункт [A → α · Bβ, a] из I )
            for ( каждая продукция B → γ из G' )
                for ( каждый терминал b ∈ FIRST(βα) )
                    Добавить [B → ·γ, b] в множество I;
    until нет больше пунктов для добавления в I;
    return I;
}

SetOfItems GOTO(I, X) {
    Инициализировать J пустым множеством;
    for ( каждый пункт [A → α · Xβ, a] из I )
        Добавить пункт [A → αX · β, a] в множество J;
    return CLOSURE(J);
}

void items(G') {
    Инициализировать C множеством {CLOSURE({[S' → ·S, $]})};
    repeat
        for ( каждое множество пунктов I в C )
            for ( каждый символ грамматики X )
                if ( GOTO(I, X) не пустое множество и не входит в C )
                    Добавить GOTO(I, X) в C;
    until нет новых множеств пунктов для добавления в C;
}
```

Рис. 4.40. Построение множеств LR(1)-пунктов для грамматики G'

FIRST — функция из раздела 4.4. Заметим, что x не может содержать первый терминал из by, так что FIRST$(\beta a x)$ = FIRST(βa). Приведем теперь метод построения множеств LR(1)-пунктов.

Алгоритм 4.38. Построение множеств LR(1)-пунктов

ВХОД: расширенная грамматика G'.

ВЫХОД: множества LR(1)-пунктов, которые представляют собой множество пунктов, допустимых для одного или нескольких активных префиксов G'.

МЕТОД: процедуры CLOSURE и GOTO и основная подпрограмма *items* для построения множеств пунктов были приведены на рис. 4.40. □

Пример 4.39. Рассмотрим следующую расширенную грамматику:

$$\begin{array}{lcl} S' & \rightarrow & S \\ S & \rightarrow & C\,C \\ C & \rightarrow & c\,C \mid d \end{array} \qquad (4.16)$$

Начнем с вычисления замыкания $\{[S' \rightarrow \cdot S, \$]\}$. В процедуре CLOSURE мы сопоставляем пункт $[S' \rightarrow \cdot S, \$]$ с пунктом $[A \rightarrow \alpha \cdot B\beta, a]$, т.е. $A = S'$, $\alpha = \epsilon$, $B = S$, $\beta = \epsilon$ и $a = \$$. Функция CLOSURE говорит нам, что следует добавить пункт $[B \rightarrow \cdot\gamma, b]$ для каждой продукции $B \rightarrow \gamma$ и терминала b из FIRST (βa). В терминах рассматриваемой грамматики $B \rightarrow \gamma$ должно быть $S \rightarrow C\,C$, а поскольку β равно ϵ, а a равно \$, b может быть только \$. Таким образом, мы добавляем $[S \rightarrow \cdot CC, \$]$.

Продолжим вычисление замыкания путем добавления всех пунктов $[C \rightarrow \cdot\gamma, b]$ для b из FIRST $(C\$)$, т.е. сопоставляя $[S \rightarrow \cdot CC, \$]$ с $[A \rightarrow \alpha \cdot B\beta, a]$ и получая $A = S$, $\alpha = \epsilon$, $B = C$, $\beta = C$ и $a = \$$. Поскольку C не порождает пустой строки, FIRST $(C\$)$ = FIRST (C), а поскольку FIRST (C) содержит терминалы c и d, мы добавляем пункты $[C \rightarrow \cdot cC, c]$, $[C \rightarrow \cdot cC, d]$, $[C \rightarrow \cdot d, c]$ и $[C \rightarrow \cdot d, d]$. Ни один из новых пунктов не имеет нетерминала непосредственно справа от точки, так что первое множество LR (1)-пунктов завершено. Начальное множество пунктов представляет собой

$$\begin{array}{ll} I_0: & S \rightarrow \cdot S, \$ \\ & S \rightarrow \cdot CC, \$ \\ & C \rightarrow \cdot cC, c/d \\ & C \rightarrow \cdot d, c/d \end{array}$$

Для удобства записи здесь опущены квадратные скобки, а запись $[C \rightarrow \cdot cC, c/d]$ представляет собой сокращенную запись двух пунктов — $[C \rightarrow \cdot cC, c]$ и $[C \rightarrow \cdot cC, d]$.

Теперь вычислим GOTO (I_0, X) для различных значений X. Для $X = S$ мы должны вычислить замыкание пункта $[S' \rightarrow S\cdot, \$]$. Никакие дополнительные замыкания невозможны, поскольку точка располагается крайней справа. Таким образом, мы получаем следующее множество пунктов:

$$I_1: \quad S' \rightarrow S\cdot, \$$$

Для $X = C$ вычисляем замыкание $[S \rightarrow C \cdot C, \$]$. Добавляем C-продукции со вторым компонентом \$, после чего не можем добавить ничего более и, таким образом, получаем

$$\begin{array}{ll} I_2: & S \rightarrow C \cdot C, \$ \\ & C \rightarrow \cdot cC, \$ \\ & C \rightarrow \cdot d, \$ \end{array}$$

Далее положим $X = c$. Теперь надо выполнить замыкание $\{[C \to c \cdot C, c/d]\}$. Добавляем C-продукции со вторым компонентом c/d, что дает

$$\begin{aligned} I_3: \quad & C \to c \cdot C, c/d \\ & C \to \cdot cC, c/d \\ & C \to \cdot d, c/d \end{aligned}$$

Наконец, полагая $X = d$, получаем множество пунктов

$$I_4: \quad C \to d \cdot, c/d$$

Этим завершается рассмотрение GOTO для I_0. Из I_1 новые множества не получаются, но зато I_2 имеет переходы для C, c и d. Для GOTO (I_2, C) получаем

$$I_5: \quad S \to CC \cdot, \$$$

Для него не требуется замыкания. Чтобы вычислить GOTO (I_2, c), берем замыкание $\{[C \to c \cdot C, \$]\}$ и получаем

$$\begin{aligned} I_6: \quad & C \to c \cdot C, \$ \\ & C \to \cdot cC, \$ \\ & C \to \cdot d, \$ \end{aligned}$$

Заметим, что I_6 отличается от I_3 только вторыми компонентами. Мы увидим, что это достаточно распространенное явление, когда несколько различных множеств LR(1)-пунктов грамматики имеют одни и те же первые компоненты и отличаются друг от друга своими вторыми компонентами. При построении наборов множеств LR(0)-пунктов для той же самой грамматики каждое множество LR(0)-пунктов совпадает с множеством первых компонентов одного или нескольких множеств LR(1)-пунктов. На этом явлении мы более подробно остановимся позже, при рассмотрении LALR-анализа.

Продолжая вычисление функции GOTO для I_2, получаем, что GOTO (I_2, d) имеет вид

$$I_7: \quad C \to d \cdot, \$$$

Перейдем теперь к I_3. Значениями функции GOTO для I_3 при входных символах c и d являются соответственно I_3 и I_4, а GOTO (I_3, C) равна

$$I_8: \quad C \to cC \cdot, c/d$$

I_4 и I_5 не имеют функций GOTO, поскольку все пункты в них содержат точки в крайнем положении справа. Значения GOTO для I_6 при входных символах c и d равны соответственно I_6 и I_7, а GOTO (I_6, C) равна

$$I_9: \quad C \to cC \cdot, \$$$

Остальные множества пунктов не дают значений GOTO, так что наша работа завершена. На рис. 4.41 показаны найденные десять множеств пунктов и их переходы. □

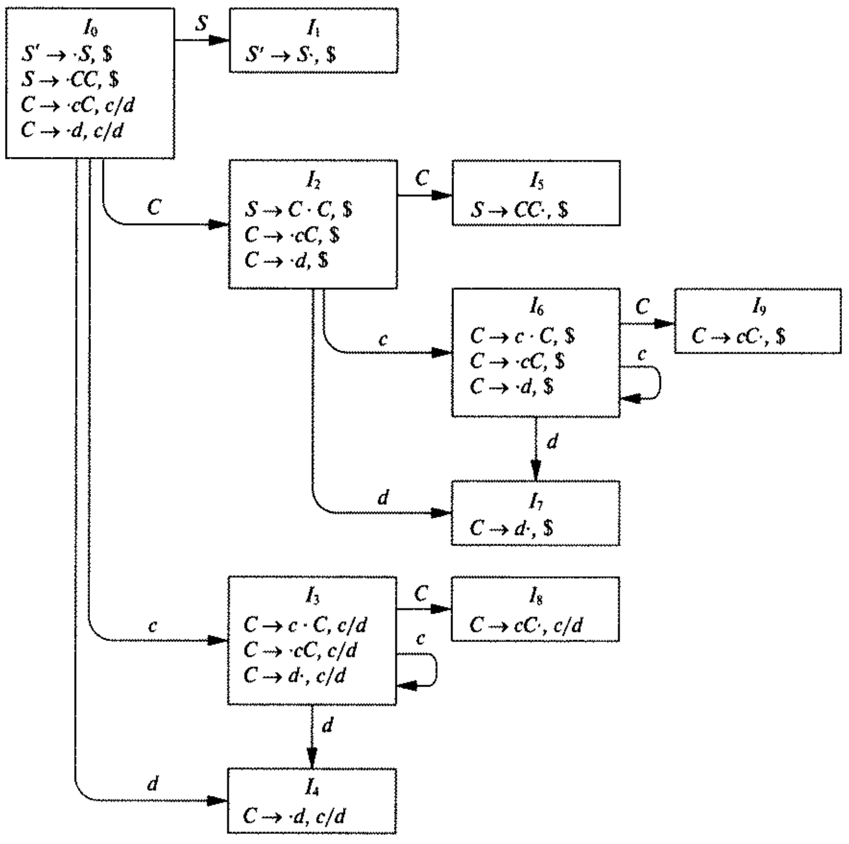


Рис. 4.41. Граф GOTO для грамматики (4.16)

4.7.3 Канонические таблицы LR(1)-анализа

Теперь приведем правила построения LR(1)-функций ACTION и GOTO из множеств LR(1)-пунктов. Эти функции, как и ранее, представлены таблицей. Единственное отличие — в значениях записей таблицы.

Алгоритм 4.40. Построение таблиц канонического LR-анализа

Вход: расширенная грамматика G'.

Выход: таблица канонического LR-анализа с функциями ACTION и GOTO для грамматики G'.

Метод: выполнить следующие действия.

1. Построить $C' = \{I_0, I_1, \ldots, I_n\}$ — набор множеств LR (1)-пунктов для G'.

2. Состояние синтаксического анализатора строится из I_i. Действие синтаксического анализа для состояния i определяется следующим образом.

   а) Если $[A \to \alpha \cdot a\beta, b]$ входит в I_i и $\text{GOTO}(I_i, a) = I_j$, установить $\text{ACTION}[i, a]$ равным "перенос j". Здесь a должно быть терминалом.

   б) Если $[A \to \alpha\cdot, a]$ входит в I_i и $A \neq S'$, то установить $\text{ACTION}[i, a]$ равным "свертка $A \to \alpha$".

   в) Если $[S' \to S\cdot, \$]$ входит в I_i, установить $\text{ACTION}[i, \$]$ равным "принятие".

   Если при применении указанных правил обнаруживаются конфликтующие действия, грамматика не принадлежит классу LR (1).

3. Переходы для состояния i строятся для всех нетерминалов A с использованием следующего правила: если $\text{GOTO}(I_i, A) = I_j$, то $\text{GOTO}[i, A] = j$.

4. Все записи, не определенные правилами (2) и (3), считаются записями "ошибка".

5. Начальное состояние синтаксического анализатора — состояние, построенное из множества пунктов, содержащего $[S' \to S\cdot, \$]$. □

Таблица, образованная функциями действий и переходов, полученными при помощи алгоритма 4.40, называется *канонической* таблицей LR (1)-анализа. LR-анализатор, использующий эту таблицу, называется каноническим LR (1)-синтаксическим анализатором. Если функция действий синтаксического анализа не имеет многократно определенных записей, то соответствующая грамматика называется LR (1)-*грамматикой*. Как и ранее, (1) опускается везде, где оно очевидно из контекста.

Пример 4.41. Каноническая таблица синтаксического анализа для грамматики (4.16) показана на рис. 4.42. Продукциями 1, 2 и 3 являются соответственно продукции $S \to C\,C$, $C \to c\,C$ и $C \to d$. □

Каждая SLR (1)-грамматика является LR (1)-грамматикой, но канонический LR-анализатор для SLR (1)-грамматики может иметь большее количество состояний, чем SLR-анализатор для той же грамматики. Грамматика из предыдущего примера является SLR-грамматикой и имеет SLR-анализатор с семью состояниями (сравните с десятью состояниями на рис. 4.42).

Состояние	ACTION			GOTO	
	c	d	$	S	C
0	s3	s4		1	2
1			acc		
2	s6	s7			5
3	s3	s4			8
4	r1	r3			
5			r1		
6	s6	s7			9
7			r3		
8	r2	r2			
9			r2		

Рис. 4.42. Каноническая таблица синтаксического анализа для грамматики (4.16)

4.7.4 Построение LALR-таблиц синтаксического анализа

Перейдем к последнему из рассматриваемых нами методов построения синтаксических анализаторов, а именно — к методу LALR (*LR с предпросмотром* — lookahead LR). Этот метод часто используется на практике из-за того, что получаемые при его использовании таблицы получаются сравнительно небольшими по сравнению с каноническими LR-таблицами, а также поскольку большинство распространенных синтаксических конструкций языков программирования могут быть легко выражены LALR-грамматикой. Практически то же самое можно сказать и о SLR-грамматиках, но в этом случае имеется несколько конструкций, которые не могут быть легко обработаны при помощи метода SLR (см., в частности, пример 4.34).

Сравнивая размеры синтаксических анализаторов, можно сказать, что SLR- и LALR-таблицы для грамматики почти всегда имеют одно и то же количество состояний, и это количество обычно составляет несколько сотен состояний для

языков наподобие C[10]. Каноническая LR-таблица для языка такого типа обычно содержит несколько тысяч состояний. Таким образом, построить SLR- или LARL-таблицы существенно проще и экономичнее, чем канонические LR-таблицы.

В качестве введения обратимся вновь к грамматике (4.16), множества LR (1)-пунктов которой были приведены на рис. 4.41. Возьмем пару похожих состояний, таких как I_4 и I_7. Эти состояния содержат только пункты с первым компонентом $C \to d\cdot$. В I_4 предпросматриваемыми символами могут быть c и d, а в I_7 — только $.

Чтобы увидеть разницу между I_4 и I_7 в синтаксическом анализаторе, обратите внимание, что грамматика (4.16) порождает регулярный язык $\mathbf{c^*dc^*d}$. При считывании входного потока $cc \cdots cdcc \cdots cd$ синтаксический анализатор переносит первую группу символов c и следующий за ними d в стек, попадая после считывания символа d в состояние 4. Затем синтаксический анализатор вызывает свертку по продукции $C \to d$, обусловленную следующим входным символом c или d. Требование, чтобы следующим входным символом был c или d, имеет смысл, поскольку это символы, с которых может начинаться строка $\mathbf{c^*d}$. Если после первого d следует $, то получается входной поток наподобие ccd, который не принадлежит рассматриваемому языку, и состояние 4 совершенно справедливо обнаруживает ошибку при очередном входном символе $.

В состояние 7 синтаксический анализатор попадает после чтения второго d. Соответственно, синтаксический анализатор должен обнаружить во входном по-

[10] Вот некоторые сравнительные характеристики реальных языков программирования.

Язык	Количество			
	Токены	Продукций (непустых)	Терминалы	Ключевые слова
Algol-60	890	102(90)	88	24
Pascal	1004	109(85)	84	35
Modula-2	887	70(69)	88	39
Ada 95	2999	327(258)	98	69
Turbo Pascal 6.0	1479	141(117)	89	55
Delphi 7.0	2041	186(165)	92	107
C (K&R)	913	52(49)	122	27
C99	1413	110(106)	133	37
C++ (Страуструп, 1990)	1654	124(117)	131	48
C++ (ISO/IEC 14882-1998)	2292	176(166)	136	63
Java	1813	172(158)	121	48
C#	3036	295(268)	115	88

Для ознакомления со сравнительным анализом различных языков программирования можно обратиться к книге Свердлов С.З. *Языки программирования и методы трансляции: Учебное пособие.* СПб.: Питер, 2007. — 638 с. — *Прим. ред.*

токе символ \$, иначе входная строка не соответствует регулярному выражению **c*dc*d**. Таким образом, состояние 7 должно приводить к свертке $C \to d$ при входном символе \$ и к ошибке при входном символе c или d.

Заменим теперь состояния I_4 и I_7 состоянием I_{47}, которое представляет собой объединение I_4 и I_7, состоящее из трех пунктов — $[C \to d\cdot, c/d/\$]$. Все переходы по d в I_4 или I_7 из I_0, I_2, I_3 и I_6 ведут теперь в I_{47}. Действие в состоянии 47 — свертка при любом входном символе. Такой синтаксический анализатор в целом ведет себя так же, как исходный, хотя и может свернуть d в C в условиях, когда исходный синтаксический анализатор объявил бы об ошибке, например при входной строке ccd или $cdcdc$. В конце концов ошибка будет обнаружена — перед тем как мы получим любой символ, вызывающий перенос.

Обобщая, мы можем рассмотреть множества LR (1)-пунктов, имеющих одно и то же *ядро* (core), т.е. множество первых компонентов, и объединить эти множества с общими ядрами в одно множество пунктов. Например, на рис. 4.41 такую пару с ядром $\{C \to d\cdot\}$ образуют состояния I_4 и I_7. Аналогично множества I_3 и I_6 образуют другую пару — с ядром $\{C \to c \cdot C, C \to \cdot cC, C \to \cdot d\}$. Имеется и еще одна пара — I_8 и I_9 — с общим ядром $\{C \to cC\cdot\}$. Заметим, что, вообще говоря, ядро является множеством LR (0)-пунктов рассматриваемой грамматики и что LR (1)-грамматика может давать более двух множеств пунктов с одним и тем же ядром.

Поскольку ядро множества GOTO (I, X) зависит только от ядра множества I, значения функции GOTO объединяемых множеств также могут быть объединены. Таким образом, проблем вычисления функции GOTO при слиянии множеств не возникает. Функция же ACTION должна быть изменена, чтобы отражать не ошибочные действия всех объединяемых множеств пунктов.

Предположим, имеется LR (1)-грамматика, т.е. грамматика, множества LR (1)-пунктов которой не вызывают конфликтов действий синтаксического анализа. Если заменить все состояния, имеющие одно и то же ядро, их объединениями, возможно, полученное в результате состояние будет иметь конфликт, хотя это маловероятно по следующей причине. Предположим, что в объединении возникает конфликт при просмотре входного символа a, поскольку существует пункт $[A \to \alpha\cdot, a]$, вызывающий свертку по продукции $A \to \alpha$, а также другой пункт, $[B \to \beta \cdot a\gamma, b]$, приводящий к переносу. Тогда некоторое множество пунктов, из которого было сформировано объединение, имело пункт $[A \to \alpha\cdot, a]$. Поскольку ядра объединяемых множеств совпадают, это множество должно также иметь пункт $[B \to \beta \cdot a\gamma, c]$ для некоторого c. Но в таком случае это состояние имело бы конфликт переноса/свертки для символа a и вопреки нашему предположению грамматика не была бы LR (1)-грамматикой. Следовательно, объединение состояний с одинаковыми ядрами не может привести к конфликту переноса/свертки, если такой конфликт не присутствовал ни в одном из исходных состояний (поскольку переносы зависят только от ядра, но не от предпросмотра).

Тем не менее при объединении возможно появление конфликта "свертка/свертка", как показано в следующем примере.

Пример 4.42. Рассмотрим грамматику

$$\begin{aligned} S' &\to S \\ S &\to a\,A\,d \mid b\,B\,d \mid a\,B\,e \mid b\,A\,e \\ A &\to c \\ B &\to c \end{aligned}$$

Она генерирует четыре строки — *acd*, *ace*, *bcd* и *bce*. Читатель может убедиться, что грамматика является LR (1), построив множества пунктов. Сделав это, мы обнаружим множество пунктов $\{[A \to c\cdot, d], [B \to c\cdot, e]\}$, допустимых для активного префикса *ac*, и $\{[A \to c\cdot, e], [B \to c\cdot, d]\}$ — для префикса *bc*. Ни одно из этих множеств не вызывает конфликта; ядра их одинаковы. Однако их объединение

$$\begin{aligned} A &\to c\cdot, d/e \\ B &\to c\cdot, d/e \end{aligned}$$

вызывает конфликт "свертка/свертка", поскольку при входных символах d и e вызываются две свертки: $A \to c$ и $B \to c$. □

Теперь мы готовы рассмотреть первый из двух алгоритмов построения LALR-таблиц. Основная идея состоит в создании множеств LR (1)-пунктов и, если это не вызывает конфликтов, объединении множеств с одинаковыми ядрами. Затем на базе набора множеств пунктов строим таблицу синтаксического анализа. Описываемый метод служит, в первую очередь, определением LALR (1)-грамматик. Построение полного набора множеств LR (1)-пунктов требует слишком много памяти и времени, чтобы использоваться на практике.

Алгоритм 4.43. Простое построение LALR-таблицы (с большими затратами памяти)

Вход: расширенная грамматика G'.

Выход: таблица функций LALR-анализа ACTION и GOTO для грамматики G'.

Метод: выполняем следующие действия.

1. Строим набор множеств LR (1)-пунктов $C = \{I_0, I_1, \ldots, I_n\}$ для грамматики G'.

2. Для каждого ядра, имеющегося среди множества LR (1)-пунктов, находим все множества, имеющие это ядро, и заменяем эти множества их объединением.

3. Пусть $C' = \{J_0, J_1, \ldots, J_m\}$ — полученные в результате множества LR (1)-пунктов. Функцию ACTION для состояния i строим из J_i так же, как и в алгоритме 4.40. Если при этом обнаруживается конфликт, алгоритм не в состоянии построить синтаксический анализатор, а грамматика не является LALR (1)-грамматикой.

4. Таблица GOTO строим следующим образом. Если J — объединение одного или нескольких множеств LR (1)-пунктов, т.е. $J = I_1 \cup I_2 \cup \cdots \cup I_k$, то ядра множеств $\text{GOTO}(I_1, X)$, $\text{GOTO}(I_2, X)$, $\ldots$, $\text{GOTO}(I_k, X)$ одни и те же, поскольку $I_1, I_2, \ldots, I_k$ имеют одно и то же ядро. Обозначим через K объединение всех множеств пунктов, имеющих то же ядро, что и $\text{GOTO}(I_1, X)$. Тогда $\text{GOTO}(J, X) = K$. □

Таблица, полученная при помощи алгоритма 4.43, называется *таблицей LALR-анализа* для грамматики G. Если конфликты действий отсутствуют, то данная грамматика называется LALR (1)-*грамматикой*. Набор множеств пунктов, построенный на шаге 3, называется LALR (1)-набором.

Пример 4.44. Вновь обратимся к грамматике (4.16), граф GOTO которой показан на рис. 4.41. Как уже упоминалось, имеется три пары множеств пунктов, которые могут быть объединены. I_3 и I_6 заменяются их объединением

$$\begin{aligned} I_{36}: \quad & C \to c \cdot C, c/d/\$ \\ & C \to \cdot cC, c/d/\$ \\ & C \to \cdot d, c/d/\$ \end{aligned}$$

I_4 и I_7 заменяются их объединением

$$I_{47}: \quad C \to d\cdot, c/d/\$$$

I_8 и I_9 заменяются их объединением

$$I_{89}: \quad C \to cC\cdot, c/d/\$$$

Функции действий и переходов LALR для объединенных множеств пунктов показаны на рис. 4.43.

Чтобы увидеть, каким образом вычисляется функция GOTO, рассмотрим $\text{GOTO}(I_{36}, C)$. В исходном множестве LR (1)-пунктов $\text{GOTO}(I_3, C) = I_8$, а I_8 теперь является частью I_{89}, так что мы определяем, что $\text{GOTO}(I_{36}, C) = I_{89}$. Тот же вывод моно сделать, рассматривая I_6 — вторую часть I_{36}: $\text{GOTO}(I_6, C) = I_9$, а I_9 теперь также является частью I_{89}. В качестве другого примера рассмотрим $\text{GOTO}(I_2, c)$, переход, выполняемый после переноса состояния I_2 при входном символе c. В исходных множествах LR (1)-пунктов $\text{GOTO}(I_2, c) = I_6$. Поскольку

Состояние	ACTION			GOTO	
	c	d	$	S	C
0	s36	s47		1	2
1			acc		
2	s36	s47			5
36	s36	s47			89
47	r3	r3	r3		
5			r1		
89	r2	r2	r2		

Рис. 4.43. Таблица LALR-анализа для грамматики из примера 4.39

теперь I_6 является частью I_{36}, $\text{GOTO}(I_2, c)$ становится равным I_{36}; таким образом, запись на рис. 4.43 для состояния 2 и входного символа c – s36, что означает перенос и помещение в стек состояния 36. □

При входной строке из языка **c*dc*d** как LR-анализатор на рис. 4.42, так и LALR-анализатор на рис. 4.43 выполняют одну и ту же последовательность переносов и сверток, хотя имена состояний в стеке могут отличаться. Так, если LR-анализатор помещает в стек I_3 или I_6, LALR-анализатор помещает в стек I_{36}. Это верно и в общем случае LALR-грамматики. LR- и LALR-анализаторы имитируют друг друга при корректной входной строке.

Однако при строке с ошибками LALR-анализатор может выполнить несколько сверток после того, как LR-анализатор уже объявит об ошибке. Однако LALR-анализатор никогда не перенесет символ после того, как ошибка распознана LR-анализатором. Например, для входной строки $ccd\$$ LR-анализатор на рис. 4.42 поместит в стек

$$0\ 3\ 3\ 4$$

и в состоянии 4 обнаружит ошибку, поскольку следующий входной символ – $, а для состояния 4 соответствующая запись таблицы ACTION – "ошибка". В противоположность этому LALR-анализатор, показанный на рис. 4.43, выполняет соответствующие действия, помещая в стек

$$0\ 36\ 36\ 47$$

Однако состояние 47 при входном символе $ приводит к свертке $C \to d$. Таким образом, LALR-анализатор изменит содержимое стека на

$$0\ 36\ 36\ 89$$

Действие в состоянии 89 для входного символа \$ — свертка $C \to cC$, после чего содержимое стека приобретает вид

$$0\ 36\ 89$$

После этого та же свертка выполняется повторно, что приводит к содержимому стека

$$0\ 2$$

Наконец, мы обнаруживаем ошибку в соответствии с записью ACTION для состояния 2 и входного символа \$.

4.7.5 Эффективное построение таблиц LALR-анализа

Имеется несколько изменений, которые можно внести в алгоритм 4.43, чтобы избежать построения полного набора множеств LR(1)-пунктов в процессе создания таблицы LALR(1)-анализа.

- Можно представить любое множество LR(0)- или LR(1)-пунктов I его ядром, т.е. теми пунктами, которые либо являются начальным пунктом ($[S' \to \cdot S]$ или $[S' \to \cdot S, \$]$), либо содержат точку не в начале тела продукции.
- Можно построить ядра LALR(1)-пунктов на основе ядер LR(0)-пунктов при помощи процесса распространения и спонтанной генерации символов предпросмотра, который будет описан ниже.
- Если имеются ядра LALR(1), то можно сгенерировать LALR(1)-таблицу путем замыкания каждого ядра с использованием функции CLOSURE из рис. 4.40, а затем вычислить записи таблицы при помощи алгоритма 4.40, как если бы LALR(1)-множества пунктов были каноническими LR(1)-множествами пунктов.

Пример 4.45. Используем в качестве примера метода эффективного построения LALR(1)-таблицы грамматику из примера 4.34, не являющуюся SLR-грамматикой, которую мы повторим здесь в расширенном виде:

$$\begin{array}{lcl} S' & \to & S \\ S & \to & L = R \mid R \\ L & \to & *R \mid \mathbf{id} \\ R & \to & L \end{array}$$

Полные множества LR(0)-пунктов для этой грамматики были приведены на рис. 4.39. Ядра этих пунктов показаны на рис. 4.44. □

$$
\begin{array}{llll}
I_0: & S' \to \cdot S & I_5: & L \to \mathbf{id}\cdot \\
I_1: & S' \to S\cdot & I_6: & S \to L = \cdot R \\
I_2: & S \to L\cdot = R & I_7: & L \to *R\cdot \\
 & R \to L\cdot & & \\
 & & I_8: & R \to L\cdot \\
I_3: & S \to R\cdot & & \\
 & & I_9: & S \to L = R\cdot \\
I_4: & L \to * \cdot R & &
\end{array}
$$

Рис. 4.44. Ядра множеств LR (0)-пунктов для грамматики (4.15)

Теперь для создания ядер множеств LALR (1)-пунктов требуется назначить корректные предпросматриваемые символы LR (0)-пунктам ядер. Есть два способа, которыми предпросматриваемый символ b может быть назначен LR (0)-пункту $B \to \gamma \cdot \delta$ из некоторого множества LALR (1)-пунктов J.

1. Существует множество пунктов I с базисным пунктом $A \to \alpha \cdot \beta, a$, $J = \text{GOTO}(I, X)$, и построение

$$\text{GOTO}\left(\text{CLOSURE}\left(\{[A \to \alpha \cdot \beta, a]\}\right), X\right)$$

содержит $[B \to \gamma \cdot \delta, \beta]$ безотносительно к a, как показано на рис. 4.40. Такой символ b называется *спонтанно* (spontaneously) сгенерированным для $B \to \gamma \cdot \delta$.

2. В качестве частного случая предпросматриваемый символ \$ спонтанно генерируется для пункта $S' \to \cdot S$ в начальном множестве пунктов.

3. Как и в случае 1, но $a = b$ и, как показано на рис. 4.40,

$$\text{GOTO}\left(\text{CLOSURE}\left(\{[A \to \alpha \cdot \beta, b]\}\right), X\right)$$

содержит $[B \to \gamma \cdot \delta, b]$ только потому, что одним из связанных с $A \to \alpha \cdot \beta$ предпросматриваемых символов является b. В этом случае мы говорим о *распространении* (propagate) символа предпросмотра от $A \to \alpha \cdot \beta$ в ядре I к $B \to \gamma \cdot \delta$ в ядре J. Заметим, что распространение не зависит от конкретного символа предпросмотра; либо все символы предпросмотра распространяются от одного пункта к другому, либо ни один из них.

Нам требуется определить спонтанно генерируемые символы предпросмотра для каждого множества LR (0)-пунктов, а также определить, для каких пунктов происходит распространение предпросмотров и из каких именно пунктов. Это достаточно просто. Пусть # — символ, отсутствующий в рассматриваемой грамматике, и пусть $A \to \alpha \cdot \beta$ — ядро LR (0)-пункта в множестве I. Для каждого X вычислим $J = \text{GOTO}(\text{CLOSURE}(\{[A \to \alpha \cdot \beta, \#]\}), X)$. Для каждого базисного пункта в J проверим его множество символов предпросмотра. Если # является символом предпросмотра, то предпросмотры распространяются к этому пункту от $A \to \alpha \cdot \beta$. Все прочие предпросмотры генерируются спонтанно. Эти идеи строго изложены в приведенном далее алгоритме, который, кроме того, использует тот факт, что только в базисных пунктах J точка непосредственно следует за X, так что они должны иметь вид $B \to \gamma X \cdot \delta$.

Алгоритм 4.46. Определение предпросмотров

Вход: ядро K множества LR (0)-пунктов I и символ грамматики X.

Выход: спонтанно генерируемые пунктами из I предпросмотры для базисных пунктов в $\text{GOTO}(I, X)$, а также пункты из I, от которых предпросмотры распространяются к базисным пунктам в $\text{GOTO}(I, X)$.

Метод: алгоритм, показанный на рис. 4.45. □

```
for ( каждый пункт A → α · β из K ) {
    J := CLOSURE({[A → α · β, #]}) );
    if ( [B → γ · Xδ, a] ∈ J и a ≠ # )
        Заключаем, что символ предпросмотра a спонтанно генерируется
            для пункта B → γX · δ в GOTO(I, X);
    if ( [B → γ · Xδ, #] ∈ J )
        Заключаем, что символы предпросмотра распространяются
            от A → α · β из I к B → γX · δ в GOTO(I, X);
}
```

Рис. 4.45. Выявление спонтанно генерируемых и распространяемых предпросмотров

Теперь мы готовы к назначению предпросмотров ядрам множеств LR (0)-пунктов для формирования множеств LALR (1)-пунктов. Мы знаем, что \$ является символом предпросмотра для $S' \to \cdot S$ в исходном множестве LR (0)-пунктов. Алгоритм 4.46 дает нам все спонтанно генерируемые символы предпросмотра. После того как будут перечислены все такие символы предпросмотра, мы должны позволить им распространяться до тех пор, пока не останется ни одного возможного распространения. Существует много различных подходов, которые в определенном смысле отслеживают "новые" символы предпросмотра, которые распространились на некоторый пункт, но пока что не распространились далее. Приведенный

далее алгоритм описывает один из возможных методов распространения символов предпросмотра на все пункты.

Алгоритм 4.47. Эффективное вычисление ядер наборов множеств LALR (1)-пунктов

Вход: расширенная грамматика G'.

Выход: ядра наборов множеств LALR (1)-пунктов для грамматики G'.

Метод: выполним следующие действия.

1. Построим ядра множеств LR (0)-пунктов для G. Если количество используемой памяти не является главным, то простейший метод состоит в построении LR (0) множеств пунктов, как в разделе 4.6.2, с последующим удалением пунктов, не являющихся базисными. Если же память ограничена, то вместо этого можно сохранять для каждого множества только базисные пункты и вычислять GOTO для множества пунктов I, сначала вычисляя замыкание I.

2. Применим алгоритм 4.46 к ядру каждого множества LR (0)-пунктов и грамматическому символу X, чтобы определить, какие символы предпросмотра спонтанно генерируются для базисных пунктов в GOTO (I, X) и из каких пунктов I символы предпросмотра распространяются на базисные пункты GOTO (I, X).

3. Инициализируем таблицу, которая для каждого базисного пункта в каждом множестве пунктов дает связанные с ними предпросмотры. Изначально с каждым пунктом связаны только те предпросмотры, которые в п. 2 определены как сгенерированные спонтанно.

4. Повторим проходы по базисным пунктам во всех множествах. При посещении пункта i мы с помощью таблицы, построенной в п. 2, ищем базисные пункты, на которые i распространяет свои предпросмотры. Текущее множество предпросмотров для i добавляется к множествам, связанным с каждым из пунктов, на которые i распространяет свои предпросмотры. Мы продолжаем выполнять такие проходы по базисным пунктам до тех пор, пока не останется новых предпросмотров для распространения. □

Пример 4.48. Построим ядра LALR (1)-пунктов для грамматики из примера 4.45. Ядра LR (0)-пунктов были показаны на рис. 4.44. Применяя алгоритм 4.46 к ядру множества пунктов I_0, мы сначала вычисляем CLOSURE $(\{[S' \to \cdot S, \#]\})$, которое представляет собой

$$\begin{array}{ll} S' \to \cdot S, \# & L \to \cdot * R, \#/= \\ S \to \cdot L = R, \# & L \to \cdot \mathbf{id}, \#/= \\ S \to \cdot R, \# & R \to \cdot L, \# \end{array}$$

Среди пунктов в этом замыкании имеются два, у которых символ предпросмотра = генерируется спонтанно. Первый из них — $L \to \cdot * R$. Этот пункт, у которого справа от точки находится *, приводит к $[L \to * \cdot R, =]$, т.е. = — спонтанно сгенерированный символ предпросмотра для $L \to * \cdot R$, представляющего собой множество пунктов I_4. Аналогично $[L \to \cdot \mathbf{id}, =]$ говорит нам о том, что = — спонтанно сгенерированный символ предпросмотра для $L \to \mathbf{id}\cdot$ из I_5.

Поскольку # является символом предпросмотра для всех шести пунктов в замыкании, мы определяем, что пункт $S' \to \cdot S$ в I_0 распространяет символы предпросмотра на следующие шесть пунктов:

$$
\begin{array}{ll}
S' \to S \cdot \text{ в } I_1 & L \to * \cdot R \text{ в } I_4 \\
S \to L \cdot = R \text{ в } I_2 & L \to \mathbf{id} \cdot \text{ в } I_5 \\
S \to R \cdot \text{ в } I_3 & R \to L \cdot \text{ в } I_2
\end{array}
$$

От	К
$I_0: \quad S' \to \cdot S$	$I_1: \quad S' \to S \cdot$
	$I_2: \quad S \to L \cdot = R$
	$I_2: \quad R \to L \cdot$
	$I_3: \quad S \to R \cdot$
	$I_4: \quad L \to * \cdot R$
	$I_5: \quad L \to \mathbf{id} \cdot$
$I_2: \quad S \to L \cdot = R$	$I_6: \quad S \to L = \cdot R$
$I_4: \quad L \to * \cdot R$	$I_4: \quad L \to * \cdot R$
	$I_5: \quad L \to \mathbf{id} \cdot$
	$I_7: \quad L \to * R \cdot$
	$I_8: \quad R \to L \cdot$
$I_6: \quad S \to L = \cdot R$	$I_4: \quad L \to * \cdot R$
	$I_5: \quad L \to \mathbf{id} \cdot$
	$I_8: \quad R \to L \cdot$
	$I_9: \quad S \to L = R \cdot$

Рис. 4.46. Распространение символов предпросмотра

На рис. 4.47 показаны шаги 3 и 4 алгоритма 4.47. Столбец "Изначально" указывает спонтанно сгенерированные символы предпросмотра для каждого базисного пункта. Здесь только два раза встречается рассмотренный ранее символ = и спонтанно сгенерированный символ предпросмотра \$ для начального пункта $S' \to \cdot S$.

Множество	Пункт	Символы предпросмотра			
		Изначально	Проход 1	Проход 2	Проход 3
I_0 :	$S' \to \cdot S$	\$	\$	\$	\$
I_1 :	$S' \to S\cdot$		\$	\$	\$
I_2 :	$S \to L\cdot = R$		\$	\$	\$
	$R \to L\cdot$		\$	\$	\$
I_3 :	$S \to R\cdot$		\$	\$	\$
I_4 :	$L \to * \cdot R$	=	=/\$	=/\$	=/\$
I_5 :	$L \to \mathbf{id}\cdot$	=	=/\$	=/\$	=/\$
I_6 :	$S \to L = \cdot R$			\$	\$
I_7 :	$L \to *R\cdot$		=	=/\$	=/\$
I_8 :	$R \to L\cdot$		=	=/\$	=/\$
I_9 :	$S \to L = R\cdot$				\$

Рис. 4.47. Вычисление символов предпросмотра

При первом проходе предпросмотр \$ распространяется от $S' \to \cdot S$ в I_0 к шести пунктам, перечисленным на рис. 4.46. Предпросмотр = распространяется от $L \to * \cdot R$ в I_4 к пунктам $L \to *R\cdot$ в I_7 и к $R \to L\cdot$ в I_8. Кроме того, он также распространяется к самому себе и к $L \to \mathbf{id}\cdot$ из I_5, но эти предпросмотры уже есть. При втором и третьем проходах единственный распространяемый предпросмотр — \$ (находится для I_2 и I_4 при втором проходе и для I_6 — при третьем). При четвертом проходе распространения не происходит, и окончательное множество предпросмотров показано в последнем столбце на рис. 4.47.

Обратите внимание, что конфликт "перенос/свертка", обнаруженный в примере 4.34 при использовании SLR-метода, исчезает при использовании LALR-технологии. Причина в том, что с $R \to L\cdot$ в I_2 связан только предпросмотр \$, так что нет конфликта с переносом =, генерируемым пунктом $S \to L\cdot = R$ из I_2. □

4.7.6 Уплотнение таблиц LR-анализа

Грамматика типичного языка программирования с 50–100 терминалами и 100 продукциями может иметь таблицу LALR-анализа с несколькими сотнями состояний. Функция ACTION может легко иметь 20 000 записей, каждая из которых требует минимум 8 бит. Очевидно, что немаловажной задачей является поиск более эффективного, по сравнению с двумерной таблицей, представления информации. В этом разделе будут вкратце рассмотрены некоторые технологии, используемые для сжатия полей ACTION и GOTO таблицы LR-анализа.

Одна из таких технологий уплотнения части ACTION таблицы основана на том, что обычно в таблице многие строки идентичны. Так, на рис. 4.42 состояния 0 и 3 имеют одинаковые записи действий; то же самое можно сказать и о состояниях 2 и 6. Следовательно, память можно сэкономить (ценой небольшого повышения времени работы), создавая указатель в одномерный массив для каждого состояния. Указатели для состояний с одними и теми же действиями указывают на одно и то же место. Для получения информации из этого массива присвоим каждому терминалу номер — от нуля до числа, на единицу меньшего количества терминалов, и используем затем это целое число как смещение от значения указателя для каждого состояния. Для данного состояния действие синтаксического анализа для i-го терминала представляет собой i-ю запись после той, на которую указывает упомянутый указатель для данного состояния.

Дальнейшее повышение эффективности использования памяти достигается путем создания списка действий для каждого состояния. Список состоит из пар (терминальный символ, действие). Наиболее часто встречающиеся действия для данного состояния могут быть размещены в конце списка; в списке можно использовать специальный терминал "любой", который означает, что если в списке не найден текущий входной символ, то действие синтаксического анализатора выполняется независимо от входного символа. Кроме того, записи ошибок можно для единообразия заменить записями сверток — это не повлияет на выявление ошибок, а просто отложит его до использования переноса.

Пример 4.49. Рассмотрим таблицу синтаксического анализа на рис. 4.37. Начнем с того, что заметим, что действия для состояний 0, 4, 6 и 7 совпадают. Мы можем представить их в виде списка:

Символ	Действие
id	s5
(	s4
любой	ошибка

Состояние 1 имеет похожий список:

+	s6
$	acc
любой	ошибка

В состоянии 2 мы можем заменить записи ошибок действием r2, так что для любого символа, кроме *, будет выполняться свертка по продукции 2.

*	s7
любой	r2

Состояние 3 имеет только записи ошибок и свертки r4, поэтому можем заменить первые записи последними, и список для состояния 3 будет содержать только одну пару — (**любой**, r4). Состояния 5, 10 и 11 могут рассматриваться аналогично. Список для состояния 8 представляет собой

+	s6
)	s11
любой	ошибка

а список для состояния 9 —

*	s7
любой	r1

□

Мы можем закодировать в список и таблицу GOTO, но более эффективный путь состоит в создании списка пар для каждого нетерминала A. Каждая пара в списке для A имеет вид (*текущее состояние*, *следующее состояние*), указывающий, что

$$\text{GOTO}\,[\textit{текущее состояние}, A] = \textit{следующее состояние}$$

Такой способ предпочтительнее по той причине, что обычно в одном столбце таблицы GOTO имеется только небольшое количество состояний. Это связано с тем, что GOTO для нетерминала A может быть только состоянием, порождаемым из множества пунктов, в котором у некоторых пунктов символ A расположен непосредственно слева от точки. Ни одно множество не имеет пунктов с X и Y непосредственно слева от точки, если $X \neq Y$. Следовательно, каждое состояние появляется не более чем в одном столбце таблицы переходов.

Для еще большей экономии памяти заметим, что к записям ошибок в таблице GOTO никогда не производится обращений. Такую запись можно заменить наиболее часто встречающейся неошибочной записью в том же столбце. Эта запись становится записью по умолчанию и представляется в списке одной парой со специальным текущим состоянием "любое".

Пример 4.50. Вновь рассмотрим рис. 4.37. Столбец F имеет запись 10 для состояния 7, а все остальные записи — либо 3, либо "ошибка". Мы можем заменить ошибку третьим состоянием и создать для столбца F следующий список:

Текущее состояние	Следующее состояние
7	10
любое	3

Аналогично список для столбца T представляет собой

6	9
любое	2

Для столбца E можно выбрать в качестве значения по умолчанию либо 1, либо 8; в любом случае необходимы две записи. Например, можно создать для столбца E следующий список:

4	8
любое	1

Экономия памяти в этих маленьких примерах слишком мала. Если читатель подсчитает число записей в списках, созданных в этом и предыдущем примерах, а затем добавит указатели от состояний в списки действий и указатели от нетерминалов в списки следующих состояний, то экономия памяти по сравнению с матричной реализацией на рис. 4.37 не произведет на него особого впечатления. В реальных же грамматиках память, необходимая для представления в виде списков, обычно составляет менее 10% от памяти, требуемой для матричного представления.

Следует также отметить, что методы сжатия таблиц конечных автоматов, рассмотренные в разделе 3.9.8, могут использоваться и для представления таблиц LR-анализа.

4.7.7 Упражнения к разделу 4.7

Упражнение 4.7.1. Постройте для грамматики $S \to S\,S+ \mid S\,S* \mid a$ из упражнения 4.2.1 следующие множества пунктов:

а) каноническое LR;

б) LALR.

Упражнение 4.7.2. Повторите упражнение 4.7.1 для каждой из (расширенных) грамматик из упражнений 4.2.2, *а–ж*.

! **Упражнение 4.7.3.** Воспользуйтесь алгоритмом 4.47 для вычисления набора LALR-множеств пунктов на основе ядер LR (0)-множеств пунктов для грамматики из упражнения 4.7.1.

! **Упражнение 4.7.4.** Покажите, что грамматика

$$\begin{array}{lcl} S & \to & A\,a \mid b\,A\,c \mid d\,c \mid b\,d\,a \\ A & \to & d \end{array}$$

принадлежит классу LALR (1), но не SLR (1).

! Упражнение 4.7.5. Покажите, что грамматика

$$\begin{array}{lcl} S & \rightarrow & A\,a \mid b\,A\,c \mid B\,c \mid b\,B\,a \\ A & \rightarrow & d \\ B & \rightarrow & d \end{array}$$

принадлежит классу LR(1), но не LALR(1).

4.8 Использование неоднозначных грамматик

Тот факт, что каждая неоднозначная грамматика не может быть LR-грамматикой и, следовательно, не может относиться ни к одному из рассмотренных ранее классов грамматик, является доказанной теоремой. Ряд типов неоднозначных грамматик, однако, вполне пригоден для определения и реализации языков, как мы увидим в этом разделе. Для языковых конструкций наподобие арифметических выражений неоднозначные грамматики обеспечивают более краткую и естественную спецификацию по сравнению с эквивалентными однозначными грамматиками. Другое использование неоднозначных грамматик состоит в выделении распространенных синтаксических конструкций для специализированной оптимизации. Имея неоднозначную грамматику, можно определить специализированные конструкции путем аккуратного добавления в грамматику новых продукций.

Следует особо отметить, что, хотя используемые грамматики являются неоднозначными, во всех случаях задаются специальные правила разрешения неоднозначности, обеспечивающие для каждого предложения только одно дерево разбора. В этом смысле полное описание языка остается однозначным, так что иногда оказывается возможной разработка LR-анализатора, который следует тем же правилам разрешения неоднозначности. Подчеркнем также, что неоднозначные конструкции должны использоваться как можно реже и предельно аккуратно; в противном случае нет никакой гарантии, что синтаксический анализатор распознает соответствующий язык.

4.8.1 Использование приоритетов и ассоциативности для разрешения конфликтов

Рассмотрим неоднозначную грамматику (4.3) для выражений с операторами + и * (здесь она повторена для удобства читателя):

$$E \rightarrow E + E \mid E * E \mid (E) \mid \mathbf{id}$$

Данная грамматика неоднозначна, поскольку не определяет ассоциативность или приоритет операторов + и *. Однозначная грамматика (4.1), включающая продукции $E \rightarrow E + T$ и $T \rightarrow T * F$, генерирует тот же язык, но придает оператору +

низший приоритет по сравнению с оператором * и делает оба оператора левоассоциативными. Имеется две причины, по которым использование неоднозначной грамматики может оказаться предпочтительнее. Во-первых, как мы увидим ниже, можно легко изменить ассоциативность и уровень приоритета операторов + и * без изменения продукций (4.3) или количества состояний получающегося синтаксического анализатора. Во-вторых, синтаксический анализатор для однозначной грамматики будет тратить значительную часть времени на свертку по продукциям $E \to T$ и $T \to F$, единственная функция которых — определять приоритеты и ассоциативность операторов.

Множества LR (0)-пунктов для неоднозначной грамматики выражений (4.3), расширенной продукцией $E' \to E$, показаны на рис. 4.48. Поскольку грамматика (4.3) неоднозначна, при попытке построить таблицу LR-анализа по этому множеству возникают конфликты действий. В частности, такие конфликты порождают состояния, соответствующие множествам пунктов I_7 и I_8. Предположим, что мы используем SLR-подход для построения таблицы действий синтаксического анализа. Конфликт порождается множеством I_7 — между сверткой по продукции $E \to E + E$ и переносом + или * — и не может быть разрешен, поскольку и +, и * принадлежат множеству FOLLOW (E). Таким образом, для входных символов + и * могут выполняться оба действия. Подобный конфликт порождается множеством I_8 — между сверткой $E \to E * E$ и переносом при входных символах + и *. Фактически эти конфликты порождаются при любом способе построения таблиц LR-анализа, а не только при SLR-подходе.

Однако эти проблемы могут быть решены с использованием информации о приоритетах и ассоциативности операторов + и *. Рассмотрим входную строку **id** + **id** * **id**, которая заставляет синтаксический анализатор, основанный на рис. 4.48, попасть в состояние 7 после обработки **id** + **id**; в частности, синтаксический анализатор достигает конфигурации

Префикс	Стек	Вход
$E + E$	0 1 4 7	* **id** $

Для удобства символы, соответствующие состояниям 1, 4 и 7, показаны в столбце "Префикс".

Если приоритет оператора * выше приоритета +, мы знаем, что синтаксический анализатор должен выполнить перенос * в стек, подготавливая свертку * и соседних с этим оператором **id** в выражение. Эти же действия были совершены и SLR-анализатором на рис. 4.37, основанном на однозначной грамматике для того же языка. С другой стороны, если приоритет + выше приоритета *, синтаксический анализатор должен свернуть $E + E$ к E. Таким образом, относительное старшинство +, за которым следует *, однозначно определяет, каким образом

$$I_0: \quad E' \to \cdot E$$
$$E \to \cdot E + E$$
$$E \to \cdot E * E$$
$$E \to \cdot (E)$$
$$E \to \cdot \mathbf{id}$$

$$I_1: \quad E' \to E \cdot$$
$$E \to E \cdot + E$$
$$E \to E \cdot * E$$

$$I_2: \quad E \to (\cdot E)$$
$$E \to \cdot E + E$$
$$E \to \cdot E * E$$
$$E \to \cdot (E)$$
$$E \to \cdot \mathbf{id}$$

$$I_3: \quad E \to \mathbf{id} \cdot$$

$$I_4: \quad E \to E + \cdot E$$
$$E \to \cdot E + E$$
$$E \to \cdot E * E$$
$$E \to \cdot (E)$$
$$E \to \cdot \mathbf{id}$$

$$I_5: \quad E \to E * \cdot E$$
$$E \to \cdot E + E$$
$$E \to \cdot E * E$$
$$E \to \cdot (E)$$
$$E \to \cdot \mathbf{id}$$

$$I_6: \quad E \to (E \cdot)$$
$$E \to E \cdot + E$$
$$E \to E \cdot * E$$

$$I_7: \quad E \to E + E \cdot$$
$$E \to E \cdot + E$$
$$E \to E \cdot * E$$

$$I_8: \quad E \to E * E \cdot$$
$$E \to E \cdot + E$$
$$E \to E \cdot * E$$

$$I_9: \quad E \to (E) \cdot$$

Рис. 4.48. Множества LR (0)-пунктов расширенной грамматики выражений

должен быть разрешен конфликт между сверткой $E \to E + E$ и переносом * в состоянии 7.

Если входная строка имеет вид **id** + **id** + **id**, то синтаксический анализатор после обработки части строки **id** + **id** достигнет конфигурации, при которой содержимое стека — 0 1 4 7. При очередном входном символе + в состоянии 7 вновь возникает конфликт переноса/свертки. Однако теперь способ разрешения возникшего конфликта определяется ассоциативностью оператора +. Если этот оператор левоассоциативен, правильным действием синтаксического анализатора будет свертка по продукции $E \to E + E$. Первыми должны быть сгруппированы

id, окружающие первый оператор +. Этот выбор вновь совпадает с действиями SLR-анализатора для однозначной грамматики[11].

Итак, если оператор + левоассоциативен, действие в состоянии 7 для входного символа + должно приводить к свертке по продукции $E \to E + E$, а предположение о превосходстве приоритета оператора * над + ведет к переносу входного символа *. Аналогично, если оператор * левоассоциативен и старше оператора +, то можно доказать, что состояние 8, появляющееся на вершине стека только в том случае, когда три верхних символа представляют собой $E * E$, должно приводить к свертке по продукции $E \to E * E$ как при очередном входном символе +, так и при *. В случае входного символа + причина этого заключается в том, что оператор * старше оператора +, а в случае входного символа * — что оператор * левоассоциативен.

Продолжая рассмотрение, можно получить таблицу LR-анализа, показанную на рис. 4.49. Продукции 1–4 представляют собой соответственно $E \to E + E$, $E \to E * E$, $E \to (E)$ и $E \to$ **id**. Интересно, что подобная таблица может быть получена путем устранения сверток согласно продукциям $E \to T$ и $T \to F$ из SLR-таблицы для однозначной грамматики (4.1). В контексте LALR- и канонического LR-анализа с неоднозначными грамматиками такого типа можно поступать аналогично.

Состояние	ACTION						GOTO
	id	+	*	(	)	$	E
0	s3			s2			1
1		s4	s5			acc	
2	s3			s2			6
3		r4	r4		r4	r4	
4	s3			s2			7
5	s3			s2			8
6		s4	s5		s9		
7		r1	s5		r1	r1	
8		r2	r2		r2	r2	
9		r3	r3		r3	r3	

Рис. 4.49. Таблица синтаксического анализа для грамматики (4.3)

[11] В принципе, левоассоциативность (правоассоциативность) можно рассматривать как превосходство приоритета левого (правого) оператора одного и того же вида в случае их последовательного применения (естественно, отношения приоритетов операторов этого типа с другими операторами остаются в силе). — *Прим. ред.*

4.8.2 Неоднозначность "висящего else"

Вновь обратимся к грамматике для условных инструкций:

$$
\begin{array}{rcl}
\mathit{stmt} & \rightarrow & \textbf{if } \mathit{expr} \textbf{ then } \mathit{stmt} \textbf{ else } \mathit{stmt} \\
& | & \textbf{if } \mathit{expr} \textbf{ then } \mathit{stmt} \\
& | & \textbf{other}
\end{array}
$$

Как отмечалось в разделе 4.3.2, эта грамматика неоднозначна, поскольку не разрешает неоднозначности "висящего else". Для упрощения рассмотрим абстрактное представление приведенной выше грамматики, где i означает **if** *expr* **then**, e — **else**, a — "все остальные продукции". Тогда рассматриваемая грамматика с расширяющей продукцией $S' \rightarrow S$ может быть записана в виде

$$
\begin{array}{rcl}
S' & \rightarrow & S \\
S & \rightarrow & i\,S\,e\,S \mid i\,S \mid a
\end{array}
\tag{4.17}
$$

Множества LR (0)-пунктов для грамматики (4.17) показаны на рис. 4.50. Неоднозначность грамматики (4.17) приводит к конфликту переноса/свертки в I_4. Здесь пункт $S \rightarrow iS \cdot eS$ приводит к переносу e, а поскольку $\text{FOLLOW}(S) = \{e, \$\}$, пункт $S \rightarrow iS\cdot$ приводит к свертке по продукции $S \rightarrow iS$ при входном символе e.

$$
\begin{array}{llcll}
I_0: & S' \rightarrow \cdot S & \quad & I_3: & S \rightarrow a\cdot \\
& S \rightarrow \cdot iSeS & & & \\
& S \rightarrow \cdot iS & & I_4: & S \rightarrow iS \cdot eS \\
& S \rightarrow \cdot a & & & \\
& & & I_5: & S \rightarrow iSe \cdot S \\
I_1: & S' \rightarrow S\cdot & & & S \rightarrow \cdot iSeS \\
& & & & S \rightarrow \cdot iS \\
I_2: & S \rightarrow i \cdot SeS & & & S \rightarrow \cdot a \\
& S \rightarrow i \cdot S & & & \\
& S \rightarrow \cdot iSeS & & I_6: & S \rightarrow iSeS\cdot \\
& S \rightarrow \cdot iS & & & \\
& S \rightarrow \cdot a & & &
\end{array}
$$

Рис. 4.50. LR (0)-состояния для расширенной грамматики (4.17)

Если перевести это обратно в термины **if-then-else**, то, при наличии в стеке

if *expr* **then** *stmt*

и **else** в качестве первого входного символа, мы должны перенести **else** в стек (т.е. выполнить перенос e) или свернуть **if** *expr* **then** *stmt* в *stmt* (т.е. выполнить свертку по продукции $S \rightarrow iS$)? Ответ заключается в том, что мы должны перенести **else**, так как оно "связано" с предыдущим **then**. В терминах грамматики (4.17) входной символ e (означающий **else**) может быть только частью тела продукции, начинающегося с части iS, которая находится на вершине стека.

Итак, мы приходим к заключению, что конфликт переноса/свертки в I_4 должен быть разрешен в пользу переноса входного символа e. Таблица SLR-анализа, построенная по множествам пунктов на рис. 4.50 с использованием описанного разрешения конфликта, показана на рис. 4.51. Здесь продукции 1–3 представляют собой соответственно $S \rightarrow iSeS$, $S \rightarrow iS$ и $S \rightarrow a$.

Состояние	ACTION				GOTO
	i	e	a	$	S
0	s2		s3		1
1				acc	
2	s2		s3		4
3		r3		r3	
4		s5		r2	
5	s2		s3		6
6		r1		r1	

Рис. 4.51. Таблица LR-анализа для грамматики с "висящим else"

Например, при входной строке $iiaea$ синтаксический анализатор выполняет действия, показанные на рис. 4.52, в соответствии с корректным разрешением проблемы "висящего else". В строке (5) в состоянии 4 происходит перенос входного символа e, а в строке (9) в состоянии (4) при входном $ выполняется свертка по продукции $S \rightarrow iS$.

В случае, когда мы не можем использовать неоднозначную грамматику для условных инструкций, можно воспользоваться более громоздкой грамматикой из примера 4.16.

4.8.3 Восстановление после ошибок в LR-анализе

LR-анализатор обнаруживает ошибку при обращении к таблице ACTION синтаксического анализа и нахождении в ней записи об ошибке (при обращении к таблице GOTO ошибки не выявляются). LR-анализатор сообщит об ошибке, как только не сможет обнаружить корректного продолжения уже отсканированной части входного потока. Канонический LR-анализ в этом случае даже не выполнит ни

	Стек	Символы	Вход	Действия
(1)	0		$i\ i\ a\ e\ a\$$	Перенос
(2)	0 2	i	$i\ a\ e\ a\$$	Перенос
(3)	0 2 2	$i\ i$	$a\ e\ a\$$	Перенос
(4)	0 2 2 3	$i\ i\ a$	$e\ a\$$	Перенос
(5)	0 2 2 4	$i\ i\ S$	$e\ a\$$	Свертка по $S \to a$
(6)	0 2 2 4 5	$i\ i\ S\ e$	$a\$$	Перенос
(7)	0 2 2 4 5 3	$i\ i\ S\ e\ a$	$\$$	Свертка по $S \to a$
(8)	0 2 2 4 5 6	$i\ i\ S\ e\ S$	$\$$	Свертка по $S \to iSeS$
(9)	0 2 4	$i\ S$	$\$$	Свертка по $S \to iS$
(10)	0 1	S	$\$$	Принятие

Рис. 4.52. Действия синтаксического анализа для входной строки *iiaea*

одной свертки перед объявлением об ошибке; SLR- и LALR-анализаторы могут произвести ряд сверток перед тем, как сообщить об ошибке, но никогда не будут переносить в стек неверный символ.

При LR-анализе восстановление после ошибок "в режиме паники" реализуется следующим образом. Мы сканируем стек от вершины до состояния s с записью в таблице GOTO для некоторого нетерминала A. После этого пропускаем нуль или несколько символов входного потока, пока не будет найден символ a, который при отсутствии ошибки может законным образом следовать за A. После этого синтаксический анализатор переносит в стек состояние GOTO(s, A) и продолжает обычный синтаксический анализ. При выборе нетерминала A возможны несколько вариантов. Обычно это нетерминалы, представляющие крупные фрагменты программы, такие как выражение, инструкция и блок. Например, если A — нетерминал *stmt*, a может быть символом *;* или }, помечающим конец последовательности инструкции.

При таком методе восстановления производится попытка выделить фразу, содержащую синтаксическую ошибку. Синтаксический анализатор определяет, что строка, порождаемая из A, содержит ошибку. Часть этой строки уже обработана, и результат этой обработки — последовательность состояний, находящаяся на вершине стека. Остаток строки все еще находится во входном потоке, и синтаксический анализатор пытается пропустить этот остаток, находя символ, который может следовать за A в корректной программе. Удаляя состояния из стека, пропуская часть входного потока и помещая в стек GOTO(s, A), синтаксический анализатор полагает, что им найден экземпляр A, и продолжает обычный процесс анализа.

Восстановление на уровне фразы реализуется путем проверки каждой ошибочной записи в таблице LR-анализа и принятия решения (на основе знания особенностей языка) о том, какая наиболее вероятная ошибка программиста могла привести к данной ситуации. После этого можно построить подходящую процедуру восстановления после ошибки; возможно, при этом придется изменить вершину стека и/или первые символы входного потока способом, соответствующим данной записи ошибки.

При разработке специализированных подпрограмм обработки ошибок для LR-синтаксического анализатора можно заполнить каждую пустую запись таблицы действия указателем на подпрограмму, которая будет выполнять действия, выбранные для данного конкретного случая разработчиком компилятора. Эти действия могут включать вставку символов в стек или входной поток (или и туда, и туда) и удаление их оттуда или изменение и перестановку входных символов. Выбор должен делаться таким образом, чтобы исключить возможность зацикливания LR-анализатора. Безопасная стратегия должна гарантированно удалять или переносить при каждом цикле по крайней мере один символ из входного потока или при достижении его конца гарантированно уменьшать стек на каждой итерации. Снятия со стека состояния над нетерминалом следует избегать, поскольку такое изменение удаляет из стека успешно разобранную конструкцию.

Пример 4.51. Обратимся еще раз к грамматике выражений

$$E \to E + E \mid E * E \mid (E) \mid \mathbf{id}$$

На рис. 4.53 показана таблица LR-анализа для этой грамматики, представляющая собой дополненную программами обработки ошибок таблицу из рис. 4.49. Для каждого состояния, вызывающего свертку при том или ином входном символе, все записи ошибок заменены записями некоторых сверток. Такая замена приводит к отложенному обнаружению ошибок после выполнения одной или нескольких лишних сверток; ошибка в любом случае будет найдена до выполнения первого переноса. Оставшиеся пустыми ячейки заполнены указателями на подпрограммы обработки ошибок.

Далее приведены описания подпрограмм обработки ошибок.

e1: Эта подпрограмма вызывается из состояний 0, 2, 4 и 5; все они ожидают начало операнда — **id** или левую скобку. Вместо этого обнаруживается оператор + или * либо окончание входного потока.

Поместить в стек состояние 3 (переход из состояний 0, 2, 4 и 5 при входном символе **id**).
Вывести сообщение "Отсутствует операнд".

Состояние	ACTION						GOTO
	id	+	*	(	)	$	E
0	s3	e1	e1	s2	e2	e1	1
1	e3	s4	s5	e3	e2	acc	
2	s3	e1	e1	s2	e2	e1	6
3	r4	r4	r4	r4	r4	r4	
4	s3	e1	e1	s2	e2	e1	7
5	s3	e1	e1	s2	e2	e1	8
6	e3	s4	s5	e3	s9	e4	
7	r1	r1	s5	r1	r1	r1	
8	r2	r2	r2	r2	r2	r2	
9	r3	r3	r3	r3	r3	r3	

Рис. 4.53. Таблица LR-анализа с программами обработки ошибок

e2: Эта подпрограмма вызывается из состояний 0, 1, 2, 4 и 5 при обнаружении правой скобки.

Удалить правую скобку из входного потока.
Вывести сообщение "Несбалансированная правая скобка".

e3: Эта подпрограмма вызывается из состояний 1 и 6, когда ожидается оператор, а обнаруживается **id** или правая скобка.

Поместить в стек состояние 4 (соответствующее символу +).
Вывести сообщение "Отсутствует оператор".

e4: Эта подпрограмма вызывается из состояния 6 при обнаружении конца входного потока

Поместить в стек состояние 9 (для правой скобки).
Вывести сообщение "Отсутствует правая скобка".

Для ошибочной входной строки **id**+) последовательность конфигураций синтаксического анализатора показана на рис. 4.54. □

4.8.4 Упражнения к разделу 4.8

! **Упражнение 4.8.1.** Ниже приведена неоднозначная грамматика для выражений с n бинарными инфиксными операторами и n различными уровнями приоритетов:

$$E \to E\ \theta_1\ E \mid E\ \theta_2\ E \mid \cdots \mid E\ \theta_n\ E \mid (E) \mid \mathbf{id}$$

Стек	Символы	Входной поток	Действия
0		**id**+)$	
0 3	**id**	+)$	
0 1	E	+)$	
0 1 4	$E+$	)$	"Несбалансированная правая скобка"
			e2 удаляет правую скобку
0 1 4	$E+$	$	"Отсутствует операнд"
			e1 помещает в стек состояние 3
0 1 4 3	$E+$ **id**	$	
0 1 4 7	$E+$	$	
0 1	$E+$	$	

Рис. 4.54. Анализ и восстановление после ошибок в LR-анализаторе

а) Что собой представляют SLR-множества пунктов для этой грамматики как функция от n?

б) Как бы вы разрешили конфликты SLR-пунктов, если все операторы левоассоциативны, а приоритет оператора θ_1 выше приоритета оператора θ_2, который, в свою очередь, выше приоритета θ_3, и т.д.?

в) Приведите таблицу SLR-анализа, получающуюся в результате решения части *б* данного упражнения.

г) Повторите части *а* и *в* для однозначной грамматики, которая определяет то же множество выражений и показана на рис. 4.55.

д) Подсчитайте и сравните количества множеств пунктов и размеры таблиц для однозначной и неоднозначной грамматик. Что говорит это сравнение об использовании неоднозначных грамматик выражений?

$$
\begin{array}{lcl}
E_1 & \to & E_1 \; \theta_n \; E_2 \mid E_2 \\
E_2 & \to & E_2 \; \theta_{n-1} \; E_3 \mid E_3 \\
 & \cdots & \\
E_n & \to & E_n \; \theta_1 \; E_{n+1} \mid E_{n+1} \\
E_{n+1} & \to & (E_1) \mid \mathbf{id}
\end{array}
$$

Рис. 4.55. Однозначная грамматика для n операторов

! **Упражнение 4.8.2.** На рис. 4.56 представлена грамматика для ряда инструкций, похожая на грамматику из упражнения 4.4.12. Здесь, как и ранее, e и s — терминалы, обозначающие соответственно условные выражения и "иные инструкции".

а) Постройте таблицу LR-анализа для данной грамматики с разрешением конфликтов обычным для "висящего else" способом.

б) Реализуйте исправление ошибок путем заполнения пустых записей таблицы синтаксического анализа дополнительными свертками или подходящими подпрограммами восстановления после ошибок.

в) Покажите, как будет вести себя ваш синтаксический анализатор для следующих входных строк:

i) **if** e **then** s ; **if** e **then** s **end**

ii) **while** e **do begin** s ; **if** e **then** s ; **end**

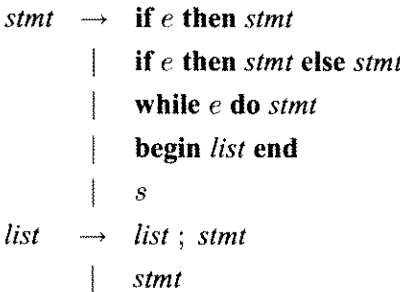


Рис. 4.56. Грамматика для инструкций некоторых видов

4.9 Генераторы синтаксических анализаторов

В этом разделе будет рассмотрен генератор синтаксических анализаторов, используемый для облегчения построения начальной фазы компилятора. Мы обратимся к генератору LALR-анализаторов `Yacc`, поскольку он реализует многие концепции из числа рассмотренных в двух предыдущих разделах и широко распространен. Название `Yacc` означает "Yet another compiler-compiler" (еще один компилятор компиляторов), что отражает популярность генераторов синтаксических анализаторов в начале 1970-х годов, когда С. Джонсоном (S.C. Johnson) была создана первая версия `Yacc`. Этот генератор доступен в качестве команды в UNIX и использовался при разработке многих промышленных компиляторов.

4.9.1 Генератор синтаксических анализаторов Yacc

Создание транслятора с использованием Yacc схематично показано на рис. 4.57. Вначале создается файл, скажем, translate.y, содержащий Yacc-спецификацию разрабатываемого транслятора. Команда UNIX

```
yacc translate.y
```

преобразует файл translate.y в программу y.tab.c на языке C с использованием LALR-метода, описанного в алгоритме 4.63. Программа y.tab.c является синтаксическим LALR-анализатором, написанным на языке C и объединенным с другими подпрограммами на языке C, которые могут быть подготовлены пользователем. Таблица LALR-анализа уплотнена с помощью технологии, описанной в разделе 4.7. Путем компиляции y.tab.c вместе с библиотекой ly, содержащей программу LR-анализа, с использованием команды

```
cc y.tab.c -ly
```

мы получим требуемую объектную программу a.out, которая выполняет трансляцию, определенную исходной программой Yacc[12]. Если необходимы другие процедуры, они могут быть скомпилированы или загружены вместе с y.tab.c точно так же, как и с любой другой программой на языке C.

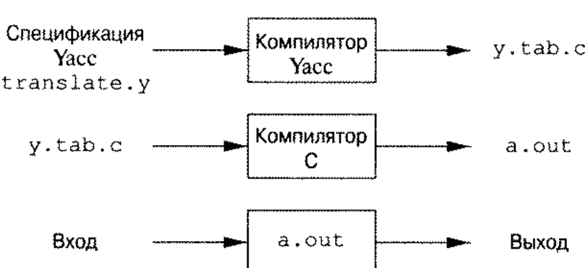


Рис. 4.57. Создание транслятора с помощью Yacc

Исходная Yacc-программа состоит из трех частей:

```
Объявления
%%
Правила трансляции
%%
C-подпрограммы поддержки
```

[12]Имя библиотеки (указанное параметром -ly) системно зависимо (т.е. может быть разным в разных системах).

Пример 4.52. Чтобы проиллюстрировать подготовку Yacc-программы, построим простой калькулятор, который считывает арифметические выражения, вычисляет их и выводит соответствующие числовые значения. Построение калькулятора начнем со следующей грамматики для арифметических выражений:

$$\begin{array}{lcl} E & \to & E + T \mid T \\ T & \to & T * F \mid F \\ F & \to & (E) \mid \textbf{digit} \end{array}$$

Токен **digit** представляет отдельную цифру от 0 до 9. Yacc-калькулятор на основе этой грамматики показан на рис. 4.58. □

Часть объявлений

В Yacc-программе имеется два необязательных раздела объявлений. В первом размещаются обычные объявления С, ограниченные `%{` и `%}`. Здесь мы помещаем объявления временных переменных, используемых правилами трансляции или процедурами второй и третьей частей. На рис. 4.58 этот раздел содержит только директиву

```
#include <ctype.h>
```

Она заставляет препроцессор С включить стандартный заголовочный файл `<ctype.h>`, содержащий описание предиката `isdigit`.

В части объявлений находятся также объявления токенов грамматики. На рис. 4.58 инструкция

```
%token DIGIT
```

объявляет токен `DIGIT`. Токены, объявленные в этом разделе, могут использоваться во второй и третьей частях спецификации Yacc. Если для создания лексического анализатора, передающего токены Yacc, использовался Lex, эти объявления токенов делаются доступными и для этого лексического анализатора, как говорилось в разделе 3.5.2.

Часть правил трансляции

В этой части спецификации Yacc после первой пары `%%` мы размещаем правила трансляции. Каждое правило состоит из продукции грамматики и связанных семантических действий. Множество продукций, которые мы записывали как

$$\langle\text{head}\rangle \to \langle\text{body}\rangle_1 \mid \langle\text{body}\rangle_2 \mid \cdots \mid \langle\text{body}\rangle_n ,$$

```
%{
#include <ctype.h>
%}

%token DIGIT

%%
line   : expr '\n'          { printf("%d\n", $1); }
       ;
expr   : expr '+' term      { $$ = $1 + $3; }
       | term
       ;
term   : term '*' factor    { $$ = $1 * $3; }
       | factor
       ;
factor : '(' expr ')'       { $$ = $2; }
       | DIGIT
       ;

%%
yylex() {
    int c;
    c = getchar();
    if (isdigit(c)) {
        yylval = c-'0';
        return DIGIT;
    }
    return c;
}
```

Рис. 4.58. Yacc-спецификация простейшего калькулятора

в Yacc записываются следующим образом:

$$
\begin{array}{lcll}
\langle\text{head}\rangle & \to & \langle\text{body}\rangle_1 & \{\ \langle\text{Семантическое действие}\rangle_1\ \} \\
 & | & \langle\text{body}\rangle_2 & \{\ \langle\text{Семантическое действие}\rangle_2\ \} \\
 & & \dots & \\
 & | & \langle\text{body}\rangle_n & \{\ \langle\text{Семантическое действие}\rangle_n\ \} \\
 & ; & &
\end{array}
$$

В продукциях Yacc строки букв и цифр без кавычек, не объявленные как токены, считаются нетерминалами. Отдельный символ в одинарных кавычках, например 'c', представляет терминальный символ c, а равно целочисленный

код токена, представленного этим символом (т.е. Lex должен вернуть синтаксическому анализатору код символа 'c' как целое число). Альтернативные тела продукций разделяются вертикальной чертой, а за каждой продукцией с ее альтернативами и семантическими действиями ставится точка с запятой. Первый заголовок считается стартовым символом.

Семантические действия Yacc представляют собой последовательности инструкций C. В них могут использоваться специальные символы: $$ представляет значение атрибута, связанного с нетерминалом в заголовке продукции, а $i — значение, связанное с i-м грамматическим символом (терминалом или нетерминалом) тела продукции. Семантическое действие выполняется при свертке связанной с ним продукции, так что обычно семантическое действие вычисляет значение $$ на основании значений $i. В спецификации Yacc две E-продукции

$$E \to E + T \mid T$$

и связанные с ними семантические действия выглядят как

```
expr : expr '+' term    { $$ = $1 + $3; }
     | term
     ;
```

Заметим, что нетерминал term в первой продукции представляет собой третий грамматический символ (вторым является оператор '+'). Семантическое действие, связанное с первой продукцией, суммирует значения expr и term тела продукции и присваивает полученную сумму атрибуту нетерминала expr в заголовке. Мы опускаем семантическое действие для второй продукции, поскольку для продукции телом из одного символа действие по умолчанию состоит в копировании значения атрибута. В общем случае семантическим действием по умолчанию является { $$ = $1; }.

Заметьте, что в спецификации Yacc добавлена новая стартовая продукция

```
line : expr '\n'    { printf("%d\n", $1); }
```

Эта продукция говорит о том, что входной поток калькулятора представляет собой выражение, за которым следует символ новой строки. Семантическое действие, связанное с этой продукцией, состоит в выводе десятичного значения выражения, за которым следует символ новой строки.

Часть C-подпрограмм поддержки

Третья часть спецификации Yacc содержит C-подпрограммы поддержки. Среди них обязательно должна находиться функция лексического анализатора yylex(). Обычно для получения yylex() используется Lex; см. раздел 4.9.3.

Другие функции, такие как подпрограммы восстановления после ошибок, могут быть добавлены при необходимости.

Лексический анализатор `yylex()` производит токены, каждый из которых состоит из имени токена и связанного значения атрибута. Если функция возвращает имя токена, такое как `DIGIT`, это имя токена должно быть объявлено в первой части спецификации `Yacc`. Значение атрибута, связанного с токеном, передается синтаксическому анализатору через предопределенную в `Yacc` переменную `yylval`.

Лексический анализатор на рис. 4.58 очень "сырой". Он считывает входной поток по одному символу с использованием функции `getchar()`. Если считанный символ — цифра, ее значение сохраняется в переменной `yylval`, а лексический анализатор возвращает имя токена `DIGIT`. В противном случае в качестве имени токена возвращается сам считанный символ.

4.9.2 Использование Yacc с неоднозначной грамматикой

Модифицируем спецификацию `Yacc` так, чтобы полученный в результате калькулятор стал немного полезнее. Во-первых, обеспечим возможность работы с несколькими выражениями, по одному в строке (разрешив также пустые строки между выражениями). Для этого изменим первое правило следующим образом:

```
lines : lines expr '\n'    { printf("%g\n", $2); }
      | lines '\n'
      | /* empty */
      ;
```

В `Yacc` пустая альтернатива (третья в данном примере) означает ϵ.

Во-вторых, расширим класс обрабатываемых выражений, включив числа вместо одиночных цифр, а также все арифметические операторы — `+`, `-` (как бинарные, так и унарные), `*` и `/`. Простейший путь определения такого класса выражений — неоднозначная грамматика

$$E \rightarrow E + E \mid E - E \mid E * E \mid E/E \mid -E \mid (E) \mid \textbf{number}$$

Полученная в результате спецификация `Yacc` приведена на рис. 4.59.

Поскольку данная грамматика неоднозначна, LALR-алгоритм будет генерировать конфликты действий синтаксического анализа. При этом `Yacc` сообщит о количестве имеющихся конфликтов. Описание множеств пунктов и конфликтов можно получить при запуске `Yacc` с опцией командной строки `-v`. При этом `Yacc` выводит дополнительный файл `y.output`, содержащий ядра множеств пунктов, найденные при синтаксическом анализе, описание конфликтов и удобочитаемое представление таблицы LR-анализа, показывающее, каким образом разрешены

```
%{
#include <ctype.h>
#include <stdio.h>
#define YYSTYPE double  /* Тип double для стека Yacc */
%}
%token NUMBER

%left '+' '-'
%left '*' '/'
%right UMINUS
%%

lines : lines expr '\n'    { printf("%g\n", $2); }
      | lines '\n'
      | /* empty */
      ;
expr  : expr '+' expr      { $$ = $1 + $3; }
      | expr '-' expr      { $$ = $1 - $3; }
      | expr '*' expr      { $$ = $1 * $3; }
      | expr '/' expr      { $$ = $1 / $3; }
      | '(' expr ')'       { $$ = $2; }
      | '-' expr  %prec UMINUS { $$ = - $2; }
      | NUMBER
      ;
%%
yylex() {
    int c;
    while ( ( c = getchar() ) == ' ' );
    if ( (c == '.') || (isdigit(c)) ) {
        ungetc(c, stdin);
        scanf("%lf", &yylval);
        return NUMBER;
    }
    return c;
}
```

Рис. 4.59. Спецификация Yacc улучшенного калькулятора

конфликты. Если `Yacc` выявляет конфликты, файл `y.output` с информацией о конфликтах создается без подсказки со стороны пользователя.

Если `Yacc` не получает иных указаний в своей спецификации, все конфликты разрешаются им исходя из следующих двух правил.

1. Конфликт "свертка/свертка" разрешается выбором продукции, находящейся первой в спецификациях `Yacc`.

2. Конфликт "перенос/свертка" разрешается в пользу переноса. Это правило корректно разрешает конфликт, возникающий из-за неоднозначности "висящего else".

Поскольку эти правила по умолчанию не всегда представляют именно то, что требуется создателю компилятора, `Yacc` обеспечивает общий механизм для разрешения конфликтов "перенос/свертка". В части объявлений терминалам можно назначить приоритеты и ассоциативность. Объявление

```
%left '+' '-'
```

задает операторы `+` и `-` как левоассоциативные, имеющие один и тот же уровень приоритета. Объявить оператор как правоассоциативный можно следующим образом:

```
%right '^'
```

Кроме того, можно обеспечить неассоциативность оператора (т.е. два оператора не могут быть скомбинированы никоим образом) следующим объявлением:

```
%nonassoc '<'
```

Токены получают уровни приоритетов в том порядке, в котором они встречаются в части объявлений, начиная с наинизшего. Токены в одном объявлении имеют одинаковые приоритеты. Таким образом, объявление

```
%right UMINUS
```

на рис. 4.2 дает токену `UMINUS` наивысший приоритет по сравнению с пятью предшествующими терминалами.

`Yacc` разрешает конфликты "перенос/свертка" назначением приоритета и ассоциативности каждой продукции, участвующей в конфликте (так же, как и каждому терминалу, вовлеченному в конфликтную ситуацию). Если необходимо осуществить выбор между переносом входного символа a и сверткой в соответствии с продукцией $A \to \alpha$, то `Yacc` выполняет свертку, если приоритет продукции выше приоритета a или если приоритеты одинаковы и ассоциативность продукции — левая. В противном случае выбирается перенос.

Обычно приоритет продукции устанавливается таким же, как у ее крайнего справа терминала. В большинстве случаев это разумное решение. Например, в продукции

$$E \to E + E \mid E * E$$

мы предпочитаем свертку по продукции $E \to E + E$, если очередной входной символ — $+$, поскольку $+$ в теле продукции имеет тот же приоритет, что и входной символ, и при этом он левоассоциативен. В случае очередного входного символа $*$ мы предпочитаем перенос, поскольку входной символ имеет более высокий приоритет, чем $+$ в теле продукции.

В тех ситуациях, когда крайний справа терминал не обеспечивает корректный приоритет продукции, его можно изменить путем добавления к продукции дескриптора

`%prec` ⟨терминал⟩

При этом приоритет и ассоциативность продукции будут такими же, как у использованного в дескрипторе терминала, который, как правило, определен в разделе объявлений. `Yacc` не сообщает о конфликтах "перенос/свертка", разрешенных с использованием механизма приоритета и ассоциативности.

Используемый "терминал" может быть искусственно введенным, как, например, `UMINUS` в рассматриваемом калькуляторе. Такой терминал не возвращается лексическим анализатором и объявляется исключительно для того, чтобы определить приоритет продукции. Объявление на рис. 4.2

```
%right UMINUS
```

назначает токену `UMINUS` наивысший приоритет, больший, чем у операторов `*` и `/`. При описании правил трансляции дескриптор

```
%prec UMINUS
```

в конце продукции

```
expr  : '-' expr
```

делает унарный минус, определяемый этой продукцией, оператором, имеющим наивысший по сравнению с остальными операторами приоритет.

4.9.3 Создание лексического анализатора в Yacc с помощью Lex

`Lex` предназначен для создания лексических анализаторов, которые могут использоваться в `Yacc`. Библиотека `Lex ll` предоставляет подпрограмму-драйвер с именем `yylex()`, которое `Yacc` использует при вызове лексического анализатора. При использовании `Lex` в качестве генератора лексического анализатора функция `yylex()` в третьей части спецификации `Yacc` заменяется инструкцией

```
#include "lex.yy.c"
```

и каждый вызов лексического анализатора возвращает известный Yacc терминал. Указанная инструкция обеспечивает функции yylex() доступ к именам, используемым Yacc для токенов, поскольку выходной файл Lex компилируется как составная часть выходного файла Yacc y.tab.c.

В UNIX, если спецификации Lex находятся в файле first.l, а спецификации Yacc — в файле second.y, транслятор может быть скомпилирован последовательностью команд

```
lex first.l
yacc second.y
cc y.tab.c -ly -ll
```

Приведенная на рис. 4.60 спецификация Lex может использоваться для получения лексического анализатора вместо используемого на рис. 4.2. Последний шаблон спецификации, означающий "любой символ", должен быть записан как \n|., поскольку в Lex точка означает любой символ, кроме символа новой строки.

```
number   [0-9]+\e.?|[0-9]*\e.[0-9]+
%%
[ ]      { /* skip blanks */ }
{number} { sscanf(yytext, "%lf", &yylval);
               return NUMBER; }
\n|.    { return yytext[0]; }
```

Рис. 4.60. Спецификация Lex для yylex() из рис. 4.2

4.9.4 Восстановление после ошибок в Yacc

В Yacc восстановление после ошибок может быть выполнено с использованием специальных продукций для ошибок. Вначале следует решить, какие "крупные" нетерминалы будут иметь связанные с ними продукции ошибок. Типичным выбором является некоторое подмножество нетерминалов, порождающих выражения, инструкции, блоки и процедуры. Затем пользователь добавляет к грамматике продукции ошибок в виде $A \to$ **error** α, где A — "крупный" нетерминал, а α — строка символов грамматики, возможно, пустая; **error** является в Yacc зарезервированным словом. Yacc генерирует синтаксический анализатор с использованием таких спецификаций, рассматривая продукцию ошибки как обычную.

Однако когда синтаксический анализатор, сгенерированный Yacc, сталкивается с ошибкой, он рассматривает состояния, множества пунктов которых содержат

продукции ошибок, особым образом. При обнаружении ошибки Yacc снимает символы со стека до тех пор, пока на вершине стека не окажется состояние, множество пунктов которого включает пункт вида $A \to \cdot$**error** α. После этого синтаксический анализатор выполняет "перенос" фиктивного токена **error** в стек, как если бы такой токен присутствовал во входном потоке.

Если α представляет собой ϵ, немедленно производится свертка в A и выполняется семантическое действие, связанное с продукцией $A \to$ **error** (которое может представлять собой пользовательскую программу восстановления после ошибки). После этого синтаксический анализатор отбрасывает входные символы, пока не будет найден символ, позволяющий продолжить нормальный анализ.

Если α — не пустая строка, то Yacc пропускает символы входного потока в поисках подстроки, которая может быть свернута в α. Если α полностью состоит из терминалов, Yacc ищет соответствующую строку во входном потоке и "сворачивает" ее, перенося в стек. В этот момент на вершине стека синтаксического анализатора находится **error** α. После этого синтаксический анализатор сворачивает **error** α в A и продолжает нормальный анализ.

Например, продукция ошибки

$$smtp \to \textbf{error} \;;$$

говорит синтаксическому анализатору о том, что он должен пропустить весь входной поток до точки с запятой и считать, что был найден нетерминал *stmt*. Семантической подпрограмме для такой ошибки не нужно работать с входным потоком, но она может, например, вывести диагностическое сообщение и выставить флаг, запрещающий генерацию объектного кода.

Пример 4.53. На рис. 4.61 показана спецификация калькулятора с продукцией ошибки

```
lines : error '\n'
```

Эта продукция заставляет калькулятор приостановить нормальную работу при обнаружении ошибки во входной строке. Обнаружив ошибку, синтаксический анализатор начинает снимать символы со стека, пока не встретит состояние, имеющее перенос по токену **error**. Состояние 0 является таковым (и в нашем примере единственным), поскольку его пункты включают

$$lines \to \cdot\textbf{error}\ \texttt{'\textbackslash n'}$$

Состояние 0 всегда находится на дне стека. Синтаксический анализатор переносит **error** в стек и пропускает все символы входного потока, пока не обнаружит символ новой строки. Синтаксический анализатор переносит его в стек, сворачивает **error** `'\n'` в *lines* и выводит диагностическое сообщение "Повторите ввод

```
%{
#include <ctype.h>
#include <stdio.h>
#define YYSTYPE double  /* Тип double для стека Yacc */
%}
%token NUMBER

%left '+' '-'
%left '*' '/'
%right UMINUS
%%

lines : lines expr '\n'   { printf("%g\n", $2); }
      | lines '\n'
      | /* empty */
      | error '\n' { yyerror("Повторите ввод"
                       " последней строки:");
                     yyerrok; }
      ;
expr  : expr '+' expr     { $$ = $1 + $3; }
      | expr '-' expr     { $$ = $1 - $3; }
      | expr '*' expr     { $$ = $1 * $3; }
      | expr '/' expr     { $$ = $1 / $3; }
      | '(' expr ')'      { $$ = $2; }
      | '-' expr  %prec UMINUS { $$ = - $2; }
      | NUMBER
      ;
%%
#include "lex.yy.c"
```

Рис. 4.61. Калькулятор с восстановлением после ошибок

последней строки:". Специальная подпрограмма Yacc yyerrok[13] переводит синтаксический анализатор в режим обычной работы. □

[13]В листинге yyerrok приводится без скобок, указывающих, что это функция, поскольку реально yyerrok — макроопределение. — *Прим. ред.*

4.9.5 Упражнения к разделу 4.9

! **Упражнение 4.9.1.** Напишите Yacc-программу, которая получает в качестве входа булево выражение (см. грамматику из упражнения 4.2.2, *ж*) и выводит его значение.

! **Упражнение 4.9.2.** Напишите Yacc-программу, которая получает списки (см. грамматику из упражнения 4.2.2, *д*, но с произвольным символом в качестве элемента, а не только *a*) и выводит линейное представление списка, т.е. единый список элементов в том же порядке, в котором они поступают на вход.

! **Упражнение 4.9.3.** Напишите Yacc-программу, которая проверяет, является ли ее вход *палиндромом*, т.е. последовательностью символов, одинаково читающихся как в прямом, так и в обратном направлении.

!! **Упражнение 4.9.4.** Напишите Yacc-программу, которая получает регулярное выражение (см. грамматику из упражнения 4.2.2, *г*, но с любым произвольным символом в качестве аргумента, а не только *a*) и выводит таблицу переходов для недетерминированного конечного автомата, распознающего этот же язык.

4.10 Резюме к главе 4

- *Синтаксические анализаторы*. Синтаксический анализатор получает в качестве входных данных токены от лексического анализатора и рассматривает имена токенов как терминальные символы контекстно-свободной грамматики. Затем синтаксический анализатор строит дерево разбора для входной последовательности токенов; дерево разбора может быть построено фигурально (проходом по соответствующим шагам порождения) или буквально.

- *Контекстно-свободные грамматики*. Грамматика определяет множество терминальных символов (входных символов), множество нетерминалов (символов, представляющих синтаксические конструкции) и множество продукций, каждая из которых указывает способ, которым строки, представленные одним нетерминалом, могут быть построены из терминальных символов и строк, представленных некоторыми другими нетерминалами. Продукция состоит из заголовка (заменяемого нетерминала) и тела (замещающей строки из символов грамматики).

- *Порождения*. Процесс, начинающийся со стартового нетерминала грамматики и последовательно замещающий каждый нетерминал телом одной из его продукций, называется порождением. Если всегда замещается крайний слева (справа) нетерминал, то такое порождение называется левым (правым).

- *Деревья разбора.* Дерево разбора представляет собой рисунок порождения, на котором для каждого нетерминала в порождении имеется свой узел. Дочерние узлы представляют собой символы, которыми заменяется в процессе разбора нетерминал родительского узла. Существует взаимно-однозначное соответствие между деревьями разбора и левыми и правыми порождениями одной и той же строки терминалов.

- *Неоднозначность.* Грамматика, у которой некоторая строка терминалов имеет два или более различных деревьев разбора (или, что то же самое, два или более левых (или правых) порождения), называется неоднозначной. В большинстве случаев, представляющих практический интерес, неоднозначную грамматику можно преобразовать так, чтобы она стала однозначной для того же языка. Однако неоднозначные грамматики с применением некоторых специальных приемов приводят к более эффективным синтаксическим анализаторам.

- *Нисходящий и восходящий синтаксический анализ.* Синтаксические анализаторы в общем случае различаются по тому, работают ли они в нисходящем направлении (начиная со стартового символа грамматики и строя дерево разбора сверху вниз) или в восходящем (начиная с терминальных символов, образующих листья дерева разбора, и строя дерево снизу вверх). Нисходящие синтаксические анализаторы включают синтаксические анализаторы, работающие методом рекурсивного спуска, и LL-анализаторы; наиболее распространенными среди восходящих синтаксических анализаторов являются LR-анализаторы.

- *Проектирование грамматик.* Грамматики для нисходящего синтаксического анализа спроектировать зачастую сложнее, чем грамматики для восходящего разбора. В них необходимо устранить левую рекурсию — ситуацию, когда нетерминал порождает строку, которая начинается с того же нетерминала. Следует также выполнить левую факторизацию — сгруппировать продукции для одного и того же нетерминала с общим префиксом тела продукции.

- *Синтаксические анализаторы, работающие методом рекурсивного спуска.* Эти синтаксические анализаторы используют для каждого нетерминала свою процедуру. Эта процедура просматривает входной поток и принимает решение о том, какая продукция должна быть применена для ее нетерминала. Терминалы в теле продукции в нужный момент проверяются на соответствие входным данным, а обработка нетерминалов в теле продукции заключается в вызове соответствующих процедур. При этом методе возможен возврат в случае выбора неверной продукции.

- ✦ LL (1)-анализаторы. Грамматика, такая, что корректная продукция для данного нетерминала может быть выбрана путем просмотра только одного очередного входного символа, называется LL (1)-грамматикой. Такие грамматики позволяют строить таблицы предиктивного синтаксического анализа, которые для каждого нетерминала и входного символа дают корректный выбор продукции. Обработка ошибок может быть облегчена путем размещения подпрограмм для обработки ошибок в некоторых (или во всех) записях таблицы, в которых нет корректных продукций.

- ✦ *Синтаксический анализ методом переноса/свертки.* Восходящие синтаксические анализаторы в общем случае работают путем выбора на основе очередного входного символа (символа предпросмотра) и содержимого стека выполняемого действия — переноса очередного входного символа в стек или свертки нескольких символов на вершине стека. Свертка заменяет тело продукции на вершине стека заголовком этой продукции.

- ✦ *Активные префиксы.* В синтаксическом анализе методом переноса/свертки содержимое стека всегда представляет собой активный префикс, т.е. префикс некоторой правосентенциальной формы, который завершается не правее конца основы этой правосентенциальной формы. Основа представляет собой подстроку, образующуюся на последнем шаге правого порождения сентенциальной формы.

- ✦ *Допустимые пункты.* Пункт представляет собой продукцию с точкой в некотором месте в теле продукции. Пункт является допустимым для активного префикса, если продукция этого пункта используется для генерации основы, а активный префикс включает все символы слева от точки, но не справа.

- ✦ *LR-анализаторы.* Каждый из нескольких видов LR-синтаксических анализаторов начинает работу с построения множеств допустимых пунктов (именуемых LR-состояниями) для всех возможных активных префиксов и отслеживает состояние для каждого префикса в стеке. Множество допустимых пунктов определяет принятие решения о выполняемом действии. Если существует допустимый пункт с точкой на правом конце тела продукции, выбирается свертка; если очередной символ находится непосредственно справа от точки в некотором допустимом пункте, то выполняется перенос этого символа в стек.

- ✦ *SLR-синтаксические анализаторы.* В SLR-анализаторе мы выполняем свертку, вызванную допустимым пунктом с точкой на правом конце, при условии, что очередной входной символ может следовать за заголовком применяемой для свертки продукции в некоторой сентенциальной форме.

Грамматика принадлежит классу SLR, а описанный метод применим, если отсутствуют конфликты действий, т.е. если не существует такого множества пунктов и входного символа, для которых имеются две продукции для свертки или имеется возможность как свертки, так и переноса.

- *Канонические LR-анализаторы.* Этот более сложный вид LR-анализаторов использует пункты, дополненные символами входного потока, которые могут следовать после свертки по соответствующей продукции. Свертка осуществляется только в том случае, если существует допустимый пункт с точкой на правом конце и текущий входной символ — один из разрешенных для данного пункта. Канонический LR-анализатор может избежать некоторых конфликтов действий SLR-анализатора, но зачастую имеет существенно большее количество состояний, чем SLR-анализатор для той же грамматики.

- *LALR-синтаксические анализаторы.* LALR-анализаторы обладают многими преимуществами SLR и канонических LR-анализаторов, объединяя состояния, которые имеют одни и те же ядра (множества пунктов без связанных с ними множеств входных символов). Таким образом, количество состояний оказывается тем же, что и у SLR-анализатора, но в LALR-анализаторе может быть устранен ряд конфликтов SLR-анализатора. На практике чаще всего используется именно этот метод синтаксического анализа.

- *Восходящий синтаксический анализ неоднозначных грамматик.* Во многих важных ситуациях, таких как синтаксический анализ арифметических выражений, можно использовать неоднозначную грамматику с применением сторонней информации, такой как приоритеты операторов, для разрешения конфликтов между переносом и сверткой или между переносами по двум разным продукциям. Таким образом, LR-метод синтаксического анализа распространяется на многие неоднозначные грамматики.

- `Yacc`. Генератор синтаксических анализаторов `Yacc` принимает (возможно) неоднозначную грамматику и информацию для разрешения конфликтов и строит LALR-состояния. Затем он создает функцию, которая использует эти состояния для выполнения восходящего синтаксического анализа и вызывает при выполнении сверток связанные с ними функции.

4.11 Список литературы к главе 4

Формализм контекстно-свободных грамматик был введен Хомски (Chomsky) [5] в процессе изучения естественных языков. Эта идея была использована в описании синтаксиса двух ранних языков: Fortran — Бэкусом (Backus) [2] и Algol 60 —

Науром (Naur) [26]. Ученый Панини (Panini) разработал эквивалентную синтаксическую запись для описания правил грамматики санскрита между 400 и 200 годами до н.э. [19].

Явление неоднозначности впервые было рассмотрено Кантором (Cantor) [4] и Флойдом (Floyd) [13]. Нормальная форма Хомски (см. упражнение 4.4.8) была разработана в [6]. Обзор теории контекстно-свободных грамматик можно найти в [17].

Синтаксический анализ методом рекурсивного спуска использовался в ранних компиляторах, таких как [16], и системах разработки компиляторов, таких как META [28] и TMG [25]. LL-грамматики были предложены Льюисом (Lewis) и Стирнсом (Stearns) [24]. Упражнение 4.4.5 взято из [3].

Идея приоритета операторов и использования функций приоритетов принадлежит Флойду (Floyd) [14]. Эта идея была обобщена для части языка, не содержащей операторы, Виртом (Wirth) и Вебером (Weber) [29]. Сегодня эти методы сами по себе используются редко, но они являются ведущими среди усовершенствований LR-анализа.

LR-грамматики и синтаксические анализаторы были введены Кнутом (Knuth) [22], который описал построение канонических таблиц LR-анализа. LR-метод не рассматривался как практический, поскольку таблицы синтаксического анализа превышали имевшуюся в то время оперативную память типичного компьютера, до тех пор, пока Кореньяк (Korenjak) [23] не разработал метод получения таблиц вполне разумного размера для синтаксического анализа типичного языка программирования. Де Ремер (DeRemer) разработал методы LALR [8] и SLR [9], которые используются и сегодня. Построение таблиц LR-анализа для неоднозначных грамматик рассматривалось в [1 и 12].

`Yacc` Джонсона (Johnson) очень быстро продемонстрировал практичность генерации LALR-синтаксических анализаторов для разработки компиляторов. Руководство по использованию `Yacc` можно найти в [20]. Версия с открытым кодом — `Bison` — описана в [10]. Похожий генератор на основе LALR-анализа, называющийся `CUP` [18], поддерживает действия, написанные на Java. Существующие в настоящее время генераторы нисходящих анализаторов включают `Antlr` [27] — поддерживающий действия на C++, Java и C# генератор анализаторов, работающих методом рекурсивного спуска, и `LLGen` [15] — генератор LL (1)-анализаторов.

Библиография по обработке синтаксических ошибок собрана Дейном (Dain) [7].

Алгоритм синтаксического анализа общего назначения с использованием динамического программирования был разработан независимо Коком (J. Cocke) (не опубликовано), Янгером (Younger) [30] и Касами (Kasami) [21], откуда и получил свое название. Имеется более сложный алгоритм общего назначения Эрли (Earley) [11], который табулирует LR-пункты для каждой подстроки данной входной

строки; время работы этого алгоритма в общем случае — $O(n^3)$, но для однозначных грамматик оно сокращается до $O(n^2)$.

1. Aho, A. V., S. C. Johnson, and J. D. Ullman, "Deterministic parsing of ambiguous grammars", *Comm. ACM* **18**:8 (Aug., 1975), pp. 441–452.

2. Backus, J. W., "The syntax and semantics of the proposed international algebraic language of the Zurich-ACM-GAMM Conference", *Proc. Intl. Conf. Information Processing*, UNESCO, Paris (1959), pp. 125–132.

3. Birman, A. and J. D. Ullman, "Parsing algorithms with backtrack", *Information and Control* **23**:1 (1973), pp. 1–34.

4. Cantor, D. C., "On the ambiguity problem of Backus systems", *J. ACM* **9**:4 (1962), pp. 477–479.

5. Chomsky, N., "Three models for the description of language", *IRE Trans. on Information Theory* **IT-2**:3 (1956), pp. 113–124.

6. Chomsky, N., "On certain formal properties of grammars", *Information and Control* **2**:2 (1959), pp. 137–167.

7. Dain, J., "Bibliography on Syntax Error Handling in Language Translation Systems", 1991. Доступен в группе новостей `comp.compilers`; см. `http://compilers.iecc.com/comparch/article/91-04-050`.

8. DeRemer, F., "Practical Translators for LR(k) Languages", Ph.D. thesis, MIT, Cambridge, MA, 1969.

9. DeRemer, F., "Simple LR(k) grammars", *Comm. ACM* **14**:7 (July, 1971), pp. 453–460.

10. Donnelly, C. and R. Stallman, "Bison: The YACC-compatible Parser Generator", `http://www.gnu.org/software/bison/manual/`.

11. Earley, J., "An efficient context-free parsing algorithm", *Comm. ACM* **13**:2 (Feb., 1970), pp. 94–102.

12. Earley, J., "Ambiguity and precedence in syntax description", *Acta Informatica* **4**:2 (1975), pp. 183–192.

13. Floyd, R. W., "On ambiguity in phrase-structure languages", *Comm. ACM* **5**:10 (Oct., 1962), pp. 526–534.

14. Floyd, R. W., "Syntactic analysis and operator precedence", *J. ACM* **10**:3 (1963), pp. 316–333.

15. Grune, D and C. J. H. Jacobs, "A programmer-friendly LL(1) parser generator", *Software Practice and Experience* **18**:1 (Jan., 1988), pp. 29–38. См. также `http://www.cs.vu.nl/~ceriel/LLgen.html`.

16. Hoare, C. A. R., "Report on the Elliott Algol translator", *Computer J.* **5**:2 (1962), pp. 127–129.

17. Hopcroft, J. E., R. Motwani, and J. D. Ullman, *Introduction to Automata Theory, Languages, and Computation*, Addison-Wesley, Boston MA, 2001.

18. Hudson, S. E. et al., "CUP LALR Parser Generator in Java", доступно по адресу `http://www2.cs.tum.edu/projects/cup/`.

19. Ingerman, P. Z., "Panini-Backus form suggested", *Comm. ACM* 10:3 (March 1967), p. 137.

20. Johnson, S. C., "Yacc — Yet Another Compiler Compiler", Computing Science Technical Report 32, Bell Laboratories, Murray Hill, NJ, 1975. Доступно по адресу `http://dinosaur.compilertools.net/yacc/`.

21. Kasami, T., "An efficient recognition and syntax analysis algorithm for context-free languages", AFCRL-65-758, Air Force Cambridge Research Laboratory, Bedford, MA, 1965.

22. Knuth, D. E., "On the translation of languages from left to right", *Information and Control* **8**:6 (1965), pp. 607–639.

23. Korenjak, A. J., "A practical method for constructing LR(k) processors", *Comm. ACM* **12**:11 (Nov., 1969), pp. 613–623.

24. Lewis, P. M. II and R. E. Stearns, "Syntax-directed transduction", *J. ACM* **15**:3 (1968), pp. 465–488.

25. McClure, R. M., "TMG — a syntax-directed compiler", *Proc. 20th ACM Natl. Conf.* (1965), pp. 262–274.

26. Naur, P. et al., "Report on the algorithmic language ALGOL 60", *Comm. ACM* **3**:5 (May, 1960), pp. 299–314. См. также *Comm. ACM* **6**:1 (Jan., 1963), pp. 1–17.

27. Parr, T., "ANTLR", `http://www.antlr.org/`.

28. Schorre, D. V., "Meta-II: a syntax-oriented compiler writing language", *Proc. 19th ACM Natl. Conf.* (1964), pp. D1.3-1–D1.3-11.

29. Wirth, N. and H. Weber, "Euler: a generalization of Algol and its formal definition: Part I", *Comm. ACM* **9**:1 (Jan., 1966), pp. 13–23.

30. Younger, D.H., "Recognition and parsing of context-free languages in time n^3", *Information and Control* **10**:2 (1967), pp. 189–208.

Глава 5

Синтаксически управляемая трансляция

В данной главе развивается затронутая в разделе 2.3 тема — трансляция языков, управляемая контекстно-свободной грамматикой. Методы трансляции из этой главы будут использоваться в главе 6 для проверки типов и генерации промежуточного кода. Эти методы применимы также при реализации небольших языков для специализированных задач; в данной главе приведен пример из полиграфии.

Мы связываем информацию с конструкциями языка, назначая *атрибуты* символу (или символам) грамматики, представляющему конструкцию (этот вопрос уже рассматривался в разделе 2.3.2). Синтаксически управляемое определение указывает значения атрибутов при помощи связывания с грамматическими продукциями семантических правил. Например, транслятор инфиксных выражений в постфиксные может иметь следующие продукцию и правило:

$$\begin{array}{ll} \text{Продукция} & \text{Семантическое правило} \\ E \rightarrow E_1 + T & E.code = E_1.code \| T.code \| '+' \end{array} \quad (5.1)$$

Эта продукция содержит два нетерминала, E и T; индекс у E_1 предназначен для того, чтобы отличать E в теле продукции от E в заголовке. И E, и T имеют атрибут *code*, представляющий собой строку. Семантическое правило указывает, что строка $E.code$ образуется путем конкатенации строк $E_1.code$, $T.code$ и символа '+'. Правило явно указывает, что трансляция E образуется из трансляций E, T и '+', но реализация в ходе непосредственной работы со строками может оказаться неэффективной.

В разделе 2.3.5 схема синтаксически управляемой трансляции вставляет в тела продукций программные фрагменты, называющиеся семантическими действиями, как в следующем случае:

$$E \rightarrow E_1 + T \; \{print\; '+'\} \quad (5.2)$$

По соглашению семантические действия заключаются в фигурные скобки. (Если фигурная скобка встречается в качестве грамматического символа, она заключается в одинарные кавычки — '{' и '}'.) Положение семантического действия

в теле продукции определяет порядок, в котором выполняются действия. В продукции (5.2) действие выполняется в конце, после всех грамматических символов; в общем случае семантические действия могут располагаться в любом месте тела продукции.

Сравнивая эти два варианта записи, можно заметить, что синтаксически управляемые определения более удобочитаемы, а следовательно, более подходят в качестве спецификаций. Однако схемы трансляций могут оказаться более эффективны и потому в большей степени пригодны для реализаций.

Наиболее общий подход к синтаксически управляемой трансляции состоит в построении дерева разбора или синтаксического дерева с последующим вычислением значений атрибутов в узлах путем обхода узлов этого дерева. Во многих случаях трансляция может выполняться в процессе синтаксического анализа, без явного построения дерева разбора. Мы познакомимся с классом синтаксически управляемых трансляций, которые называются "L-атрибутными трансляциями" (здесь L означает "слева направо") и включают почти все трансляции, которые могут выполняться в процессе синтаксического анализа. Мы также изучим меньший класс — "S-атрибутные трансляции" (здесь S означает "синтезируемые"), которые легко выполняются при восходящем синтаксическом анализе.

5.1 Синтаксически управляемые определения

Синтаксически управляемое определение (СУО) является контекстно-свободной грамматикой с атрибутами и правилами. Атрибуты связаны с грамматическими символами, а правила — с продукциями. Если X представляет собой символ, а a — один из его атрибутов, то значение a в некотором узле дерева разбора, помеченном X, записывается как $X.a$. Если узлы дерева разбора реализованы в виде записей или объектов, то атрибуты X могут быть реализованы как поля данных в записях, представляющих узлы X. Атрибуты могут быть любого вида: числами, типами, таблицами ссылок или строками. Строки могут представлять собой, в частности, длинные последовательности кода, например кода на промежуточном языке, используемом компилятором.

5.1.1 Наследуемые и синтезируемые атрибуты

Мы будем работать с двумя типами атрибутов для нетерминалов.

1. *Синтезируемый атрибут* для нетерминала A в узле N дерева разбора определяется семантическим правилом, связанным с продукцией в N. Заметим, что в качестве заголовка этой продукции должен выступать нетерминал A. Синтезируемый атрибут в узле N определяется только с использованием значений атрибутов в дочерних по отношению к N узлах и в самом узле N.

Альтернативное определение наследуемых атрибутов

Если позволить наследуемому атрибуту $B.c$ в узле N быть определенным с использованием значений атрибутов в дочерних узлах N, а также в самом N, его родителе и братьях, то не нужны никакие дополнительные трансляции. Такие правила могут быть "имитированы" путем создания дополнительных атрибутов B, скажем, $B.c_1$, $B.c_2$, ... Это синтезируемые атрибуты, которые копируют необходимые атрибуты узлов, дочерних по отношению к узлу B. Затем $B.c$ вычисляется как наследуемый атрибут с использованием атрибутов $B.c_1$, $B.c_2$, ... вместо атрибутов дочерних узлов. На практике такие атрибуты требуются редко.

2. *Наследуемый атрибут* для нетерминала B в узле N дерева разбора определяется семантическим правилом, связанным с продукцией в N. Заметим, что нетерминал B должен участвовать в продукции в качестве символа в ее теле. Наследуемый атрибут в узле N определяется с использованием значений атрибутов в родительском по отношению к N узле, в самом узле N и дочерних узлах его родительского узла ("братьях" N).

Наследуемые атрибуты в узле N не могут определяться через атрибуты дочерних по отношению к N узлов; синтезируемые же атрибуты в узле N могут определяться через значения наследуемых атрибутов в самом узле N.

Терминалы могут иметь синтезируемые, но не наследуемые атрибуты. Атрибуты терминалов имеют лексические значения, придаваемые им лексическим анализатором. В СУО не существует семантических правил для вычисления значений атрибутов терминалов.

Пример 5.1. СУО на рис. 5.1 основано на знакомой нам грамматике для арифметических выражений с операторами + и *. Оно вычисляет выражения, завершающиеся ограничивающим маркером **n**. Каждый нетерминал в СУО имеет единственный синтезируемый атрибут *val*. Мы также считаем, что терминал **digit** имеет синтезируемый атрибут *lexval*, представляющий собой целое значение, возвращаемое лексическим анализатором.

Правило для продукции 1, $L \to E$ **n**, устанавливает $L.val$ равным $E.val$, которое, как мы увидим, равно числовому значению всего выражения.

Продукция 2, $E \to E_1 + T$, также имеет одно правило, которое вычисляет атрибут *val* заголовка E как сумму значений E_1 и T. В любом помеченном E узле N дерева разбора значение *val* для E представляет собой сумму значений *val* дочерних по отношению к N узлов с метками E и T.

	Продукция	Семантическое правило
1)	$L \to E$ **n**	$L.val = E.val$
2)	$E \to E_1 + T$	$E.val = E_1.val + T.val$
3)	$E \to T$	$E.val = T.val$
4)	$T \to T_1 * F$	$T.val = T_1.val \times F.val$
5)	$T \to F$	$T.val = F.val$
6)	$F \to (E)$	$F.val = E.val$
7)	$F \to$ **digit**	$F.val =$ **digit**.*lexval*

Рис. 5.1. Синтаксически управляемое определение простого настольного калькулятора

Продукция 3, $E \to T$, имеет единственное правило, которое определяет значение *val* для E как значение того же атрибута у дочернего узла T. Продукция 4 подобна продукции 2: ее правило просто умножает значения атрибутов в дочерних узлах вместо их сложения. Правила продукций 5 и 6 копируют значения атрибутов дочерних узлов, так же, как и в продукции 3. Продукция 7 присваивает атрибуту $F.val$ числовое значение токена **digit**, возвращаемое лексическим анализатором.□

СУО, которые включают только синтезируемые атрибуты, называются *S-атрибутными* определениями; СУО на рис. 5.1 обладает данным свойством. В S-атрибутном СУО каждое правило вычисляет атрибут нетерминала в заголовке продукции, используя атрибуты, полученные из тела продукции.

Для простоты в примерах этого раздела используются семантические правила без побочных действий. На практике бывает удобно допускать в СУО небольшие ограниченные побочные действия, такие как печать результатов вычислений настольного калькулятора или работа с таблицей символов. При рассмотрении порядка вычислений атрибутов в разделе 5.2 мы позволим семантическим правилам вычислять произвольные функции, возможно, с побочными действиями.

S-атрибутное СУО может быть естественным образом реализовано вместе с LR-синтаксическим анализатором. Фактически СУО на рис. 5.1 отображает Yacc-программу, представленную на рис. 4.58, которая иллюстрирует трансляцию во время синтаксического анализа. Отличие заключается в том, что в правиле для продукции 1 Yacc-программа в качестве побочного действия выводит $E.val$ вместо определения значения атрибута $L.val$.

СУО без побочных эффектов иногда называют *атрибутной грамматикой*. Правила атрибутной грамматики определяют значения атрибутов через значения других атрибутов и констант и не выполняют никаких иных действий.

5.1.2 Вычисление СУО в узлах дерева разбора

Визуализировать трансляцию, определяемую СУО, может помочь работа с деревом разбора, хотя в действительности транслятор может его и не строить. Представим, что правила СУО применяются путем построения дерева разбора с последующим использованием этих правил для вычисления атрибутов в каждом узле дерева. Дерево разбора с указанием значений его атрибутов называется *аннотированным деревом разбора*.

Каким образом строится аннотированное дерево разбора? В каком порядке вычисляются его атрибуты? Перед тем как вычислять атрибут в узле дерева разбора, следует вычислить все атрибуты, от которых может зависеть вычисляемое значение. Например, если все атрибуты являются синтезируемыми, как в примере 5.1, то, прежде чем приступить к вычислению значения атрибута *val* в узле, требуется вычислить атрибуты *val* во всех его дочерних узлах.

Синтезируемые атрибуты можно вычислять в произвольном восходящем порядке, таком как обратный порядок обхода дерева разбора; вычисление S-атрибутных определений рассматривается в разделе 5.2.3.

Для СУО с атрибутами обоих типов — наследуемыми и синтезируемыми — не существует гарантии наличия даже одного порядка обхода для вычисления всех атрибутов в узлах дерева разбора. Рассмотрим, например, нетерминалы A и B с синтезируемым и наследуемым атрибутами $A.s$ и $B.i$ соответственно, а также следующую продукцию и правила:

Продукция	Семантические правила
$A \to B$	$A.s = B.i$
	$B.i = A.s + 1$

Данные правила циклические; невозможно вычислить ни $A.s$ в узле N, ни $B.i$ в дочернем по отношению к N узле, не зная значение другого атрибута. Циклическая зависимость $A.s$ и $B.i$ в некоторой паре узлов дерева разбора проиллюстрирована на рис. 5.2.

Задача поиска циклов в дереве разбора для данного СУО вычислительно достаточно сложна[1]. К счастью, существуют подклассы СУО, для которых можно гарантировать существование порядка вычисления (этот вопрос рассматривается в разделе 5.2).

Пример 5.2. На рис. 5.3 показано аннотированное дерево разбора для входной строки $3 * 5 + 4$ **n**, построенное с применением грамматики и правил, представленных на рис. 5.1. Предполагается, что значения *lexval* предоставляются лексическим анализатором. Каждый из узлов для нетерминалов имеет атрибут *val*,

[1]Не вдаваясь в детали, отметим, что хотя данная задача и разрешима, она не может быть решена алгоритмом с полиномиальным временем работы (даже если $\mathcal{P} = \mathcal{NP}$) в силу экспоненциальной сложности.

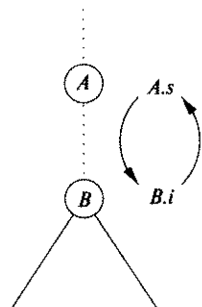


Рис. 5.2. Циклическая зависимость атрибутов $A.s$ и $B.i$ друг от друга

вычисляемый в восходящем порядке; на рисунке показаны значения атрибутов в каждом узле дерева разбора. Например, к узлу, среди дочерних узлов которого имеется узел, помеченный *, после вычисления значений атрибутов $T.val = 3$ и $F.val = 5$ в его первом и третьем дочернем узлах применяется правило, гласящее, что $T.val$ является произведением указанных значений, или 15. □

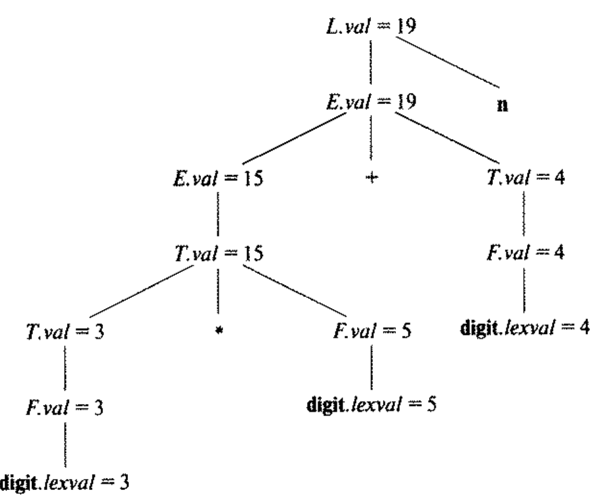


Рис. 5.3. Аннотированное дерево разбора для $3 * 5 + 4$ **n**

Наследуемые атрибуты полезны в случае, когда структура дерева разбора "не соответствует" абстрактному синтаксису исходного кода. Приведенный далее пример показывает, как наследуемые атрибуты могут использоваться для преодоления такого несоответствия, возникшего вследствие того, что грамматика разрабатывалась для синтаксического анализа, а не для трансляции.

Пример 5.3. СУО на рис. 5.4 вычисляет выражения наподобие $3 * 5$ и $3 * 5 * 7$. Нисходящий синтаксический анализ входной строки $3*5$ начинается с продукции $T \to F\ T'$. Здесь F генерирует цифру 3, но оператор * генерируется нетерминалом T'. Таким образом, левый операнд 3 находится в дереве разбора в другом поддереве, не в том, в котором находится оператор *. Следовательно, для передачи операнда оператору используются наследуемые атрибуты.

	Продукция	Семантические правила
1)	$T \to F\ T'$	$T'.inh = F.val$
		$T.val = T'.syn$
2)	$T' \to *F\ T'_1$	$T'_1.inh = T'.inh \times F.val$
		$T'.syn = T'_1.syn$
3)	$T' \to \epsilon$	$T'.syn = T'.inh$
4)	$F \to$ **digit**	$F.val =$ **digit**.*lexval*

Рис. 5.4. СУО на основе грамматики, пригодной для нисходящего синтаксического анализа

Грамматика в этом примере представляет собой фрагмент нелеворекурсивной версии знакомой нам грамматики выражений; мы использовали такую грамматику в примере, иллюстрирующем нисходящий синтаксический анализ (см. раздел 4.4).

Каждый из нетерминалов T и F имеет синтезируемый атрибут *val*; терминал **digit** имеет синтезируемый атрибут *lexval*. Нетерминал T' имеет два атрибута: наследуемый атрибут *inh* и синтезируемый атрибут *syn*.

Семантические правила основаны на той идее, что левый операнд оператора * является наследуемым. Точнее, заголовок T' продукции $T' \to *\ F\ T'_1$ наследует левый операнд оператора * в теле продукции. У выражения $x * y * z$ корень поддерева для $*y * z$ наследует x. Далее, корень поддерева $*z$ наследует значение $x * y$, и так далее при наличии большего количества сомножителей в выражении. Когда все сомножители накоплены, результат передается назад по дереву с использованием синтезируемых атрибутов.

Чтобы увидеть, как используются семантические правила, рассмотрим аннотированное дерево разбора для выражения $3 * 5$ на рис. 5.5. Крайний слева лист дерева, помеченный **digit**, имеет значение атрибута *lexval* $= 3$, где 3 передается лексическим анализатором. Родительским узлом для него является узел продукции $F \to$ **digit**. Единственное семантическое правило, связанное с этой продукцией, определяет, что $F.val =$ **digit**.*lexval*, равному 3.

Во втором дочернем узле корня наследуемый атрибут $T'.inh$ определяется семантическим правилом $T'.inh = F.val$, связанным с продукцией 1. Таким образом,

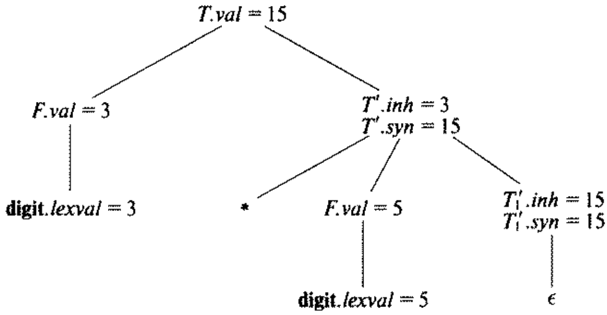


Рис. 5.5. Аннотированное дерево разбора для выражения $3 * 5$

левый операнд оператора * — 3 — передается слева направо между дочерними узлами корня.

Продукция в узле для T' – $T' \to * \ F \ T_1'$ (индекс 1 оставлен в аннотированном дереве разбора для того, чтобы различать два узла T'). Наследуемый атрибут $T_1'.inh$ определяется семантическим правилом $T_1'.inh = T'.inh \times F.val$, связанным с продукцией 2.

При $T'.inh = 3$ и $F.val = 5$ мы получаем $T_1'.inh = 15$. Узлу для T_1' соответствует продукция $T' \to \epsilon$. Семантическое правило $T'.syn = T'.inh$ определяет значение $T_1'.syn = 15$. Атрибут *syn* в узлах для нетерминала T' передает значение 15 вверх по дереву до узла для нетерминала T, так что $T.val = 15$. □

5.1.3 Упражнения к разделу 5.1

Упражнение 5.1.1. Для СУО на рис. 5.1 постройте аннотированные деревья разбора для следующих выражений:

а) $(3 + 4) * (5 + 6)$ **n**;

б) $1 * 2 * 3 * (4 + 5)$ **n**;

в) $(9 + 8 * (7 + 6) + 5) * 4$ **n**.

Упражнение 5.1.2. Расширьте СУО на рис. 5.4 так, чтобы оно могло обрабатывать те же выражения, что и СУО на рис. 5.1.

Упражнение 5.1.3. Повторите упражнение 5.1.1 с использованием СУО, построенного вами при решении упражнения 5.1.2.

5.2 Порядок вычисления в СУО

Полезным инструментом для установления порядка вычисления атрибутов в конкретном дереве разбора является граф зависимостей. В то время как аннотированное дерево разбора показывает значения атрибутов, граф зависимостей помогает определить, каким образом эти значения могут быть вычислены.

В этом разделе в дополнение к графам зависимостей мы определим два важных класса СУО: S-атрибутные и более общие L-атрибутные СУО. Трансляции, определяемые этими двумя классами, хорошо согласуются с изучавшимися нами методами синтаксического анализа, и большинство трансляторов, встречающихся на практике, могут быть написаны так, чтобы соответствовать требованиям как минимум одного из упомянутых классов.

5.2.1 Графы зависимостей

Граф зависимостей (dependency graph) изображает поток информации между экземплярами атрибутов в определенном дереве разбора; ребро от одного экземпляра атрибута к другому означает, что значение первого атрибута необходимо для вычисления второго. Ребра выражают ограничения, следующие из семантических правил. Вот более детальное описание графа зависимостей.

- Для каждого узла дерева разбора, скажем, узла, помеченного грамматическим символом X, в графе зависимостей имеются узлы для каждого из атрибутов, связанных с X.

- Предположим, что семантическое правило, связанное с продукцией p, определяет значение синтезируемого атрибута $A.b$ через значение $X.c$ (правило может использовать для определения $A.b$ и другие атрибуты, помимо $X.c$). В таком случае в графе зависимостей имеется ребро от $X.c$ к $A.b$. Точнее, в каждом узле N, помеченном A, в котором применяется продукция p, создается ребро к атрибуту b в N от атрибута c в дочернем по отношению к N узле, соответствующем экземпляру символа X в теле продукции[2].

- Предположим, что семантическое правило, связанное с продукцией p, определяет значение наследуемого атрибута $B.c$ с использованием значения $X.a$. Тогда граф зависимостей содержит ребро от $X.a$ к $B.c$. Для каждого узла N, помеченного B, который соответствует появлению этого B в теле продукции p, создается ребро к атрибуту c в N от атрибута a в узле M, который соответствует данному вхождению X. Заметим, что M может быть родителем либо братом N.

[2]Поскольку узел N может иметь несколько дочерних узлов, помеченных X, считаем, что различать один и тот же символ в разных местах продукции позволяют их индексы.

Пример 5.4. Рассмотрим следующую продукцию и правило:

Продукция	Семантическое правило
$E \to E_1 + T$	$E.val = E_1.val + T.val$

В каждом узле N, помеченном E, с дочерними узлами, соответствующими телу этой продукции, синтезируемый атрибут *val* в N вычисляется с использованием значений *val* в двух дочерних узлах, помеченных E и T. Таким образом, часть графа зависимостей для каждого дерева разбора, в котором используется эта продукция, выглядит так, как показано на рис. 5.6. По соглашению ребра дерева разбора будут изображаться пунктирной, а ребра графа зависимостей — сплошной линией. □

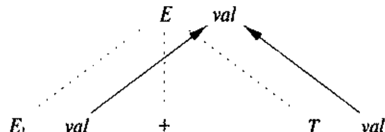


Рис. 5.6. $E.val$ синтезируется из $E_1.val$ и $T.val$

Пример 5.5. Пример полного графа зависимостей приведен на рис. 5.7. Узлы графа зависимостей, представленные числами от 1 до 9, соответствуют атрибутам в аннотированном дереве разбора на рис. 5.5.

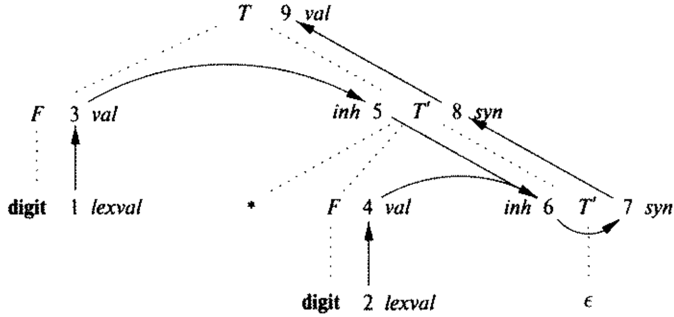


Рис. 5.7. Граф зависимостей для аннотированного дерева разбора, представленного на рис. 5.5

Узлы 1 и 2 представляют атрибут *lexval*, связанный с двумя листьями с метками **digit**. Узлы 3 и 4 представляют атрибут *val*, связанный с двумя узлами с метками F. Ребра в узел 3 из 1 и в узел 4 из 2 получаются из семантического правила, определяющего $F.val$ через **digit**.*lexval*. В действительности $F.val$ равно **digit**.*lexval*, но ребро представляет зависимость, а не равенство.

Узлы 5 и 6 представляют наследуемый атрибут $T'.inh$, связанный с каждым появлением нетерминала T'. Причиной наличия ребра из 3 в 5 служит правило $T'.inh = F.val$, определяющее $T'.inh$ в правом дочернем узле корня с использованием значения $F.val$ в левом дочернем узле. Имеются также ребра в узел 6 из узлов 5 (атрибут $T'.inh$) и 4 (атрибут $F.val$), поскольку указанные значения перемножаются для получения атрибута inh в узле 6.

Узлы 7 и 8 представляют синтезируемый атрибут syn, связанный с вхождениями T'. Ребро из узла 6 в узел 7 связано с семантическим правилом $T'.syn = T'.inh$, назначенным продукции 3 на рис. 5.4. Ребро из узла 7 в узел 8 связано с семантическим правилом, назначенным продукции 2.

Наконец, узел 9 представляет атрибут $T.val$. Ребро из узла 8 в узел 9 вызвано семантическим правилом $T.val = T'.syn$, связанным с продукцией 1. □

5.2.2 Упорядочение вычисления атрибутов

Граф зависимостей определяет возможные порядки вычисления атрибутов в различных узлах дерева разбора. Если граф зависимостей имеет ребро из узла M в узел N, то атрибут, соответствующий M, должен быть вычислен до атрибута N. Таким образом, допустимыми порядками вычисления атрибутов являются последовательности узлов $N_1, N_2, \ldots, N_k$, такие, что если в графе зависимостей имеется ребро от N_i к N_j, то $i < j$. Такое упорядочение графа зависимостей выстраивает его узлы в линейном порядке и называется *топологической сортировкой* графа.

Если в графе имеется цикл, топологическая сортировка такого графа невозможна, а значит, невозможно и вычисление СУО для данного дерева разбора. Если циклов нет, то существует как минимум одна топологическая сортировка. Чтобы понять, почему это так, убедимся, что в графе без циклов всегда имеется узел, в который не входит ни одно ребро. Если бы это было не так, то, переходя по входящим ребрам от предшественника к его предшественнику, мы бы в конечном счете попали в узел, в котором уже бывали, т.е. обнаружили бы цикл. Сделаем узел без входящих в него ребер первым в топологическом порядке, удалим его из графа зависимостей и повторим тот же процесс с оставшимися узлами.

Пример 5.6. Граф зависимостей на рис. 5.7 не имеет циклов. Одной из топологических сортировок является порядок, в котором пронумерованы узлы: $1, 2, \ldots, 9$. Заметим, что все ребра графа идут из узла с меньшим номером в узел с бóльшим номером, так что данный порядок определенно является топологической сортировкой. Имеются и другие топологические сортировки, например $1, 3, 5, 2, 4, 6, 7, 8, 9$. □

5.2.3 S-атрибутные определения

Как упоминалось ранее, для заданного СУО очень трудно сказать, существует ли дерево разбора, граф зависимостей которого содержит циклы. На практике трансляция может быть реализована с использованием классов СУО, для которых гарантированно существует порядок вычислений атрибутов, поскольку они не допускают наличия графов зависимостей с циклами. Два класса, рассматриваемых в этом разделе, могут быть эффективно реализованы в связи с нисходящим и восходящим синтаксическим анализом.

Первый из этих классов определяется следующим образом.

- СУО является *S-атрибутным*, если все его атрибуты синтезируемые.

Пример 5.7. СУО на рис. 5.1 является примером S-атрибутного определения. Каждый из атрибутов $L.val$, $E.val$, $T.val$ и $F.val$ является синтезируемым. □

Когда СУО является S-атрибутным, его атрибуты могут вычисляться в любом восходящем порядке узлов в дереве разбора. Зачастую особенно просто вычислить атрибуты путем обхода дерева разбора в обратном порядке и вычисления атрибутов в узле N, когда обход покидает узел последний раз, т.е. если применить определенную ниже функцию *postorder* к корню дерева разбора (см. также врезку "Обходы в прямом и обратном порядке" из раздела 2.3.4):

postorder (N) {
  for (каждый дочерний узел C узла N, начиная слева) *postorder* (C);
  Вычислить атрибуты, связанные с узлом N;
}

S-атрибутные определения могут быть реализованы во время нисходящего синтаксического анализа, поскольку нисходящий синтаксический анализ соответствует обходу в обратном порядке. В частности, обратный порядок обхода в точности соответствует порядку, в котором LR-синтаксический анализатор сворачивает тела продукций в их заголовки. Этот факт будет использован в разделе 5.4.2 для вычисления синтезируемых атрибутов и сохранения их в стеке в процессе LR-синтаксического анализа без явного создания узлов дерева разбора.

5.2.4 L-атрибутные определения

Второй класс СУО называется *L-атрибутными определениями*. Идея, лежащая в основе этого класса, заключается в том, что ребра графа зависимостей между атрибутами, связанными с телом продукции, идут только слева направо, но не справа налево (отсюда и название — "L-атрибутный"). Точнее, каждый атрибут должен быть либо

1. синтезируемым, либо

2. наследуемым, но при выполнении определенных ограничивающих правил. Предположим, что имеется продукция $A \to X_1 X_2 \ldots X_n$ и что существует наследуемый атрибут $X_i.a$, вычисляемый при помощи правила, связанного с данной продукцией. Тогда это правило может использовать только

   а) наследуемые атрибуты, связанные с заголовком A;

   б) наследуемые либо синтезируемые атрибуты, связанные с вхождениями символов $X_1, X_2, \ldots X_{i-1}$, расположенных слева от X_i;

   в) наследуемые либо синтезируемые атрибуты, связанные с вхождениями самого X_i, но только таким образом, что в графе зависимостей, образованном атрибутами этого X_i, не имеется циклов.

Пример 5.8. СУО на рис. 5.4 является L-атрибутным. Чтобы увидеть, почему это так, рассмотрим семантические правила для наследуемых атрибутов, которые для удобства повторим еще раз:

Продукция	Семантическое правило
$T \to F\ T'$	$T'.inh = F.val$
$T' \to *F\ T'_1$	$T'_1.inh = T'.inh \times F.val$

Первое из этих правил определяет наследуемый атрибут $T'.inh$, использующий только $F.val$, причем, как и требуется, F находится в теле продукции слева от T'. Второе правило определяет $T'_1.inh$ с использованием наследуемого атрибута $T'.inh$, связанного с заголовком, и $F.val$, где F располагается слева от T'_1 в теле продукции.

В каждом из этих случаев правила используют информацию "сверху или слева", как и требуется определением класса. Остальные атрибуты синтезируемые. Следовательно, рассмотренное СУО является L-атрибутным. □

Пример 5.9. Любое СУО, содержащее следующие продукции и правила, не может быть L-атрибутным:

Продукция	Семантические правила
$A \to B\ C$	$A.s = B.b$
	$B.i = f(C.c, A.s)$

Первое правило, $A.s = B.b$, является корректным как для S-атрибутного, так и для L-атрибутного СУО. Оно определяет синтезируемый атрибут $A.s$ с использованием атрибута в дочернем узле (т.е. символа в теле продукции).

Второе правило определяет наследуемый атрибут $B.i$, так что СУО, в целом, не может быть S-атрибутным. Далее, хотя правило и корректно, СУО не может

быть L-атрибутным, поскольку в определении $B.i$ участвует $C.c$, а C находится справа от B в теле продукции. Чтобы в L-атрибутном СУО могли использоваться атрибуты из братских узлов, они должны располагаться слева от символа, для которого определяется рассматриваемый атрибут. □

5.2.5 Семантические правила с контролируемыми побочными действиями

На практике трансляция включает побочные действия: настольный калькулятор может выводить результат вычисления, генератор кода может вносить тип идентификатора в таблицу символов. В случае СУО нас интересует баланс между атрибутными грамматиками и схемами трансляции. Атрибутные грамматики не имеют побочных действий и допускают любой порядок вычислений, согласующийся с графом зависимостей. Схемы трансляции требуют вычислений слева направо и допускают семантические действия, содержащие произвольные программные фрагменты; схемы трансляции рассматриваются в разделе 5.4.

Мы будем контролировать побочные действия в СУО одним из следующих способов.

- Позволяя второстепенные побочные действия, не препятствующие вычислению атрибутов. Другими словами, побочные действия разрешаются, если вычисление атрибутов на основе любой топологической сортировки графа зависимостей дает "корректную" трансляцию, где "корректность" зависит от конкретного приложения.

- Ограничивая допустимые порядки вычисления так, что при любом из разрешенных порядков вычисления получается одна и та же трансляция. Ограничения могут рассматриваться как неявные ребра, добавленные к графу зависимостей.

В качестве примера второстепенного побочного действия изменим калькулятор из примера 5.1 так, чтобы он выводил полученный результат. Вместо правила $L.val = E.val$, которое сохраняет результат вычислений в синтезируемом атрибуте $L.val$, рассмотрим следующее правило:

	Продукция	Семантическое правило
1)	$L \to E\ \mathbf{n}$	$print\,(E.val)$

Семантические правила, выполнение которых сводится только к побочным действиям, будут рассматриваться как определения фиктивных синтезируемых атрибутов, связанных с заголовком продукции. Данное модифицированное СУО дает ту же трансляцию при любой топологической сортировке, поскольку инструкция вывода выполняется в самом конце, после того как результат вычислений сохранен в $E.val$.

Пример 5.10. СУО на рис. 5.8 получает простое объявление D, состоящее из базового типа T, за которым следует список идентификаторов L. T может быть **int** или **float**. Для каждого идентификатора в списке тип вносится в запись таблицы символов для данного идентификатора. Мы считаем, что внесение типа в запись таблицы символов для данного идентификатора не влияет на записи таблицы символов для прочих идентификаторов. Таким образом, записи могут обновляться в произвольном порядке. Это СУО не проверяет, не объявлен ли идентификатор более одного раза; для этого в СУО можно внести необходимые изменения.

	Продукция	Семантические правила
1)	$D \to T\ L$	$L.inh = T.type$
2)	$T \to$ **int**	$T.type =$ integer
3)	$T \to$ **float**	$T.type =$ float
4)	$L \to L_1$, **id**	$L_1.inh = L.inh$
		addType (**id**.*entry*, $L.inh$)
5)	$L \to$ **id**	*addType* (**id**.*entry*, $L.inh$)

Рис. 5.8. Синтаксически управляемое определение для простых объявлений типов

Нетерминал D представляет объявление, которое в соответствии с продукцией 1 состоит из типа T, за которым следует список идентификаторов L. T имеет один атрибут, $T.type$, который является типом объявления D. Нетерминал L также имеет один атрибут, который назван *inh* для того, чтобы подчеркнуть, что этот атрибут — наследуемый. Назначение $L.inh$ — передача объявленного типа вниз по списку идентификаторов, чтобы он мог быть добавлен в соответствующие записи таблицы символов.

Каждая из продукций 2 и 3 вычисляет синтезируемый атрибут $T.type$, присваивая ему корректное значение — integer или float. Этот тип передается атрибуту $L.inh$ в правиле для продукции 1. Продукция 4 передает $L.inh$ вниз по дереву разбора, т.е. значение $L_1.inh$ вычисляется в узле дерева разбора путем копирования значения $L.inh$ из родительского по отношению к данному узлу; родительский узел соответствует заголовку продукции.

Продукции 4 и 5 имеют также правило, в соответствии с которым вызывается функция *addType* с двумя аргументами:

1. **id**.*entry*, лексическим значением, которое указывает на объект таблицы символов;

2. $L.inh$, типом, назначенным каждому идентификатору из списка.

Мы предполагаем, что функция *addType* корректно устанавливает тип $L.inh$ в качестве типа представленного идентификатора.

Граф зависимостей для входной строки **float id**$_1$, **id**$_2$, **id**$_3$ показан на рис. 5.9. Числа от 1 до 10 представляют узлы графа зависимостей. Узлы 1, 2 и 3 представляют атрибут *entry*, связанный с каждым из листьев с метками **id**. Узлы 6, 8 и 10 являются фиктивными атрибутами, представляющими применение функции *addType* к типу и одному из упомянутых значений *entry*.

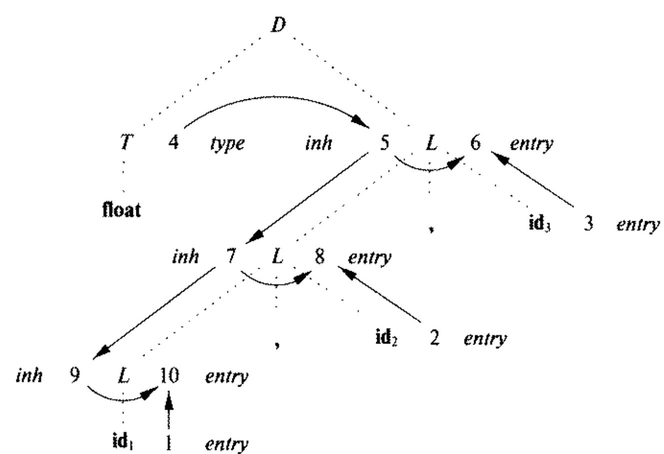


Рис. 5.9. Граф зависимостей для объявления **float id**$_1$, **id**$_2$, **id**$_3$

Узел 4 представляет атрибут *T.type*, и здесь действительно начинается вычисление атрибута. Этот тип затем передается узлам 5, 7 и 9, представляющим атрибут $L.inh$, связанный с каждым появлением нетерминала L. □

5.2.6 Упражнения к разделу 5.2

Упражнение 5.2.1. Приведите все возможные топологические сортировки для графа зависимостей на рис. 5.7.

Упражнение 5.2.2. Изобразите аннотированные деревья разбора для СУО на рис. 5.8 для следующих выражений:

а) `int a, b, c;`

б) `float w, x, y, z.`

Упражнение 5.2.3. Предположим, что у нас есть продукция $A \to B\ C\ D$. Каждый из нетерминалов A, B, C и D имеет два атрибута: синтезируемый атрибут s и наследуемый атрибут i. Для каждого из перечисленных ниже наборов правил

укажите, согласуются ли эти правила с S- и L-атрибутным определениями и существует ли вообще какой-либо порядок вычисления атрибутов в соответствии с указанными правилами.

а) $A.s = B.i + C.s$

б) $A.s = B.i + C.s$ и $D.i = A.i + B.s$

в) $A.s = B.s + D.s$

! г) $A.s = D.i$, $B.i = A.s + C.s$, $C.i = B.s$ и $D.i = B.i + C.i$

! **Упражнение 5.2.4.** Приведенная грамматика генерирует бинарные числа с "десятичной" точкой:

$$\begin{array}{lcl} S & \rightarrow & L\,.\,L \mid L \\ L & \rightarrow & L\,B \mid B \\ B & \rightarrow & 0 \mid 1 \end{array}$$

Разработайте L-атрибутное СУО для вычисления *S.val*, десятичного значения входной строки. Например, трансляция строки `101.101` должна давать десятичное число 5.625. *Указание*: воспользуйтесь наследуемым атрибутом *L.inh*, который говорит, с какой стороны от десятичной точки располагается бит.

!! **Упражнение 5.2.5.** Разработайте S-атрибутное СУО для грамматики и трансляции, описанных в упражнении 5.2.4.

!! **Упражнение 5.2.6.** Реализуйте алгоритм 3.23, который преобразует регулярные выражения в недетерминированные конечные автоматы, как L-атрибутное СУО для грамматики, к которой применим нисходящий синтаксический анализ. Считаем, что токен **char** представляет произвольный символ, а **char**.*lexval* — символ, который он представляет. Вы можете также считать, что существует функция *new* (), которая возвращает новое состояние, т.е. состояние, которое ранее этой функцией не возвращалось. Используйте любые подходящие обозначения для указания переходов НКА.

5.3 Применения синтаксически управляемой трансляции

Синтаксически управляемые методы из данной главы будут применены в главе 6 к проверке типов и генерации промежуточного кода. Здесь же мы рассмотрим избранные примеры, иллюстрирующие некоторые представительные СУО.

Основное применение в этом разделе заключается в построении синтаксических деревьев. Поскольку некоторые компиляторы в качестве промежуточного представления используют синтаксические деревья, распространенным видом

СУО является преобразование входной строки в дерево. Для завершения трансляции в промежуточный код компилятор затем может обойти построенное синтаксическое дерево, используя другой набор правил. (В главе 6 рассматривается подход к построению промежуточного кода, при котором СУО применяется без явного построения дерева.)

Мы рассмотрим два СУО для построения синтаксических деревьев для выражений. Первое S-атрибутное определение может использоваться в процессе восходящего синтаксического анализа. Второе L-атрибутное определение подходит для использования во время нисходящего синтаксического анализа.

Заключительный пример в этой главе представляет собой L-атрибутное определение, работающее с базовыми типами и массивами.

5.3.1 Построение синтаксических деревьев

Как говорилось в разделе 2.8.2, каждый узел синтаксического дерева представляет конструкцию; его дочерние узлы представляют значащие компоненты конструкции. Синтаксическое дерево, представляющее выражение $E_1 + E_2$, имеет метку + и два дочерних узла, представляющих подвыражения E_1 и E_2.

Мы реализуем узлы синтаксического дерева при помощи объектов с соответствующим количеством полей. Каждый объект будет иметь поле *op*, являющееся меткой узла. У объектов будут следующие дополнительные поля.

- Если узел является листом, дополнительное поле хранит лексическое значение листа. Конструктор $Leaf(op, val)$ создает объект листа. В качестве альтернативы, если узлы рассматриваются как записи, *Leaf* возвращает указатель на новую запись для листа.

- Если узел — внутренний, то в нем имеется столько дополнительных полей, сколько у него дочерних узлов в синтаксическом дереве. Конструктор *Node* получает два или больше аргументов: $Node\,(op, c_1, c_2, \ldots, c_k)$ создает объект с первым полем *op* и k дополнительными полями для k дочерних узлов $c_1, \ldots, c_k$.

Пример 5.11. S-атрибутное определение на рис. 5.10 строит синтаксическое дерево для простой грамматики выражений, включающей только два бинарных оператора: + и −. Как обычно, эти операторы имеют один и тот же приоритет и совместно левоассоциативны. Все нетерминалы имеют один синтезируемый атрибут *node*, который представляет узел синтаксического дерева.

Каждый раз при использовании первой продукции, $E \to E_1 + T$, ее правило создает узел с '+' в качестве *op* и двумя дочерними узлами $E_1.node$ и $T.node$ для подвыражений. Вторая продукция имеет похожее правило.

В случае продукции 3, $E \to T$, узел не создается, поскольку $E.node$ имеет то же значение, что и $T.node$. Аналогично не создается узел и при использовании

	Продукция	Семантическое правило
1)	$E \to E_1 + T$	$E.node = \mathbf{new}\ Node('+', E_1.node, T.node)$
2)	$E \to E_1 - T$	$E.node = \mathbf{new}\ Node('-', E_1.node, T.node)$
3)	$E \to T$	$E.node = T.node$
4)	$T \to (E)$	$T.node = E.node$
5)	$T \to \mathbf{id}$	$T.node = \mathbf{new}\ Leaf(\mathbf{id}, \mathbf{id}.entry)$
6)	$T \to \mathbf{num}$	$T.node = \mathbf{new}\ Leaf(\mathbf{num}, \mathbf{num}.val)$

Рис. 5.10. Построение синтаксических деревьев для простых выражений

продукции 4, $T \to (E)$. Значение $T.node$ то же, что и значение $E.node$, поскольку скобки используются для группирования; они влияют на структуру дерева разбора и синтаксического дерева, но после выполнения своих функций их присутствие в синтаксическом дереве не является необходимым.

Последние две T-продукции имеют справа один терминал. Мы используем конструктор *Leaf* для создания соответствующего узла, который становится значением $T.node$.

На рис. 5.11 показано построение синтаксического дерева для выражения $a - 4 + c$. Узлы синтаксического дерева показаны как записи с первым полем *op*. Здесь ребра синтаксического дерева показаны сплошными линиями, а лежащее в основе дерево разбора, построение которого в явном виде не требуется, — линиями, состоящими из точек. Третий вид линий — пунктирные — представляет значения $E.node$ и $T.node$; каждая линия указывает на соответствующий узел синтаксического дерева.

Внизу находятся листья для a, 4 и c, построенные с применением конструктора *Leaf*. Считаем, что лексическое значение **id**.*entry* указывает на запись в таблице символов, а лексическое значение **num**.*val* представляет собой числовую константу. Эти листья, или указатели на них, становятся значениями $T.node$ трех узлов дерева разбора, помеченных T, согласно правилам 5 и 6. Заметим, что в соответствии с правилом 3 указатель на лист a является также значением $E.node$ для крайнего слева E в дереве разбора.

Правило 2 заставляет создать узел с *op*, равным знаку −, и указателями на два первых листа. Затем правило 1 создает корневой узел синтаксического дерева, объединяя узел для − с третьим листом.

Если правила вычисляются в процессе обратного обхода дерева разбора или при свертках в процессе восходящего синтаксического анализа, то последовательность шагов, показанная на рис. 5.12, заканчивается указателем p_5, указывающим на корень построенного синтаксического дерева. □

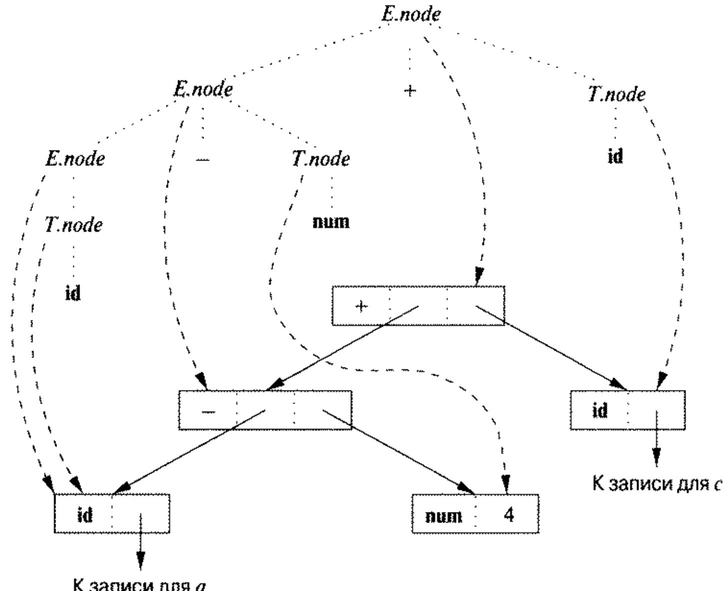


Рис. 5.11. Синтаксическое дерево для $a - 4 + c$

$$
\begin{array}{ll}
1) & p_1 = \textbf{new}\ Leaf(\textbf{id}, entry\text{-}a)\,; \\
2) & p_2 = \textbf{new}\ Leaf(\textbf{num}, 4)\,; \\
3) & p_3 = \textbf{new}\ Node\,('-', p_1, p_2)\,; \\
4) & p_4 = \textbf{new}\ Leaf(\textbf{id}, entry\text{-}c)\,; \\
5) & p_5 = \textbf{new}\ Node\,('+', p_3, p_4)\,;
\end{array}
$$

Рис. 5.12. Шаги построения синтаксического дерева для $a - 4 + c$

В случае грамматики для нисходящего синтаксического анализа строятся те же синтаксические деревья с использованием той же последовательности шагов, несмотря на то что структура деревьев разбора существенно отличается от структуры синтаксических деревьев.

Пример 5.12. L-атрибутное определение на рис. 5.13 выполняет ту же трансляцию, что и S-атрибутное определение на рис. 5.10. Атрибуты для грамматических символов E, T, **id** и **num** рассмотрены в примере 5.11.

Правила для построения синтаксических деревьев в этом примере подобны правилам для настольного калькулятора из примера 5.3. В примере с калькулятором член $x * y$ вычислялся путем передачи x как наследуемого атрибута, поскольку x и $*y$ находятся в разных частях дерева разбора. Здесь идея состоит в том, чтобы построить синтаксическое дерево для $x + y$, передавая x в качестве насле-

	Продукция	Семантические правила
1)	$E \rightarrow T\ E'$	$E.node = E'.syn$
		$E'.inh = T.node$
2)	$E' \rightarrow +\ T\ E'_1$	$E'_1.inh = \textbf{new}\ Node\,('+', E'.inh, T.node)$
		$E'.syn = E'_1.syn$
3)	$E' \rightarrow -\ T\ E'_1$	$E'_1.inh = \textbf{new}\ Node\,('-', E'.inh, T.node)$
		$E'.syn = E'_1.syn$
4)	$E' \rightarrow \epsilon$	$E'.syn = E'.inh$
5)	$T \rightarrow (E)$	$T.node = E.node$
6)	$T \rightarrow \textbf{id}$	$T.node = \textbf{new}\ Leaf(\textbf{id}, \textbf{id}.entry)$
7)	$T \rightarrow \textbf{num}$	$T.node = \textbf{new}\ Leaf(\textbf{num}, \textbf{num}.val)$

Рис. 5.13. Построение синтаксических деревьев в процессе нисходящего синтаксического анализа

дуемого атрибута, поскольку x и $+y$ находятся в разных поддеревьях. Нетерминал E' является копией нетерминала T' из примера 5.3. Сравните граф зависимостей для $a - 4 + c$ на рис. 5.14 с графом зависимостей для $3 * 5$ на рис. 5.7.

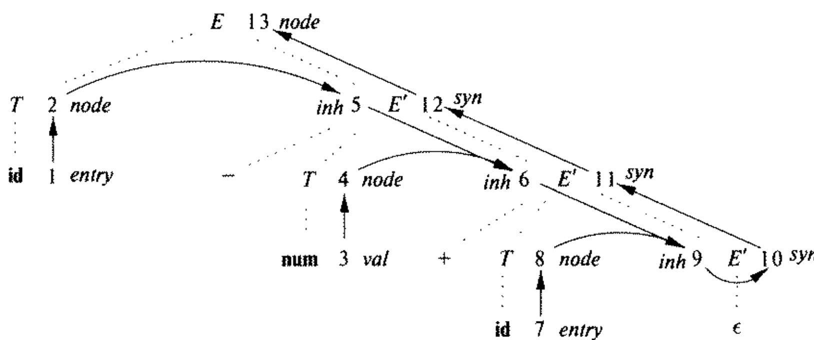


Рис. 5.14. Граф зависимостей для выражения $a - 4 + c$ при использовании СУО на рис. 5.13

Нетерминал E' имеет наследуемый атрибут *inh* и синтезируемый атрибут *syn*. Атрибут $E'.inh$ представляет неполное синтаксическое дерево, построенное к этому моменту. А именно, он представляет корень дерева для префикса входной строки, находящегося слева от поддерева для E'. В узле 5 графа зависимостей на рис. 5.14 $E'.inh$ обозначает корень неполного синтаксического дерева для идентификатора a, т.е. попросту лист для a. В узле 6 $E'.inh$ обозначает корень неполного

синтаксического дерева для входной строки $a - 4$. В узле 9 $E'.inh$ обозначает корень синтаксического дерева для $a - 4 + c$.

Поскольку на этом входная строка заканчивается, в узле 9 $E'.inh$ указывает на корень всего синтаксического дерева. Атрибут *syn* передает это значение обратно вверх по дереву разбора, пока оно не становится значением $E.node$. Точнее, значение атрибута в узле 10 определяется правилом $E'.syn = E'.inh$, связанным с продукцией $E' \to \epsilon$. Значение атрибута в узле 11 определяется правилом $E'.syn = E'_1.syn$, связанным с продукцией 2 на рис. 5.13. Аналогичные правила определяют значения атрибутов в узлах 12 и 13. □

5.3.2 Структура типа

Наследуемые атрибуты полезны, когда структура дерева разбора отличается от абстрактного синтаксиса входной строки; тогда атрибуты могут использоваться для передачи информации из одной части дерева разбора в другую. Приведенный далее пример показывает, что несоответствие структуры может являться следствием дизайна языка, но не ограничений, накладываемых методом синтаксического анализа.

Пример 5.13. В C тип **int** [2] [3] можно прочесть как "массив из двух массивов по 3 целых числа". Соответствующее выражение типа $array(2, array(3, integer))$ представлено деревом на рис. 5.15. Оператор *array* принимает два параметра, число и тип. Если типы представлены деревьями, то этот оператор возвращает дерево с меткой *array* с двумя дочерними узлами — для числа и для типа.

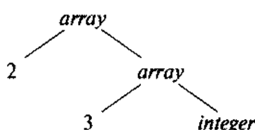


Рис. 5.15. Выражение типа для **int** [2] [3]

При использовании СУО, представленного на рис. 5.16, нетерминал T генерирует либо фундаментальный тип, либо тип массива. Нетерминал B генерирует один из базовых типов: **int** или **float**. T генерирует фундаментальный тип, если T порождает $B\ C$, а C порождает ϵ. В противном случае C генерирует компоненты массива, состоящие из последовательности целых чисел, каждое из которых заключено в квадратные скобки.

Нетерминалы B и T имеют синтезируемый атрибут t, представляющий тип. Нетерминал C имеет два атрибута: наследуемый атрибут b и синтезируемый атрибут t. Наследуемые атрибуты b передают фундаментальный тип вниз по дереву, а синтезируемые атрибуты t накапливают результат.

Продукция	Семантические правила
$T \to B\ C$	$T.t = C.t$
	$C.b = B.t$
$B \to$ **int**	$B.t = integer$
$B \to$ **float**	$B.t = float$
$C \to$ [**num**] C_1	$C.t = array\,(\mathbf{num}.val, C_1.t)$
	$C_1.b = C.b$
$C \to \epsilon$	$C.t = C.b$

Рис. 5.16. T генерирует либо фундаментальный тип, либо тип массива

Аннотированное дерево разбора для входной строки **int** [2] [3] показано на рис. 5.17. Соответствующее выражение типа на рис. 5.15 построено путем передачи типа *integer* от B вниз по цепочке C посредством наследуемых атрибутов b. Тип массива синтезируется при продвижении вверх по цепочке C посредством атрибутов t.

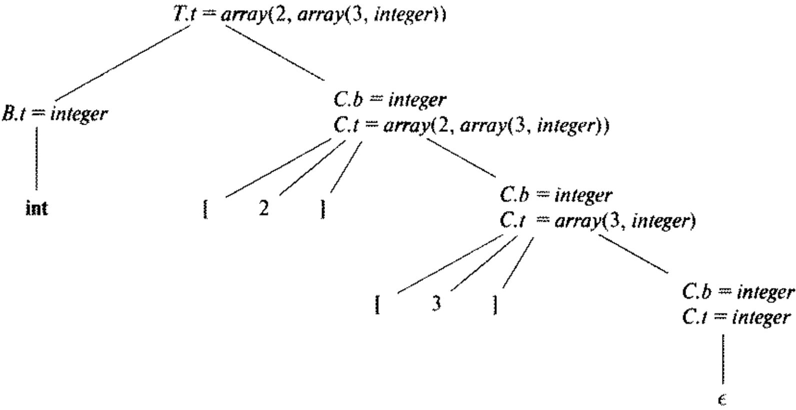


Рис. 5.17. Синтаксически управляемая трансляция типов массивов

Говоря более подробно, в корне для $T \to B\ C$ нетерминал C наследует тип от B при помощи наследуемого атрибута $C.b$. В крайнем справа узле для C используется продукция $C \to \epsilon$, так что $C.t$ равно $C.b$. Семантические правила для продукции $C \to$ [**num**] C_1 образуют $C.t$ путем применения оператора *array* к операндам **num**.*val* и $C_1.t$. □

5.3.3 Упражнения к разделу 5.3

Упражнение 5.3.1. Ниже приведена грамматика для выражений, включающих оператор + и операнды, представляющие собой целые числа либо числа с плавающей точкой (числа с плавающей точкой распознаются по наличию десятичной точки):

$$\begin{array}{lcl} E & \rightarrow & E + T \mid T \\ T & \rightarrow & \textbf{num . num} \mid \textbf{num} \end{array}$$

а) Разработайте СУО для определения типа каждого члена T и выражения E.

б) Распространите свое СУО из п. *a* на выражения в постфиксной записи. Воспользуйтесь унарным оператором **intToFloat** для преобразования целого числа в эквивалентное число с плавающей точкой.

! **Упражнение 5.3.2.** Разработайте СУО для трансляции инфиксных выражений с + и * в эквивалентные выражения без излишних скобок. Например, поскольку оба оператора левоассоциативны, а приоритет * выше приоритета +, $((a*(b+c))*(d))$ транслируется в $a*(b+c)*d$.

! **Упражнение 5.3.3.** Разработайте СУО для дифференцирования выражений наподобие $x*(3*x+x*x)$, которые включают операторы + и *, переменную x и константы. Будем считать, что никакие упрощения не выполняются, т.е. $3*x$, например, транслируется в $3*1+0*x$.

5.4 Синтаксически управляемые схемы трансляции

Синтаксически управляемые схемы трансляции представляют собой запись, дополняющую синтаксически управляемые определения. Все применения синтаксически управляемых определений в разделе 5.3 могут быть реализованы с использованием синтаксически управляемых схем трансляции.

Из раздела 2.3.5 мы знаем, что *синтаксически управляемая схема трансляции* (СУТ) представляет собой контекстно-свободную грамматику с программными фрагментами, внедренными в тела продукций. Эти фрагменты называются *семантическими действиями* и могут находиться в любой позиции в теле продукции. По соглашению действия располагаются внутри фигурных скобок; фигурные скобки, являющиеся грамматическими символами, заключаются в кавычки.

Любая СУТ может быть реализована путем построения дерева разбора с последующим выполнением действий в порядке в глубину слева направо, т.е. в порядке прямого обхода дерева. Соответствующий пример приведен в разделе 5.4.3.

Обычно СУТ реализуется в процессе синтаксического анализа, без построения дерева разбора. В данном разделе мы остановимся на использовании СУТ для реализации двух важных классов СУО.

1. Грамматика поддается LR-синтаксическому анализу, а СУО — S-атрибутное.

2. Грамматика поддается LL-синтаксическому анализу, а СУО — L-атрибутное.

Мы увидим, каким образом в обоих случаях семантические правила СУО могут быть преобразованы в СУТ с действиями, выполняемыми в нужный момент. В процессе синтаксического анализа действие в теле продукции выполняется, как только все грамматические символы слева от него сопоставлены входной строке.

СУТ, которые могут быть реализованы в процессе синтаксического анализа, можно охарактеризовать путем вставки вместо действий отличающихся друг от друга нетерминалов-маркеров; каждый маркер M имеет единственную продукцию $M \to \epsilon$. Если синтаксический анализ грамматики с такими нетерминалами-маркерами может быть выполнен некоторым методом, то СУТ может быть реализована в процессе данного синтаксического анализа.

5.4.1 Постфиксные схемы трансляции

Простейшая реализация СУТ имеет место в случае восходящего синтаксического анализа и S-атрибутного СУО. В этом случае можно построить СУТ, в которой каждое действие размещается в конце продукции и выполняется вместе со сверткой тела продукции в заголовок. СУТ со всеми действиями, расположенными на правом конце тел продукций, называются *постфиксными СУТ*.

Пример 5.14. Постфиксная СУТ на рис. 5.18 реализует СУО настольного калькулятора, представленного на рис. 5.1, с единственным изменением: действие первой продукции выводит вычисленное значение. Поскольку лежащая в основе СУТ грамматика является LR-грамматикой, а СУО — S-атрибутное, эти действия могут корректно выполняться синтаксическим анализатором при свертках. □

5.4.2 Реализация постфиксной СУТ с использованием стека синтаксического анализатора

Постфиксная СУТ может быть реализована в процессе LR-синтаксического анализа путем выполнения действий при свертках. Атрибут (или атрибуты) каждого грамматического символа может помещаться в стек в том месте, где он может быть обнаружен в процессе свертки. Лучше всего разместить атрибуты вместе с грамматическими символами (или LR-состояниями, представляющими эти символы) в записях стека.

$$
\begin{array}{ll}
L \to E\ \mathbf{n} & \{print\,(E.val)\,;\} \\
E \to E_1 + T & \{E.val = E_1.val + T.val;\} \\
E \to T & \{E.val = T.val;\} \\
T \to T_1 * F & \{T.val = T_1.val \times F.val;\} \\
T \to F & \{T.val = F.val;\} \\
F \to (E) & \{F.val = E.val;\} \\
F \to \mathbf{digit} & \{F.val = \mathbf{digit}.lexval;\}
\end{array}
$$

Рис. 5.18. Постфиксная СУТ, реализующая настольный калькулятор

На рис. 5.19 стек синтаксического анализатора содержит записи с полем для грамматических символов (или состояний синтаксического анализатора), а под ним – поле для атрибутов. На вершине стека находятся три грамматических символа – $X\ Y\ Z$; возможно, они будут свернуты в соответствии с продукцией наподобие $A \to X\ Y\ Z$. На рисунке показан атрибут $X.x$ грамматического символа X и т.д. В общем случае могут иметься несколько атрибутов; в этом случае следует либо увеличить размер записи в стеке так, чтобы она могла содержать все атрибуты, либо поместить в стек указатели на соответствующие записи. В случае атрибутов небольшого размера и в небольшом количестве может оказаться проще увеличить размер записи в стеке, даже если некоторые поля будут время от времени пустовать. Но если один или несколько атрибутов имеют неограниченный размер – например, представляют собой строки, – то лучше помещать в стек указатель на значение атрибута, который хранится в некотором другом месте памяти, не являющейся частью стека.

Рис. 5.19. Стек синтаксического анализатора с полем для синтезируемых атрибутов

Если все атрибуты синтезируемые, а действия располагаются в конце продукций, то можно вычислять атрибуты заголовков при выполнении свертки тела продукции к заголовку. Если выполняется свертка в соответствии с продукцией, такой как $A \to X\ Y\ Z$, то все атрибуты X, Y и Z оказываются доступны и находятся в известных позициях в стеке, как показано на рис. 5.19. После выполнения действия A и его атрибуты находятся на вершине стека, в позиции, которую занимала запись для X.

Пример 5.15. Перепишем действия из СУТ настольного калькулятора из примера 5.14 так, чтобы они явно работали со стеком. Такая работа со стеком обычно выполняется синтаксическим анализатором автоматически.

Предположим, что стек поддерживается в виде массива записей с именем *stack*, с курсором *top*, указывающим на вершину стека. Таким образом, $stack\,[top]$ представляет собой запись на вершине стека, $stack\,[top-1]$ — запись под ней и т.д. Предположим также, что каждая запись имеет поле с именем *val*, в котором хранится атрибут грамматического символа, представленного данной записью. Таким образом, обратиться к атрибуту $E.val$, находящемуся в третьей позиции стека, можно как к $stack\,[top-2]\,.val$. Полностью СУТ показана на рис. 5.20.

Продукция	Действия
$L \to E$ **n**	{ print($stack\,[top-1].val$); $top = top - 1$; }
$E \to E_1 + T$	{ $stack\,[top-2].val = stack\,[top-2].val + stack\,[top].val$; $top = top - 2$; }
$E \to T$	
$T \to T_1 * F$	{ $stack\,[top-2].val = stack\,[top-2].val \times stack\,[top].val$; $top = top - 2$; }
$T \to F$	
$F \to (\ E\ )$	{ $stack\,[top-2].val = stack\,[top-1].val$; $top = top - 2$; }
$F \to$ **digit**	

Рис. 5.20. Реализация настольного калькулятора в стеке восходящего синтаксического анализа

Например, во второй продукции, $E \to E_1 + T$, значение E_1 расположено на две позиции ниже вершины стека, в то время как значение T находится на вершине стека. Вычисленная сумма помещается в то место, в котором после свертки находится заголовок E, т.е. двумя позициями ниже текущей вершины стека. Причина этого в том, что после свертки три грамматических символа на вершине стека будут заменены одним. После вычисления $E.val$ со стека снимаются два символа, так что запись, в которую было помещено значение $E.val$, после этого окажется на вершине стека.

В третьей продукции $E \to T$ действие не является необходимым, поскольку размер стека не изменяется и значение атрибута $T.val$ на вершине стека просто станет значением $E.val$. Те же самые рассуждения применимы и к продукциям

$T \to F$ и $F \to$ **digit**. Продукция $F \to (E)$ несколько отличается тем, что, хотя значение атрибута и не изменяется, при свертке со стека снимаются два элемента, так что значение атрибута надо перенести в позицию, которая окажется на вершине стека после свертки.

Заметим, что здесь опущены шаги, управляющие первым полем записей в стеке — полем, которое содержит LR-состояние или представляет грамматический символ. При выполнении LR-синтаксического анализа таблица анализа говорит нам о том, в какое новое состояние будет выполнена свертка (см. алгоритм 4.44). Таким образом, можно просто поместить состояние в запись на новой вершине стека. □

5.4.3 СУТ с действиями внутри продукций

Действия могут находиться в любой позиции в теле продукции. Действие выполняется сразу же после того, как обработаны все символы слева от него. Таким образом, если у нас есть продукция $B \to X\ \{a\}\ Y$, то действие a выполняется после того, как распознан X (если X — терминал) или все терминалы, порожденные из X (если X — нетерминал). Говоря более точно,

- при восходящем синтаксическом анализе действие a выполняется, как только данный грамматический символ X оказывается на вершине стека;
- при нисходящем синтаксическом анализе действие a выполняется непосредственно перед тем, как выполняется раскрытие данного Y (если Y — нетерминал) или проверяется его наличие во входной строке (если Y — терминал).

СУТ, которые могут быть реализованы в процессе синтаксического анализа, включают постфиксные СУТ и класс СУТ, рассматриваемый в разделе 5.5, который реализует L-атрибутные определения. Как вы узнаете из приведенного ниже примера, не все СУТ могут быть реализованы в процессе синтаксического анализа.

Пример 5.16. В качестве крайнего примера проблемной СУТ предположим, что мы преобразовали наш калькулятор в СУТ, которая печатает выражения в префиксном виде, а не вычисляет их значения. Продукции и действия такой СУТ показаны на рис. 5.21.

К сожалению, эту СУТ невозможно реализовать в процессе нисходящего или восходящего синтаксического анализа, поскольку синтаксический анализатор должен выполнять критические действия, такие как вывод экземпляра * или +, задолго до того, как станет известно о его наличии во входном потоке.

Использование нетерминалов-маркеров M_2 и M_4 для действий в продукциях соответственно 2 и 4 приводит к тому, что ПС-синтаксический анализатор (см.

1) $L \to E\ \mathbf{n}$
2) $K \to \{print\,('+');\}\ E_1 + T$
3) $E \to T$
4) $T \to \{print\,('*');\}\ T_1 * F$
5) $T \to F$
6) $F \to (E)$
7) $F \to \mathbf{digit}\ \{print\,(\mathbf{digit}.lexval);\}$

Рис. 5.21. Проблемная СУТ для трансляции инфиксных выражений в префиксные в процессе синтаксического анализа

раздел 4.5.3) при входном символе, например, 3 обнаруживает конфликт между сверткой согласно $M_2 \to \epsilon$, сверткой согласно $M_4 \to \epsilon$ и переносом обнаруженной во входном потоке цифры. □

Любая СУТ может быть реализована следующим образом.

1. Игнорируя действия, выполняется синтаксический анализ входного потока и строится дерево разбора.

2. Проверяется каждый внутренний узел N, скажем, для продукции $A \to \alpha$. Добавляем к N дополнительные дочерние узлы для действий из α таким образом, что дочерние узлы N слева направо составляют символы и действия α.

3. Выполняется обход дерева в прямом порядке (см. раздел 2.3.4) и при посещении узла, помеченного действием, выполняется это действие.

Например, на рис. 5.22 показано дерево разбора со вставленными действиями для выражения 3*5+4. При посещении узлов дерева в прямом порядке получается префиксная запись выражения: $+ * 3\ 5\ 4$.

5.4.4 Устранение левой рекурсии из СУТ

Поскольку ни одна грамматика с левой рекурсией не может быть детерминированно проанализирована нисходящим синтаксическим анализатором, нам надо устранять из грамматики левую рекурсию (с этим действием мы уже встречались в разделе 4.3.3). Когда грамматика является частью СУТ, при устранении левой рекурсии необходимо побеспокоиться также и о действиях.

Начнем с рассмотрения простого случая, в котором нас интересует только порядок выполнения действий в СУТ. Например, если каждое действие состоит в печати строки, то нас интересует только порядок, в котором будут напечатаны эти строки. В этом случае можно руководствоваться следующим принципом.

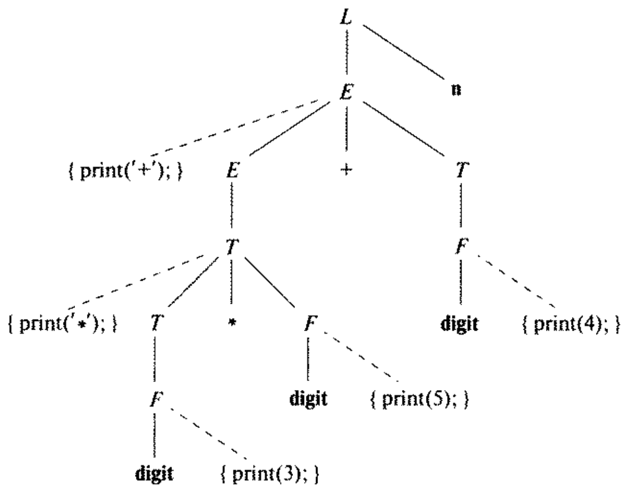


Рис. 5.22. Дерево разбора с добавленными действиями

- При внесении изменений в грамматику следует рассматривать действия так, как если бы они были терминальными символами.

Этот принцип основан на той идее, что преобразования грамматики сохраняют порядок терминалов в генерируемой строке. Действия, таким образом, выполняются в одном и том же порядке при любом синтаксическом анализе слева направо, как нисходящем, так и восходящем.

"Трюк" устранения левой рекурсии состоит в том, что мы берем продукции

$$A \quad \rightarrow \quad A\alpha \mid \beta$$

Они генерируют строки, состоящие из β и произвольного количества α, и заменяем их продукциями, которые генерируют те же строки с использованием нового нетерминала R:

$$\begin{array}{lcl} A & \rightarrow & \beta R \\ R & \rightarrow & \alpha R \mid \epsilon \end{array}$$

Если β не начинается с A, то A больше не имеет леворекурсивной продукции. В терминах регулярных определений в обоих множествах продукций A определяется как $\beta\,(\alpha)^*$. Как справиться с ситуациями, когда A имеет больше рекурсивных или нерекурсивных продукций, описано в разделе 4.3.3.

Пример 5.17. Рассмотрим следующие E-продукции из СУТ для преобразования инфиксных выражений в постфиксную запись:

$$\begin{array}{lcl} E & \rightarrow & E_1 + T \quad \{print\,('+');\} \\ E & \rightarrow & T \end{array}$$

Если мы применим стандартное преобразование к E, то остатком леворекурсивной продукции будет

$$\alpha = +T \quad \{print\,('+')\,;\}$$

А β, тело другой продукции, – T. Если мы введем R как остаток E, то получим следующее множество продукций:

$$\begin{array}{lcl} E & \rightarrow & T\,R \\ R & \rightarrow & +T\ \{print\,('+')\,;\} \\ R & \rightarrow & \epsilon \end{array}$$

□

Если же действия СУО не просто выводят строки, но и вычисляют атрибуты, то следует быть более осторожным при устранении левой рекурсии из грамматики. Однако, если СУО является S-атрибутным, то мы всегда можем построить СУТ путем размещения действий по вычислению атрибутов в соответствующих позициях в новых продукциях.

Приведем общую схему для случая одной рекурсивной продукции, одной нерекурсивной продукции и одного атрибута леворекурсивного нетерминала; обобщение на случай многих продукций каждого типа не сложное, но весьма громоздкое. Предположим, что эти две продукции –

$$\begin{array}{lcl} A & \rightarrow & A_1\,Y \quad \{A.a = g\,(A_1.a, Y.y)\} \\ A & \rightarrow & X \quad \{A.a = f\,(X.x)\} \end{array}$$

Здесь $A.a$ – синтезируемый атрибут леворекурсивного нетерминала A, а X и Y – отдельные грамматические символы с синтезируемыми атрибутами $X.x$ и $Y.y$ соответственно. Они могут представлять строки из нескольких грамматических символов, каждый со своими атрибутами, поскольку схема содержит произвольную функцию g, вычисляющую $A.a$ в рекурсивной продукции, и произвольную функцию f, вычисляющую $A.a$ в другой продукции. В любом случае f и g получают в качестве аргументов атрибуты, доступ к которым разрешен в случае S-атрибутного СУО.

Мы хотим преобразовать лежащую в основе грамматику в

$$\begin{array}{lcl} A & \rightarrow & X\,R \\ R & \rightarrow & Y\,R \mid \epsilon \end{array}$$

На рис. 5.23 показано, что должна делать СУТ на основе новой грамматики. На рис. 5.23, *а* мы видим результат работы постфиксной СУТ на основе исходной грамматики. Здесь один раз применяется функция f, соответствующая использованию продукции $A \rightarrow X$, а затем функция g применяется столько раз, сколько раз используется продукция $A \rightarrow A\,Y$. Поскольку R генерирует "остаток" строки из Y, его трансляция зависит от строки слева от этого грамматического символа, т.е.

от строки вида $XYY \cdots Y$. Каждое использование продукции $R \to Y\ R$ приводит к применению функции g. В случае R мы используем наследуемый атрибут $R.i$ для накопления результата последовательного применения функций g, начиная со значения атрибута $A.a$.

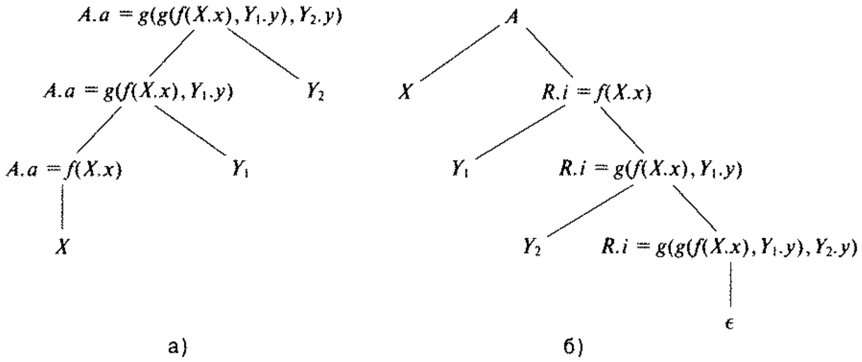


Рис. 5.23. Устранение левой рекурсии из постфиксной СУТ

Кроме того, R имеет синтезируемый атрибут $R.s$, не показанный на рис. 5.23. Этот атрибут в первый раз вычисляется, когда R завершает генерацию символов Y, о чем свидетельствует использование продукции $R \to \epsilon$. Затем $R.s$ копируется вверх по дереву, так что он становится значением $A.a$ для всего выражения $XYY \cdots Y$. На рис. 5.23 показана ситуация, когда A генерирует строку XYY, и видно, что значение $A.a$ в корне дерева на рис. 5.23, *а* включает два использования функции g. То же самое можно сказать и о $R.i$ в нижней части дерева на рис. 5.23, *б*; это значение уже как значение $R.s$ затем копируется вверх по дереву.

Итак, для завершения трансляции используется следующая СУТ:

$$\begin{array}{lcl} A & \to & X\ \{R.i = f(X.x)\}\ \ R\ \ \{A.a = R.s\} \\ R & \to & Y\ \{R_1.i = g(R.i, Y.y)\}\ \ R_1\ \ \{R.s = R_1.s\} \\ R & \to & \epsilon\ \{R.s = R.i\} \end{array}$$

Обратите внимание, что наследуемый атрибут $R.i$ вычисляется непосредственно перед использованием R в теле продукции, в то время как синтезируемые атрибуты $A.a$ и $R.s$ вычисляются в конце продукций. Таким образом, все значения, необходимые для вычисления этих атрибутов, будут доступны из вычислений, проведенных в телах продукций слева.

5.4.5 СУТ для L-атрибутных определений

В разделе 5.4.1 мы преобразовали S-атрибутное СУО в постфиксную СУТ с действиями на правом конце продукций. Поскольку лежащая в основе СУТ

грамматика принадлежит классу LR, постфиксная СУТ может анализироваться и транслироваться снизу вверх.

Рассмотрим теперь более общий случай L-атрибутного СУО. Будем считать, что грамматика поддается нисходящему синтаксическому анализу, поскольку, если это не так, зачастую трансляцию невозможно выполнить ни LL-, ни LR-синтаксическим анализатором. Для произвольной грамматики применим метод, состоящий в присоединении действий к дереву разбора и выполнении их в процессе обхода дерева в прямом порядке.

Вот правила, используемые при превращении L-атрибутного СУО в СУТ.

1. Вставить действие, которое вычисляет наследуемые атрибуты нетерминала A, непосредственно перед вхождением A в тело продукции. Если несколько наследуемых атрибутов A ациклически зависят друг от друга, следует упорядочить вычисление атрибутов с тем, чтобы сначала вычислялись те атрибуты, которые требуются первыми.

2. Поместить действия, вычисляющие синтезируемые атрибуты заголовка продукции, в конце тела соответствующей продукции.

Проиллюстрируем эти принципы на двух расширенных примерах. Первый относится к области полиграфии и иллюстрирует, как методы компиляции могут использоваться в приложениях, далеких от того, что обычно представляется при словосочетании "язык программирования". Второй пример связан с генерацией промежуточного кода для типичной конструкции языка программирования — цикла while.

Пример 5.18. Этот пример вдохновлен языками для набора математических формул. Одним из первых таких языков был язык `Eqn`; ряд идей из `Eqn` можно обнаружить и в системе TEX, которая использовалась при подготовке данной книги.

Мы сосредоточимся только на возможности определения нижних индексов, индексов у индексов и т.д., игнорируя верхние индексы, встроенные дроби и прочие особенности математических формул. В `Eqn` строка `a sub i sub j` указывала на выражение a_{i_j}. Простейшей грамматикой для *боксов* (элементов текста, ограниченных прямоугольником) является

$$B \rightarrow B_1\ B_2 \mid B_1\ \textbf{sub}\ B_2 \mid (B_1) \mid \textbf{text}$$

В соответствии с приведенными четырьмя продукциями бокс может представлять собой одно из нижеперечисленного.

1. Два смежных бокса, первый из которых, B_1, находится слева от второго, B_2.

2. Бокс и бокс нижнего индекса. Второй бокс имеет меньший размер и находится ниже и правее первого бокса.

3. Бокс в скобках для группировки боксов и индексов. Eqn и TEX для группировки используют фигурные скобки, но, чтобы избежать путаницы с фигурными скобками, указывающими действия в СУТ, здесь для группировки будут использоваться круглые скобки.

4. Текстовая строка, т.е. произвольная строка символов.

Эта грамматика неоднозначна, но ее можно использовать для восходящего синтаксического анализа при условии правоассоциативности отношения смежности и индексирования, причем приоритет оператора **sub** выше приоритета смежности.

Выражение выводится путем построения бóльших боксов, окружающих меньшие. На рис. 5.24 смежные боксы для E_1 и *.height* образуют бокс для E_1*.height*. Левый бокс — для E_1 — состоит из боксов для E и для индекса 1. Индекс 1 обрабатывается путем уменьшения его размера примерно на 30%, опускания его вниз и размещения после бокса для E. Хотя *.height* рассматривается нами как текстовая строка, прямоугольники в ее боксе показывают, каким образом он может быть построен из боксов для отдельных букв.

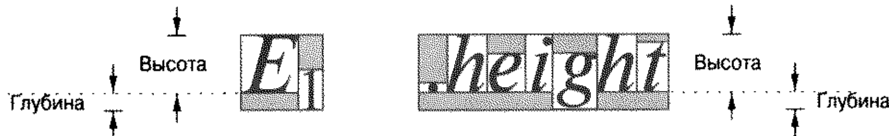


Рис. 5.24. Построение больших боксов из меньших

В этом примере мы остановимся только на вертикальной геометрии боксов. Горизонтальная геометрия — ширина боксов — также представляет интерес, в особенности в связи с тем, что различные символы имеют разную ширину. Возможно, это не сразу бросается в глаза, но каждый из различных символов на рис. 5.24 имеет свою ширину, отличную от ширины других символов.

С вертикальной геометрией боксов связаны следующие значения.

а) *Кегль* (point size) используется для размещения текста в боксе. Будем считать, что символы, не являющиеся символами индексов, имеют кегль 10. Далее, считаем, что если бокс имеет кегль p, то кегль его индекса — $0.7p$. Наследуемый атрибут $B.ps$ представляет кегль блока B. Этот атрибут должен быть наследуемым, поскольку то, насколько данный бокс должен быть меньше основного, определяется его уровнем индексирования.

б) Каждый бокс имеет *базовую линию*, или базис (baseline), представляющий собой вертикальное положение, соответствующее низу текста (без учета букв наподобие “g”, части которых опускаются ниже базовой линии). На рис. 5.24 базовая линия для E, *.height* и всего выражения в целом показана

точками. Базовая линия индекса (на рисунке — базис бокса, содержащего 1) находится ниже основной базовой линии.

в) Бокс имеет *высоту* (height), представляющую собой расстояние от базовой линии до вершины бокса. Высота бокса B определяется его синтезируемым атрибутом $B.ht$.

г) Бокс имеет *глубину* (depth), представляющую собой расстояние от базовой линии до дна бокса. Глубина бокса B определяется его синтезируемым атрибутом $B.dp$.

СУО на рис. 5.25 содержит правила для вычисления кегля, высоты и глубины боксов. Продукция 1 используется для назначения атрибуту $B.ps$ начального значения кегля 10.

Продукция	Семантические правила
1) $S \to B$	$B.ps = 10$
2) $B \to B_1\ B_2$	$B_1.ps = B.ps$
	$B_2.ps = B.ps$
	$B.ht = \max(B_1.ht, B_2.ht)$
	$B.dp = \max(B_1.dp, B_2.dp)$
3) $B \to B_1$ **sub** B_2	$B_1.ps = B.ps$
	$B_2.ps = 0.7 \times B.ps$
	$B.ht = \max(B_1.ht, B_2.ht - 0.25 \times B.ps)$
	$B.dp = \max(B_1.dp, B_2.dp + 0.25 \times B.ps)$
4) $B \to (B_1)$	$B_1.ps = B.ps$
	$B.ht = B_1.ht$
	$B.dp = B_1.dp$
5) $B \to$ **text**	$B.ht = getHt(B.ps, \textbf{text}.lexval)$
	$B.dp = getDp(B.ps, \textbf{text}.lexval)$

Рис. 5.25. СУО для текстовых боксов

Продукция 2 обрабатывает смежные боксы. Кегль копируется вниз по дереву разбора, т.е. два подбокса основного бокса наследуют тот же кегль, что и у большего бокса. Высота и глубина вычисляются вверх по дереву при помощи функции поиска максимального значения, т.е. высота большего бокса представляет собой максимальную из высот своих двух компонентов; то же касается и глубины бокса.

Продукция 3 обрабатывает индексы, что является немного более сложной задачей, чем предыдущая. В этом очень упрощенном примере считается, что кегль ин-

декса составляет 70% от родительского. Реальность существенно более сложная, поскольку индексы не могут уменьшаться до бесконечности; на практике после нескольких уровней размеры индексов перестают уменьшаться. Далее, в примере полагается, что базовая линия бокса индекса опущена на 25% от размера родителя; ситуация в реальности существенно сложнее, чем эта упрощенная модель.

Продукция 4 копирует атрибуты при использовании скобок. Наконец, продукция 5 обрабатывает листья, представляющие боксы текста. Здесь опять же реальная ситуация весьма сложна, так что в СУО показаны две не определенные функции *getHt* и *getDp*, которые работают с таблицами, указывающими для каждого шрифта максимальную высоту и максимальную глубину любого символа текстовой строки. Сама по себе строка передается как атрибут *lexval* терминала **text**.

Наша последняя задача состоит в преобразовании СУО в СУТ, следуя правилам для L-атрибутных СУО, к каковым относится и СУО на рис. 5.25. Соответствующая СУТ показана на рис. 5.26. Для большей удобочитаемости ставшие весьма длинными тела продукций разбиты на несколько строк. Тело каждой продукции состоит из всех строк, расположенных выше заголовка следующей продукции. □

Продукция			Действия
1)	S	$\to$	$\{\ B.ps = 10;\ \}$
		B	
2)	B	$\to$	$\{\ B_1.ps = B.ps;\ \}$
		B_1	$\{\ B_2.ps = B.ps;\ \}$
		B_2	$\{\ B.ht = \max(B_1.ht, B_2.ht);$
			$B.dp = \max(B_1.dp, B_2.dp);\ \}$
3)	B	$\to$	$\{\ B_1.ps = B.ps;\ \}$
		B_1 **sub**	$\{\ B_2.ps = 0.7 \times B.ps;\ \}$
		B_2	$\{\ B.ht = \max(B_1.ht, B_2.ht - 0.25 \times B.ps);$
			$B.dp = \max(B_1.dp, B_2.dp + 0.25 \times B.ps);\ \}$
4)	B	$\to$ (	$\{\ B_1.ps = B.ps;\ \}$
		B_1)	$\{\ B.ht = B_1.ht;$
			$B.dp = B_1.dp;\ \}$
5)	B	$\to$ **text**	$\{\ B.ht = getHt\,(B.ps, \mathbf{text}.lexval);$
			$B.dp = getDp\,(B.ps, \mathbf{text}.lexval);\ \}$

Рис. 5.26. СУТ для текстовых боксов

Следующий пример связан с простой инструкцией while и генерацией промежуточного кода для данного типа инструкций. Промежуточный код рассматривается как атрибут со строковым значением. Позже мы познакомимся с методами, которые включают запись частей строковых атрибутов в процессе синтаксического анализа, что позволяет избегать копирования длинных строк для построения еще более длинных строк. Такого рода метод использовался в примере 5.17, при генерации постфиксной записи инфиксного выражения "на лету" вместо вычисления атрибута. Однако пока что будем создавать строковые атрибуты путем конкатенации.

Пример 5.19. В этом примере нам достаточно одной продукции:

$$S \rightarrow \textbf{while}\ (C)\ S_1$$

Здесь S — нетерминал, который генерирует все виды инструкций, вероятно, включая условные конструкции, присваивания и др. В нашем примере C представляет собой условное выражение — булево выражение, которое после вычисления принимает значение true или false.

В этом примере единственный объект, который мы будем генерировать, — это метки. Все прочие команды промежуточного кода считаются генерируемыми не показанными частями СУТ. В частности, мы генерируем явные команды вида **label** L, где L — идентификатор, указывающие, что L — метка следующей за ней команды. Предполагается, что промежуточный код подобен рассматривавшемуся в разделе 2.8.4.

Конструкция while работает следующим образом. Сначала вычисляется условное выражение C. Если оно истинно, управление передается в начало кода для S_1; если оно ложно, управление передается коду, следующему за конструкцией while. Код S_1 должен по завершении выполнения передать управление в начало конструкции while, к коду, который вычисляет значение C и который не показан на рис. 5.27.

$$
\begin{array}{ll}
S \rightarrow \textbf{while}\ (\ C\ )\ S_1 & L1 = new();\\
& L2 = new();\\
& S_1.next = L1;\\
& C.false = S.next;\\
& C.true = L2;\\
& S.code = \textbf{label} \parallel L1 \parallel C.code \parallel \textbf{label} \parallel L2 \parallel S_1.code
\end{array}
$$

Рис. 5.27. СУО для конструкции while

Для генерации корректного промежуточного кода используются следующие атрибуты.

1. Наследуемый атрибут *S.next* помечает начало кода, который должен быть выполнен после завершения S.

2. Синтезируемый атрибут *S.code* представляет собой последовательность шагов промежуточного кода, которая реализует инструкцию S и завершается переходом к метке *S.next*.

3. Наследуемый атрибут *C.true* помечает начало кода, который должен быть выполнен, если значение C истинно.

4. Наследуемый атрибут *C.false* помечает начало кода, который должен быть выполнен, если значение C ложно.

5. Синтезируемый атрибут *C.code* представляет собой последовательность шагов промежуточного кода, которая реализует условие C и переходит в зависимости от вычисленного значения C либо к метке *C.true*, либо к метке *C.false*.

СУО, вычисляющее эти атрибуты конструкции while, приведено на рис. 5.27. Некоторые моменты этого СУО требуют пояснений.

- Функция *new* генерирует новые метки.

- Переменные $L1$ и $L2$ хранят метки, которые потребуются в генерируемом коде. $L1$ представляет начало кода конструкции while, и код S_1 по окончании работы должен выполнять переход к ней. Именно поэтому выполняется присваивание $L1$ атрибуту S_1*.next*. $L2$ представляет начало кода S_1 и становится значением атрибута *C.true*, поскольку именно сюда выполняется переход, если вычисленное условие истинно.

- Обратите внимание, что *C.false* устанавливается равным *S.next*, поскольку, когда условие ложно, выполняется код, следующий за кодом S.

- Символ $\|$ используется для обозначения конкатенации фрагментов промежуточного кода. Таким образом, значение *S.code* начинается с метки $L1$, за которой следуют код условия C, другая метка $L2$ и код S_1.

Данное СУО является L-атрибутным. При его преобразовании в СУТ остается единственный вопрос — как обрабатывать метки $L1$ и $L2$, представляющие собой переменные, а не атрибуты? Если рассматривать действия как фиктивные нетерминалы, то такие переменные могут трактоваться как синтезируемые атрибуты фиктивных нетерминалов. Поскольку $L1$ и $L2$ не зависят от каких-либо других атрибутов, они могут быть назначены первому действию продукции. В результате получается СУТ со встроенными действиями, реализующая рассмотренное L-атрибутное определение и показанная на рис. 5.28. □

$$
\begin{array}{lll}
S \rightarrow \textbf{while} \; (& \{ \; L1 = new(); \; L2 = new(); \; C.false = S.next; \; C.true = L2; \; \} \\
\quad C \;) & \{ \; S_1.next = L1; \; \} \\
\quad S_1 & \{ \; S.code = \textbf{label} \; \| \; L1 \; \| \; C.code \; \| \; \textbf{label} \; \| \; L2 \; \| \; S_1.code; \; \}
\end{array}
$$

Рис. 5.28. СУТ для конструкции while

5.4.6 Упражнения к разделу 5.4

Упражнение 5.4.1. В разделе 5.4.2 упоминалось, что из LR-состояния в стеке синтаксического анализатора можно вывести, какой именно грамматический символ представлен этим состоянием. Каким образом можно это сделать?

Упражнение 5.4.2. Перепишите СУТ

$$
\begin{array}{lll}
A & \rightarrow & A \; \{a\} \; B \mid A \; B \; \{b\} \mid 0 \\
B & \rightarrow & B \; \{c\} \; A \mid B \; A \; \{d\} \mid 1
\end{array}
$$

так, чтобы лежащая в ее основе грамматика перестала быть леворекурсивной. Здесь a, b, c и d — действия, а 0 и 1 — терминалы.

! **Упражнение 5.4.3.** Приведенная ниже СУТ вычисляет значение строки из 0 и 1, рассматривая ее как положительное бинарное число.

$$
\begin{array}{lll}
B & \rightarrow & B_1 \; 0 \; \{B.val = 2 \times B_1.val\} \\
 & \mid & B_1 \; 1 \; \{B.val = 2 \times B_1.val + 1\} \\
 & \mid & 1 \; \{B.val = 1\}
\end{array}
$$

Перепишите эту СУТ так, чтобы лежащая в ее основе грамматика перестала быть леворекурсивной, но вычисленное значение $B.val$ для входной строки оставалось тем же.

! **Упражнение 5.4.4.** Разработайте L-атрибутное СУО, аналогичное СУО из примера 5.10, для приведенных ниже продукций, каждая из которых представляет собой знакомую конструкцию управления потоком, имеющуюся в языке программирования С. Вам может потребоваться сгенерировать трехадресную команду для перехода к конкретной метке L — в этом случае генерируйте команду **goto** L.

а) $S \rightarrow \textbf{if} \; (C) \; S_1 \; \textbf{else} \; S_2$

б) $S \rightarrow \textbf{do} \; S_1 \; \textbf{while} \; (C)$

в) $S \rightarrow \text{'\{'} \; L \; \text{'\}'}; \; L \rightarrow L \; S \mid \epsilon$

Обратите внимание на то, что любая инструкция может содержать переход из средины к следующей инструкции, так что просто по порядку сгенерировать код для каждой инструкции недостаточно.

Упражнение 5.4.5. Преобразуйте каждое из ваших СУО из упражнения 5.4.4 в СУТ, как это было сделано в примере 5.19.

Упражнение 5.4.6. Модифицируйте СУО на рис. 5.25 так, чтобы оно включало синтезируемый атрибут $B.le$, длину бокса. Длина двух смежных боксов равна сумме их длин. Затем добавьте новые правила в соответствующие места СУТ на рис. 5.26.

Упражнение 5.4.7. Модифицируйте СУО на рис. 5.25 так, чтобы оно включало надстрочные индексы, с использованием оператора **sup** между боксами. Если бокс B_2 является надстрочным индексом бокса B_1, то позиция базовой линии B_2 находится на 0.6 кегля бокса B_1 выше базовой линии бокса B_1. Добавьте новую продукцию и правила в СУТ на рис. 5.26.

5.5 Реализация L-атрибутных СУО

Поскольку многие приложения трансляций используют L-атрибутные определения, в этом разделе мы рассмотрим их реализацию более подробно. Трансляция путем обхода дерева разбора осуществляется следующими методами.

1. *Построение дерева разбора и его аннотирование.* Этот метод работает для любого нециклического СУО. С аннотированными деревьями разбора мы уже знакомились в разделе 5.1.2.

2. *Построение дерева разбора, добавление действий и выполнение действий в прямом порядке обхода.* Этот подход работает для любого L-атрибутного определения. Преобразование L-атрибутного СУО в СУТ рассматривалось в разделе 5.4.5; в частности, в нем рассматривались вопросы вставки действий в продукции на основе семантических правил такого СУО.

В этом разделе мы рассмотрим другие методы трансляции в процессе синтаксического анализа.

3. *Использование синтаксического анализатора, работающего методом рекурсивного спуска*, с одной функцией для каждого нетерминала. Функция для нетерминала A получает наследуемые атрибуты A в качестве аргументов и возвращает синтезируемые атрибуты A.

4. *Генерация кода "на лету"* с применением синтаксического анализатора, работающего методом рекурсивного спуска.

5. *Реализация СУТ вместе с LL-синтаксическим анализатором.* Атрибуты хранятся в стеке синтаксического анализа, а правила выбирают требующиеся им атрибуты из известных позиций в стеке.

6. *Реализация СУТ вместе с LR-синтаксическим анализатором.* Этот метод может показаться неожиданным, поскольку СУТ для L-атрибутного СУО обычно содержит действия в средине продукций, и в процессе LR-синтаксического анализа мы не можем знать точно, какая именно продукция используется, пока не построим все ее тело. Однако, как мы увидим, если лежащая в основе грамматика принадлежит к классу LL, то всегда можно провести как синтаксический анализ, так и трансляцию в восходящем направлении.

5.5.1 Трансляция в процессе синтаксического анализа методом рекурсивного спуска

Синтаксический анализатор, работающий методом рекурсивного спуска, для каждого нетерминала A имеет отдельную функцию A (об этом говорилось в разделе 4.4.1). Можно расширить синтаксический анализатор и превратить его в транслятор, если следовать следующим правилам.

а) Аргументами функции A являются наследуемые атрибуты нетерминала A.

б) Возвращаемое функцией A значение представляет собой набор синтезируемых атрибутов нетерминала A.

В теле функции A требуется как выполнить синтаксический анализ, так и обработать атрибуты.

1. Принимается решение о том, какая продукция используется для развертывания A.
2. Когда это необходимо, проверяется каждый терминал входного потока. Будем считать, что возврат не требуется, но перейти к синтаксическому анализу методом рекурсивного спуска с возвратом можно путем восстановления позиции во входном потоке при ошибке, как описано в разделе 4.4.1.
3. В локальных переменных сохраняются значения всех атрибутов, необходимые для вычисления наследуемых атрибутов нетерминалов в теле продукции или синтезируемых атрибутов нетерминала заголовка.
4. Вызываются функции, соответствующие нетерминалам в теле выбранной продукции, с передачей им корректных аргументов. Поскольку лежащее в основе СУО является L-атрибутным, все необходимые для передачи атрибуты к этому моменту уже вычислены и сохранены в локальных переменных.

```
string S(label next) {
    string Scode, Ccode; /* Локальные переменные с фрагментами кода */
    label L1, L2; /* Локальные метки */
    if ( Текущий входной символ == токен while ) {
        Перемещение по входному потоку;
        Проверить наличие '(' во входной строке и перейти к новой
            позиции;
        L1 = new();
        L2 = new();
        Ccode = C(next, L2);
        Проверить наличие ')' во входной строке и перейти к новой
            позиции;
        Scode = S(L1);
        return("label"||L1||Ccode||"label"||L2||Scode);
    }
    else /* Инструкции других видов */
}
```

Рис. 5.29. Реализация конструкции while при помощи синтаксического анализатора, работающего методом рекурсивного спуска

Пример 5.20. Рассмотрим СУО и СУТ для конструкции while из примера 5.19. Набросок псевдокода значимой части функции S показан на рис. 5.29.

Здесь функция S показана как хранящая и возвращающая длинные строки. На практике было бы более эффективно, если бы функции наподобие S и C возвращали указатели на записи, представляющие эти строки. Тогда инструкция **return** в функции S не выполняла бы показанную конкатенацию строк, а вместо этого строила бы запись или, возможно, дерево записей, выражающее конкатенацию строк, представленных *Scode* и *Ccode*, метками $L1$ и $L2$, а также двух строк `"label"`. □

Пример 5.21. Теперь вернемся к СУТ на рис. 5.26 для текстовых боксов. Сначала мы обратимся к синтаксическому анализу, поскольку грамматика, лежащая в основе СУТ на рис. 5.26, неоднозначна. Приведенная далее преобразованная грамматика делает отношение смежности и индексирование правоассоциативными, при этом оператор **sub** имеет более высокий приоритет, чем смежность:

$$\begin{array}{lcl} S & \rightarrow & B \\ B & \rightarrow & T\ B_1 \mid T \\ T & \rightarrow & F\ \textbf{sub}\ T_1 \mid F \\ F & \rightarrow & (B) \mid \textbf{text} \end{array}$$

Два новых нетерминала, T и F, выполняют те же функции, что и слагаемые и сомножители в выражениях. Здесь "сомножитель", генерируемый F, представляет собой либо бокс в скобках, либо строку. "Слагаемое", генерируемое T, представляет собой "множитель" с последовательностью индексов, а бокс, генерируемый B, – последовательность смежных "слагаемых".

Атрибуты B переходят к T и F, поскольку эти новые нетерминалы также обозначают боксы; они введены исключительно во вспомогательных целях. Таким образом, и T, и F имеют наследуемый атрибут *ps* и синтезируемые атрибуты *ht* и *dp*, с семантическими действиями, которые могут быть получены из СУТ на рис. 5.26.

Грамматика пока что не готова для нисходящего синтаксического анализа, поскольку продукции для B и T имеют общие префиксы. Рассмотрим, например, T. Нисходящий синтаксический анализатор не может выбрать одну из двух продукций для T, просматривая только один символ из входного потока. К счастью, можно воспользоваться разновидностью левой факторизации, рассматривавшейся в разделе 4.3.4. В случае СУТ понятие общего префикса применимо также и к действиям. Обе продукции для T начинаются с нетерминала F, наследующего атрибут *ps* от T.

Псевдокод на рис. 5.30 для $T(ps)$ включает код $F(ps)$. После применения левой факторизации к $T \to F$ **sub** $T_1 \mid F$ остается только один вызов F; в псевдокоде показан результат подстановки кода F вместо этого вызова.

Функция T будет вызвана функцией B как $T(10.0)$ (этот вызов здесь не показан). Функция возвращает пару, состоящую из значений высоты и глубины бокса, генерируемого нетерминалом T; на практике она возвращает запись, содержащую высоту и глубину.

Функция T начинается с проверки наличия левой скобки – в этом случае используется продукция $F \to (B)$. Функция сохраняет значения, которые возвращает функция B, но если после B не следует правая скобка, то это означает, что мы столкнулись с синтаксической ошибкой, обработка которой здесь не показана.

В противном случае, если текущий входной символ – **text**, функция T использует функции *getHt* и *getDp* для определения высоты и глубины этого текста.

Затем T выясняет, является ли следующий бокс индексом, и если является, то соответствующим образом изменяет кегль. Для вычислений используются действия, связанные с продукцией $B \to B$ **sub** B на рис. 5.26 и вычисляющие высоту и глубину большого бокса. В противном случае функция просто возвращает значения, вычисленные функцией F: $(h1, d1)$. □

5.5.2 Генерация кода "на лету"

Построение длинных строк кода, являющихся значениями атрибутов, как это было сделано в примере 5.20, нежелательно по ряду причин, включая время,

```
(float, float) T(float ps) {
    float h1, h2, d1, d2; /* Локальные переменные для высоты и глубины */
    /* Начало кода F(ps) */
    if ( Текущий символ == '(' ) {
        Перемещение по входному потоку;
        (h1, d1) = B(ps);
        if (Текущий символ != ')' ) Синтаксическая ошибка: требуется ')';
        Перемещение по входному потоку;
    }
    else if ( Текущий символ == text ) {
        Обозначим значение text.lexval как t;
        Перемещение по входному потоку;
        h1 = getHt(ps, t);
        d1 = getDp(ps, t);
    }
    else Синтаксическая ошибка: требуется text или ')';
    /* Конец кода F(ps) */
    if ( Текущий символ == sub ) {
        Перемещение по входному потоку;
        (h2, d2) = T(0.7*ps);
        return (max(h1, h2 - 0.25 * ps), max(d1, d2 + 0.25 * ps));
    }
    return (h1, d1);
}
```

Рис. 5.30. Рекурсивный спуск для текстовых боксов

необходимое для копирования и перемещения длинных строк. В распространенных случаях, таких как наш пример с генерацией кода, вместо этого можно инкрементно генерировать части кода с записью в массив или выходной файл при помощи действий из СУТ. Для этого требуется, чтобы выполнялось следующее.

1. *Главный* (main) атрибут для одного или нескольких нетерминалов. Для удобства будем считать, что все главные атрибуты представляют собой строковые значения. В примере 5.20 таковыми были атрибуты *S.code* и *C.code*; прочие атрибуты не были главными.

2. Главные атрибуты являются синтезируемыми.

3. Правила для вычисления главных атрибутов гарантируют следующее.

   а) Главный атрибут представляет собой конкатенацию главных атрибутов нетерминалов, имеющихся в теле соответствующей продукции,

Типы главных атрибутов

Наше упрощающее предположение о том, что главные атрибуты представляют собой строки, слишком ограничительное. На самом деле типы главных атрибутов должны иметь значения, которые могут быть построены путем конкатенации элементов. Например, список объектов любого типа вполне годится в качестве главного атрибута, поскольку список может быть представлен таким образом, что к его концу можно эффективно добавлять новые элементы. Таким образом, если назначение главного атрибута состоит в представлении последовательности команд промежуточного кода, то этот код можно получить путем записи инструкций в конец массива объектов. Конечно, требования, перечисленные в разделе 5.5.2, применимы и к спискам; например, главные атрибуты должны собираться из других главных атрибутов путем конкатенации в соответствующем порядке.

возможно, с другими элементами, не являющимися главными атрибутами, такими как строки **label** или значения меток $L1$ и $L2$.

б) Главные атрибуты нетерминалов находятся в правиле в том же порядке, что и сами нетерминалы в теле продукции.

Как следствие приведенных условий главный атрибут может быть построен путем добавления в конкатенацию элементов, не являющихся главными атрибутами. Для инкрементной генерации значений главных атрибутов нетерминалов в теле продукции можно основываться на рекурсивных вызовах их функций.

Пример 5.22. Можно модифицировать функцию, приведенную на рис. 5.29, таким образом, чтобы она выводила элементы трансляции $S.code$ вместо возврата соответствующего значения. Такая измененная функция S показана на рис. 5.31.

На рис. 5.31 S и C не имеют возвращаемых значений, поскольку их синтезируемые атрибуты генерируются путем печати. Обратите внимание на важность положений инструкций *print*: сначала выводится `label` $L1$, затем — код для C (который представляет собой то же, что и значение *Ccode* на рис. 5.29), затем — `label` $L2$ и наконец — код из рекурсивного вызова для S (который представляет собой то же, что и значение *Scode* на рис. 5.29). Таким образом, код, выводимый этим вызовом S, в точности такой же, как и значение, возвращаемое на рис. 5.29.□

Кстати, можно внести те же изменения и в соответствующую СУТ — превратить построение главного атрибута в действия по выводу элементов этого атрибута. На рис. 5.32 показана СУТ из рис. 5.28, переделанная для генерации кода “на лету”.

```
void S(label next) {
    label L1, L2; /* Локальные метки */
    if ( Текущий символ == токен while ) {
        Перемещение по входному потоку;
        Проверить наличие '(' во входной строке и перейти к новой
            позиции;
        L1 = new();
        L2 = new();
        print("label", L1);
        C(next, L2);
        Проверить наличие ')' во входной строке и перейти к новой
            позиции;
        print("label", L2);
        S(L1);
    }
    else /* Инструкции других видов */
}
```

Рис. 5.31. Генерация кода для конструкции while синтаксическим анализатором методом рекурсивного спуска "на лету"

$$
\begin{array}{lll}
S \rightarrow & \textbf{while}\ (& \{\ L1 = new();\ L2 = new();\ C.false = S.next; \\
 & & \quad C.true = L2;\ print(\texttt{"label"}, L1);\ \} \\
 & C\ ) & \{\ S_1.next = L1;\ print(\texttt{"label"}, L2);\ \} \\
 & S_1 &
\end{array}
$$

Рис. 5.32. СУТ для генерации "на лету" кода конструкции while

5.5.3 L-атрибутные СУО и LL-синтаксический анализ

Предположим, что L-атрибутное СУО основано на LL-грамматике и что мы конвертировали его в СУТ с действиями, вставленными в продукции, как описано в разделе 5.4.5. После этого можно выполнить трансляцию в процессе LL-анализа путем расширения стека синтаксического анализатора таким образом, чтобы он хранил действия и некоторые данные, необходимые для вычисления атрибутов. Обычно данные представляют собой копии атрибутов.

В дополнение к записям, представляющим терминалы и нетерминалы, стек синтаксического анализатора хранит *записи действий* (action-records), представляющие выполняемые действия, и *записи синтеза* (synthesize-records) для хранения синтезируемых атрибутов нетерминалов. Управление атрибутами в стеке осуществляется в соответствии со следующими принципами.

- Наследуемые атрибуты нетерминала A помещаются в записи стека, представляющей данный нетерминал. Код для вычисления атрибутов обычно представляется записью действий непосредственно над записью стека для A; в действительности преобразование L-атрибутного СУО в СУТ гарантирует, что запись действия будет находиться непосредственно над A.

- Синтезируемые атрибуты нетерминала A размещаются в отдельной записи синтеза, которая находится в стеке непосредственно под записью для A.

Данная стратегия размещает записи различных типов в стеке синтаксического анализа, полагая, что такие вариантные типы записей корректно обрабатываются как подклассы класса "запись стека". На практике можно скомбинировать несколько записей в одну, но, пожалуй, пояснить идеи будет проще, если данные для различных целей будут храниться в различных записях.

Записи действий содержат указатели на выполняемый код. Действия могут также появляться и в записях синтеза; эти действия обычно размещают копии синтезируемых атрибутов в других записях ниже по стеку, где значения этих атрибутов потребуются после того, как запись синтеза и ее атрибуты будут сняты со стека.

Бегло взглянем на LL-синтаксический анализ, чтобы понять необходимость создания временных копий атрибутов. Из раздела 4.4.4 известно, что управляемый таблицей синтаксического анализа LL-анализатор имитирует левое порождение. Если w – входная строка, соответствие которой проверено до текущего момента, то в стеке хранится последовательность грамматических символов α, такая, что $S \underset{lm}{\overset{*}{\Rightarrow}} w\alpha$, где S – стартовый символ. Когда синтаксический анализатор выполняет раскрытие с использованием продукции $A \to B\ C$, он заменяет A на вершине стека на $B\ C$.

Предположим, что нетерминал C имеет наследуемый атрибут $C.i$. В силу продукции $A \to B\ C$ наследуемый атрибут $C.i$ может зависеть не только от наследуемых атрибутов A, но и от всех атрибутов B. Таким образом, перед тем, как вычислять атрибут $C.i$, следует полностью обработать B. Значит, все необходимые для вычисления $C.i$ атрибуты должны быть сохранены в записи действий, которые вычисляют $C.i$. В противном случае, когда синтаксический анализатор заменит A на вершине стека на $B\ C$, наследуемые атрибуты A просто исчезнут вместе с соответствующими записями в стеке.

Поскольку лежащее в основе СУО – L-атрибутное, это гарантирует, что значения наследуемых атрибутов A доступны, когда A оказывается на вершине стека. Следовательно, эти значения доступны в момент копирования в записи действий, которые вычисляют наследуемые атрибуты C. Кроме того, нет никаких проблем с памятью для синтезируемых атрибутов A, так как они находятся в записи син-

теза, которая при раскрытии с использованием продукции $A \to B\ C$ остается в стеке, ниже B и C.

При обработке B можно выполнить действия (с помощью записи, находящейся в стеке непосредственно над B), которые копируют его наследуемые атрибуты для использования нетерминалом C, а после того, как B обработан, запись синтеза для B может скопировать его синтезируемые атрибуты для использования при необходимости нетерминалом C. Аналогично синтезируемые атрибуты A могут потребовать для вычисления их значений временных переменных, которые могут быть скопированы в запись синтеза для A после обработки B, а затем C. Вот общий принцип, который обеспечивает работоспособность всех описанных копирований атрибутов.

- Все копирования выполняются между записями, которые создаются в процессе одного раскрытия одного нетерминала. Таким образом, каждая из этих записей знает, где именно ниже в стеке находится каждая другая запись, и может безопасно копировать в них необходимые значения.

Приведенный далее пример иллюстрирует реализацию наследуемых атрибутов в процессе LL-синтаксического анализа путем аккуратного копирования значений атрибутов. В приведенном примере возможны сокращения и оптимизации, в частности в случае правил копирования, состоящих в присваивании значения одного атрибута другому. Однако пока что мы отложим этот вопрос до примера 5.24, в котором иллюстрируются записи синтеза.

Пример 5.23. В этом примере реализуется представленная на рис. 5.32 СУТ, "на лету" генерирующая код для конструкции while. В этой СУТ нет синтезируемых атрибутов, не считая фиктивных атрибутов, представляющих метки.

На рис. 5.33, *а* показана ситуация, в которой для раскрытия S используется продукция while, поскольку очередным символом во входном потоке оказался символ **while**. Запись на вершине стека — это запись для S, которая содержит только наследуемый атрибут *S.next*, который, будем считать, имеет значение x. Поскольку здесь выполняется нисходящий синтаксический анализ, вершина стека в соответствии с ранее принятыми соглашениями показана на рисунке слева.

На рис. 5.33, *б* показана ситуация непосредственно после раскрытия S. Перед нетерминалами C и S_1 имеются записи действий, соответствующие действиям лежащей в основе СУТ из рис. 5.32. Запись для C содержит место для наследуемых атрибутов *true* и *false*, а запись для S_1, как и все записи для S, — место для атрибута *next*. На рисунке значения этих полей показаны как ?, поскольку пока что значения указанных атрибутов неизвестны.

Синтаксический анализатор распознает во входном потоке символы **while** и (и снимает их записи со стека. Теперь первая запись действий находится на вершине стека и должна быть выполнена. Эта запись содержит поле *snext*, предназначенное для хранения копии наследуемого атрибута *S.next*. Когда S снимается

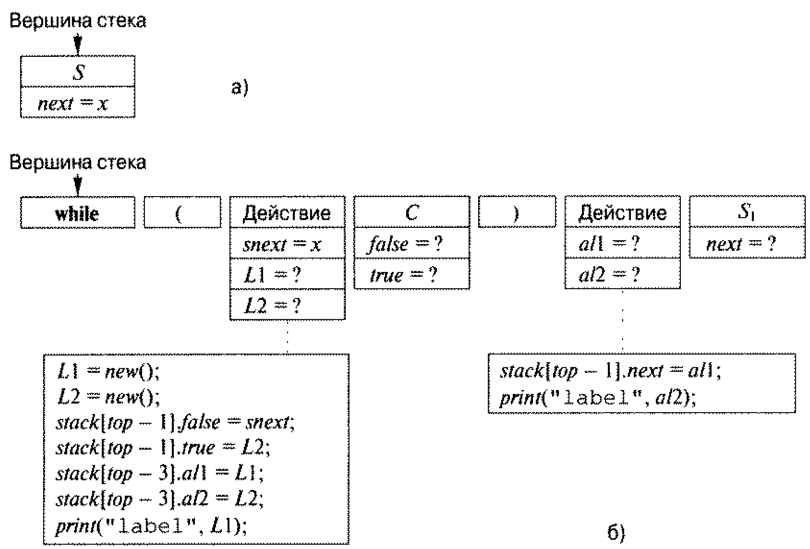


Рис. 5.33. Раскрытие S в соответствии с продукцией для конструкции while

со стека, значение $S.next$ копируется в поле *snext* для использования в процессе вычисления наследуемых атрибутов C. Код первого действия генерирует новые значения для $L1$ и $L2$, которые мы полагаем равными соответственно y и z. Следующий шаг делает z значением $C.true$. Присваивание $stack\,[top - 1]\,.true = L2$ записано с учетом знания о том, что оно будет выполняться только в тот момент, когда данная запись действий будет находиться на вершине стека, так что $top - 1$ указывает на запись непосредственно под ней, т.е. запись для C.

Затем первая запись действий копирует $L1$ в поле $al1$ второго действия, где это значение будет использовано при вычислении $S_1.next$. Она также копирует $L2$ в поле $al2$ второго действия; это значение необходимо для корректного вывода, выполняемого этим действием. Наконец, первая запись действий выводит `label` y.

Ситуация после завершения первого действия и снятия со стека его записи показана на рис. 5.34. Значения наследуемых атрибутов в записи C корректно установлены, как и значения временных переменных $al1$ и $al2$ во второй записи действий. В этот момент раскрывается C и мы полагаем, что генерируется код, реализующий этот тест, который содержит переходы к меткам x и z там, где это требуется. После того как запись для C снимается со стека, на его вершине оказывается запись для), что заставляет синтаксический анализатор проверять наличие) во входном потоке.

Когда выполняется действие из записи, находящейся в стеке над записью для S_1, его код устанавливает $S_1.next$ и выводит `label` z. После этого на вершине

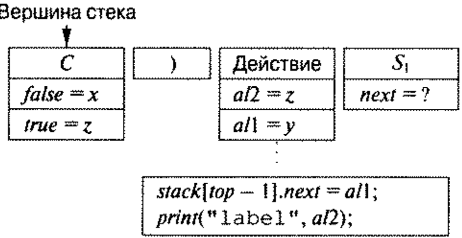


Рис. 5.34. Стек после завершения действия, находившегося в стеке над C

стека оказывается запись для нетерминала S_1, при раскрытии которого, как мы полагаем, выполняется корректная генерация кода, реализующего скрывающиеся за этим нетерминалом инструкции любых типов и переходящего к метке y. □

Пример 5.24. Рассмотрим ту же конструкцию while, но теперь трансляция генерирует код не "на лету", а как синтезируемый атрибут *S.code*. Чтобы лучше понять пояснения, приводимые в данном примере, желательно все время помнить следующий инвариант, который, как мы полагаем, выполняется для каждого нетерминала.

- Каждый нетерминал, который имеет связанный с ним код, оставляет его в виде строки в записи синтеза, находящейся в стеке непосредственно под ним.

Полагая данное утверждение истинным, мы будем работать с while-продукцией так, чтобы поддерживать это утверждение как инвариант.

На рис. 5.35, *а* показана ситуация непосредственно перед развертыванием S с использованием продукции для конструкции while. На вершине стека находится запись для S; она содержит поле для своего наследуемого атрибута *S.next*, как в примере 5.23. Непосредственно под этой записью располагается запись синтеза для данного экземпляра S. В ней имеется поле для *S.code*, как и у всех записей синтеза для S. Показаны также некоторые другие поля для локальных данных и действий, поскольку СУТ для продукции while на рис. 5.28, конечно же, является частью большей СУТ.

Наше раскрытие S основано на СУТ, представленной на рис. 5.28, и показано на рис. 5.35, *б*. Для ускорения в процессе раскрытия мы полагаем, что наследуемый атрибут *S.next* присваивается непосредственно *C.false*, а не размещается в первом действии и затем копируется в запись для C.

Рассмотрим, что делает каждая запись, оказавшись на вершине стека. Сначала запись для **while** заставляет выполнить проверку соответствия токена **while** входному символу (соответствие должно выполняться, иначе для раскрытия S будет использоваться другая продукция). После того как со стека будут сняты **while** и (,

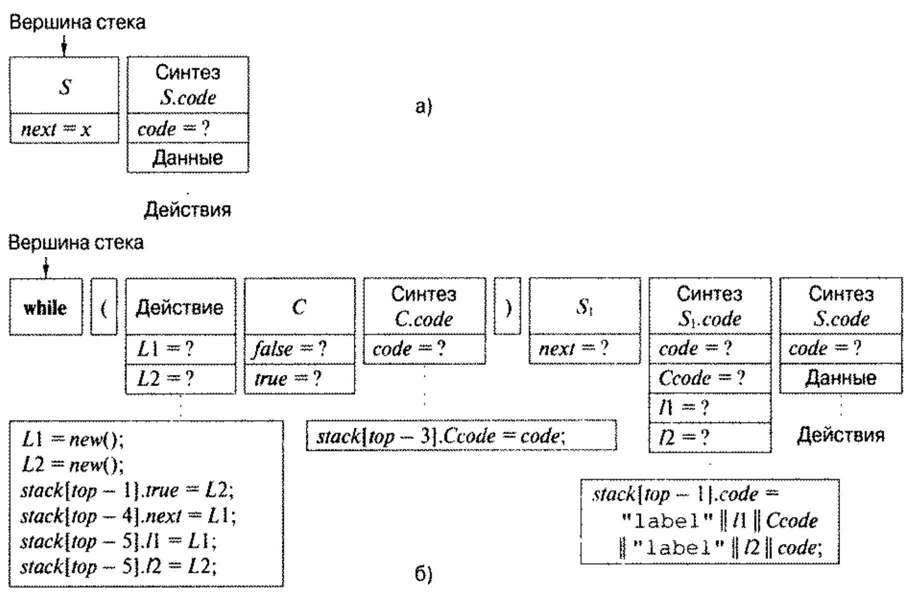


Рис. 5.35. Раскрытие S с синтезируемым атрибутом, конструируемым в стеке

будет выполнен код записи действий. Он генерирует значения $L1$ и $L2$, и мы можем сократить вычисления, копируя их непосредственно в наследуемые атрибуты $S_1.next$ $C.true$. Два последних шага действия копируют значения $L1$ и $L2$ в запись "Синтез $S_1.code$".

Запись синтеза для S_1 выполняет двойную функцию: она не только хранит синтезируемый атрибут $S_1.code$, но и служит в качестве записи действий для завершения вычислений атрибутов всей продукции $S \rightarrow$ **while** (C) S_1. В частности, когда эта запись оказывается на вершине стека, она вычисляет синтезируемый атрибут $S.code$ и помещает его значение в запись синтеза для заголовка S.

Когда C находится на вершине стека, оба его наследуемых атрибута оказываются вычисленными. В соответствии со сформулированным ранее инвариантом мы предполагаем, что при этом корректно сгенерирован код для вычисления условия с переходом к соответствующей метке. Мы также считаем, что действие, выполняемое при раскрытии C, корректно размещает этот код в записи ниже в стеке в качестве значения синтезируемого атрибута $C.code$.

После снятия C со стека на его вершине оказывается запись синтеза для $C.code$. Этот код необходим в записи синтеза для $S_1.code$, поскольку именно в ней выполняется конкатенация всех элементов кода, образующих $S.code$. Таким образом, запись синтеза для $C.code$ содержит действие, состоящее в копировании $C.code$ в запись синтеза для $S_1.code$. После этого вершины стека достигает запись для токена), что приводит к выполнению проверки наличия) во входном потоке.

В предположении, что проверка выполнена успешно, на вершину стека поднимается запись для S_1. Согласно инварианту этот нетерминал раскрыт, и в результате соответствующий код корректно генерируется и размещается в поле *code* в записи синтеза для S_1.

Теперь все поля данных записи синтеза для S_1 заполнены, так что, когда оно оказывается на вершине стека, выполнению его действий ничто не препятствует. Действие состоит в конкатенации меток и кода из $C.code$ и $S_1.code$ в правильном порядке. Получающаяся в результате строка размещается ниже, т.е. в записи синтеза для S. Теперь у нас есть корректно вычисленный атрибут $S.code$, и, когда запись синтеза для S окажется на вершине стека, этот код будет доступен для размещения в другой записи ниже в стеке, где в конечном счете он будет собран в большей строке кода, реализующей элемент программы, частью которого является S. □

5.5.4 Восходящий синтаксический анализ L-атрибутных СУО

Любая трансляция, которая может быть выполнена в нисходящем направлении, может быть выполнена и при восходящем синтаксическом анализе. Точнее, для данных L-атрибутного СУО и LL-грамматики можно адаптировать грамматику для вычисления того же самого СУО над новой грамматикой в процессе LR-синтаксического анализа. Этот "трюк" состоит из трех частей.

1. Начнем с СУТ, построенной, как в разделе 5.4.5, которая размещает перед каждым нетерминалом действия по вычислению его наследуемых атрибутов, а в конце продукции — действия по вычислению синтезируемых атрибутов.

2. Введем в грамматику нетерминалы-маркеры на месте каждого вставленного действия. В каждое такое место вставляется свой маркер; для каждого маркера M существует только одна продукция, а именно — $M \to \epsilon$.

3. Модифицируем действие a, заменяемое маркером M в некоторой продукции $A \to \alpha\ \{a\}\ \beta$, и связываем с продукцией $M \to \epsilon$ действие a', которое

   а) копирует в качестве наследуемых атрибутов M любые атрибуты A или символов из α, которые требуются действию a;

   б) вычисляет атрибуты таким же способом, что и a, но делает эти атрибуты синтезируемыми атрибутами M.

   Это изменение выглядит некорректным, поскольку получается, что действие, связанное с продукцией $M \to \epsilon$, должно иметь доступ к атрибутам,

Возможна ли обработка L-атрибутного СУО на основе LR-грамматики

В разделе 5.4.1 мы видели, что каждое S-атрибутное СУО на основе LR-грамматики может быть реализовано в процессе восходящего синтаксического анализа. Из раздела 5.5.3 известно, что каждое L-атрибутное СУО на основе LL-грамматики может быть проанализировано при помощи нисходящего синтаксического анализа. Поскольку LL-грамматики являются истинным подмножеством LR-грамматик, а S-атрибутные СУО — истинным подмножеством L-атрибутных СУО, можно ли работать с любой LR-грамматикой и L-атрибутным СУО в восходящем направлении?

Нет, как показывают следующие интуитивно понятные доводы. Предположим, имеются продукция $A \to B\ C$ из LR-грамматики и наследуемый атрибут $B.i$, который зависит от наследуемых атрибутов A. При выполнении свертки к B мы все еще не видели входного потока, генерируемого C, так что мы не можем быть уверены в том, что имеем дело с телом продукции $A \to B\ C$. Таким образом, мы все еще не можем вычислить значение $B.i$, поскольку нет гарантии, что следует использовать правило, связанное именно с данной продукцией.

Возможно, решение состоит в том, чтобы подождать свертки к C и убедиться в том, что мы должны свернуть $B\ C$ к A. Однако даже тогда мы не знаем наследуемых атрибутов A, поскольку даже после свертки может быть неизвестно, какое именно тело продукции содержит это A. Если опять воспользоваться откладыванием вычисления $B.i$, то в конечном итоге мы придем к тому, что мы не можем принять ни одного решения, пока не будет проанализирована вся входная строка. По сути, это означает применение стратегии "сначала построить дерево разбора, а затем выполнить трансляцию".

связанным с грамматическими символами, которые отсутствуют в данной продукции. Однако, так как мы реализуем действия с использованием стека LR-синтаксического анализа, необходимые атрибуты всегда будут доступны посредством известных позиций записей ниже в стеке.

Пример 5.25. Предположим, имеется LL-грамматика с продукцией $A \to B\ C$ и наследуемый атрибут $B.i$ вычисляется с использованием наследуемого атрибута $A.i$ по формуле $B.i = f(A.i)$. Значит, интересующий нас фрагмент СУТ имеет вид

$$A \to \{B.i = f(A.i);\}\ B\ C$$

Почему работают маркеры

Маркеры — это нетерминалы, порождающие только ϵ и встречающиеся только один раз в телах всех продукций. Мы не приводим здесь формального доказательства того, что нетерминалы-маркеры могут быть добавлены в любые позиции в телах продукций LL-грамматики и получившаяся в результате грамматика останется LR-грамматикой. Интуитивно это поясняется следующим образом. Если грамматика принадлежит классу LL, то можно определить, что строка w во входном потоке порождается нетерминалом A, в порождении, начинающемся с продукции $A \to \alpha$, просматривая только один первый символ w (или следующий символ, если $w = \epsilon$). Таким образом, при восходящем синтаксическом анализе w тот факт, что префикс w должен быть свернут в α, а затем в S, становится известен, как только во входном потоке появляется начало строки w. В частности, если мы вставим маркер где угодно в α, то LR-состояния включат тот факт, что в некотором месте должен иметься данный маркер, и выполнят свертку ϵ в маркер в соответствующей точке входного потока.

Введем маркер M с наследуемым атрибутом $M.i$ и синтезируемым атрибутом $M.s$. Первый является копией $A.i$, а последний — $B.i$. СУТ записывается как

$$A \to M\ B\ C$$
$$M \to \{M.i = A.i;\ M.s = f\,(M.i)\,;\}$$

Заметим, что атрибут $A.i$ не доступен правилу для M, но в действительности можно так расположить наследуемые атрибуты нетерминалов, таких как A, чтобы они находились в стеке непосредственно под тем местом, где позже будет выполнена свертка в A. Таким образом, при свертке ϵ в M $A.i$ находится непосредственно под ним, откуда и может быть считано. Значение $M.s$, находящееся в стеке слева вместе с M, в действительности представляет собой $B.i$ и находится ниже, сразу за тем местом, где позже будет выполнена свертка в B. □

Пример 5.26. Давайте превратим СУТ на рис. 5.28 в СУТ, которая может использоваться при LR-синтаксическом анализе переделанной грамматики. Введем маркер M перед C и маркер N перед S_1, так что лежащая в основе СУТ грамматика принимает вид

$$S \to \textbf{while}\ (M\ C)\ N\ S_1$$
$$M \to \epsilon$$
$$N \to \epsilon$$

Перед тем как рассмотреть действия, связанные с маркерами M и N, сформулируем "гипотезу индукции" о том, где хранятся атрибуты.

1. Ниже всего тела while-продукции — т.е. ниже **while** в стеке — находится наследуемый атрибут *S.next*. Мы можем не знать, какой нетерминал или состояние синтаксического анализатора связано с этой записью в стеке, но мы можем быть уверены, что в фиксированной позиции этой записи имеется поле, хранящее *S.next* до того, как мы начнем распознавать, что же именно порождено этим S.

2. Наследуемые атрибуты *C.true* и *C.false* находятся в стеке в записи, расположенной непосредственно под записью для C. Поскольку предполагается, что грамматика принадлежит классу LL, наличие **while** во входном потоке гарантирует, что может быть распознана единственная продукция, а именно — while-продукция, так что можно гарантировать, что M будет находиться в стеке непосредственно под C, а запись M будет содержать наследуемые атрибуты C.

3. Аналогично наследуемый атрибут S_1*.next* должен находиться в стеке непосредственно под S_1, так что можно разместить этот атрибут в записи для N.

4. Синтезируемый атрибут *C.code* будет находиться в записи для C. Как всегда, когда в качестве значения атрибута выступает длинная строка, на практике в записи хранится указатель на (объект, представляющий) строку, в то время как сама строка располагается вне стека.

5. Аналогично синтезируемый атрибут S_1*.next* находится в записи для S_1.

Проследим теперь за процессом синтаксического анализа конструкции while. Предположим, что запись, в которой хранится *S.next*, находится на вершине стека, а очередной входной символ — терминал **while**. Перенесем этот терминал в стек. Теперь определенно известно, что распознаваемая продукция представляет собой while-продукцию, так что LR-синтаксический анализатор может перенести (и определить, что следующим шагом должна быть свертка ϵ в M. Состояние стека в этот момент показано на рис. 5.36. На этом рисунке приведены также действия, связанные со сверткой в M. Здесь создаются значения $L1$ и $L2$, которые хранятся в полях M-записи. В этой же записи располагаются поля для *C.true* и *C.false*. Эти атрибуты должны находиться во втором и третьем полях записи для согласованности с другими записями стека, которые могут находиться ниже C в других контекстах и также предоставлять эти атрибуты C. Действие завершается присваиванием значений атрибутам *C.true* и *C.false*, одному — только что сгенерированного значения $L2$, а другому — значения из записи, располагающейся ниже в стеке, в которой, как мы знаем, может быть найдено значение *S.next*.

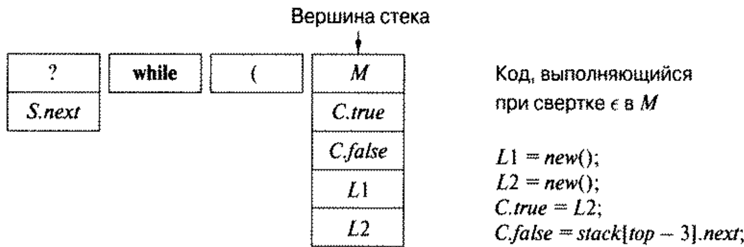


Рис. 5.36. Стек LR-синтаксического анализа после свертки ϵ в M

Мы предполагаем, что все очередные символы входного потока корректно сворачиваются в C. Следовательно, синтезированный атрибут $C.code$ размещается в записи для C. Это изменение в стеке показано на рис. 5.37; на нем показаны также несколько записей, которые позже оказываются в стеке над записью C.

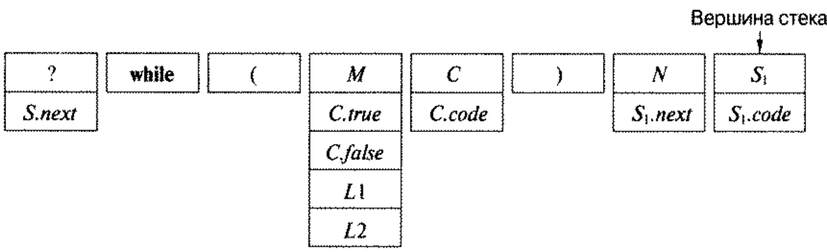


Рис. 5.37. Стек непосредственно перед сверткой тела while-продукции в S

Продолжая распознавание конструкции while, синтаксический анализатор должен обнаружить во входном потоке символ), который он поместит в стек в соответствующую запись. В этот момент синтаксический анализатор, в силу принадлежности грамматики к классу LL знающий, что он работает с конструкцией while, выполнит свертку ϵ в N. Единственные данные, связанные с N, — это наследуемый атрибут $S_1.next$. Заметим, что этот атрибут должен находиться в записи для N, потому что она находится непосредственно под записью для S_1. Выполняемый при свертке код вычисляет значение $S_1.next$ следующим образом:

$$S_1.next = stack\,[top - 3]\,.L1;$$

Это действие обращается к третьей записи под N (которая находится в момент выполнения кода на вершине стека) и получает значение $L1$.

Затем синтаксический анализатор выполняет свертку некоторого префикса остающегося входного потока в нетерминал S (который мы везде указываем с индексом, как S_1, чтобы отличать его от нетерминала S в заголовке продукции). Вычисленное при этом значение $S_1.code$ находится в записи стека для S_1. Этот шаг приводит нас в состояние, показанное на рис. 5.37.

В этот момент синтаксический анализатор выполняет свертку всей части стека от **while** до S_1 в S. Вот код, который выполняется при этой свертке:

$$\begin{aligned}&tempCode = \textbf{label} \parallel stack[top-4].L1 \parallel stack[top-3].code \parallel\\&\quad \textbf{label} \parallel stack[top-4].L2 \parallel stack[top].code;\\&top = top - 5;\\&stack[top].code = tempCode;\end{aligned}$$

Иначе говоря, мы собираем значение $S.code$ в переменной *tempCode*. Это обычный код, состоящий из меток $L1$ и $L2$, кода C и кода S_1. Затем выполняется снятие со стека, так что S оказывается в стеке на том месте, где находился терминал **while**. Код S размещается в поле *code* указанной записи, где он может рассматриваться как синтезируемый атрибут $S.code$. Заметим, что в процессе рассмотрения нашего примера мы нигде не показывали работу с LR-состояниями, которые также должны находиться в стеке в поле, в которое мы вносим грамматические символы. □

5.5.5 Упражнения к разделу 5.5

Упражнение 5.5.1. Реализуйте каждое из ваших СУО из упражнения 5.4.4 в виде синтаксического анализатора, работающего методом рекурсивного спуска, как это было сделано в разделе 5.5.1.

Упражнение 5.5.2. Реализуйте каждое из ваших СУО из упражнения 5.4.4 в виде синтаксического анализатора, работающего методом рекурсивного спуска, как это было сделано в разделе 5.5.2.

Упражнение 5.5.3. Реализуйте каждое из ваших СУО из упражнения 5.4.4 при помощи LL-синтаксического анализатора, как это было сделано в разделе 5.5.3, с генерацией кода "на лету".

Упражнение 5.5.4. Реализуйте каждое из ваших СУО из упражнения 5.4.4 при помощи LL-синтаксического анализатора, как это было сделано в разделе 5.5.3, но в этом случае код (или указатель на код) хранится в стеке.

Упражнение 5.5.5. Реализуйте каждое из ваших СУО из упражнения 5.4.4 при помощи LR-синтаксического анализатора, как это было сделано в разделе 5.5.4.

Упражнение 5.5.6. Реализуйте ваше СУО из упражнения 5.2.4 так, как это было сделано в разделе 5.5.1. Будет ли чем-то отличаться реализация в стиле раздела 5.5.2?

5.6 Резюме к главе 5

✦ *Наследуемые и синтезируемые атрибуты*. Синтаксически управляемые определения могут использовать два типа атрибутов. Синтезируемые атрибуты в узле дерева разбора вычисляются с использованием атрибутов

в дочерних узлах. Наследуемый атрибут в узле вычисляется с использованием атрибутов в родительском узле и/или в узлах-братьях.

- *Графы зависимостей.* Для данного дерева разбора и СУО мы проводим дуги между экземплярами атрибутов, связанными с каждым узлом дерева разбора, для указания того, что атрибут, на который указывает стрелка дуги, вычисляется с использованием значения атрибута, из которого выходит данная дуга.

- *Циклические определения.* В проблематичных СУО можно найти деревья разбора, для которых не существует порядка, в котором можно вычислить все атрибуты всех узлов. Такие деревья разбора содержат циклы в связанных с ними графах зависимостей. Выяснение, имеет ли СУО такие циклические графы зависимостей, — очень сложная задача.

- *S-атрибутные определения.* В S-атрибутном СУО все атрибуты синтезируемые.

- *L-атрибутные определения.* В L-атрибутном СУО атрибуты могут быть наследуемыми или синтезируемыми. Однако наследуемые атрибуты в узле дерева разбора могут зависеть только от наследуемых атрибутов в родительском узле и от любых атрибутов в братских узлах, находящихся слева от рассматриваемого.

- *Синтаксические деревья.* Каждый узел синтаксического дерева представляет конструкцию; дочерние узлы представляют значащие компоненты конструкции.

- *Реализация S-атрибутных СУО.* S-атрибутное СУО может быть реализовано при помощи СУТ, в которой все действия находятся в конце продукций (постфиксная СУТ). Действия вычисляют синтезируемые атрибуты заголовков продукций с использованием синтезируемых атрибутов символов их тел. Если лежащая в основе грамматика принадлежит классу LR, то такая СУТ может быть реализована в стеке LR-синтаксического анализатора.

- *Устранение левой рекурсии из СУТ.* Если СУТ содержит только побочные действия (без вычисления атрибутов), то применим стандартный алгоритм устранения левой рекурсии из грамматики, при использовании которого действия рассматриваются так, как если бы это были терминалы. При вычислении атрибутов устранение левой рекурсии возможно, если СУТ является постфиксной.

- *Реализация L-атрибутного СУО в процессе синтаксического анализа методом рекурсивного спуска.* Если имеется L-атрибутное определение на основе грамматики, к которой применим нисходящий синтаксический анализ, то для реализации трансляции можно построить синтаксический анализатор, работающий методом рекурсивного спуска без возврата. Наследуемые атрибуты становятся аргументами функций для их нетерминалов, а синтезируемые атрибуты этими функциями возвращаются.

- *Реализация L-атрибутного СУО на основе LL-грамматики.* Каждое L-атрибутное определение с лежащей в его основе LL-грамматикой может быть реализовано одновременно с синтаксическим анализом. Записи для хранения синтезируемых атрибутов нетерминалов размещаются в стеке под записями нетерминалов, в то время как наследуемые атрибуты нетерминалов хранятся в стеке вместе с нетерминалами. Записи действий также размещаются в стеке для выполнения вычислений атрибутов в соответствующие моменты времени.

- *Реализация L-атрибутного СУО на основе LL-грамматики при восходящем синтаксическом анализе.* L-атрибутное определение с лежащей в его основе LL-грамматикой может быть преобразовано в трансляцию на основе LR-грамматики и трансляцию, выполняемую в процессе восходящего синтаксического анализа. Преобразование грамматики включает нетерминалы-маркеры, которые появляются в стеке восходящего синтаксического анализатора и хранят наследуемые атрибуты нетерминалов, находящихся в стеке над ними. Синтезируемые атрибуты хранятся в стеке вместе с их нетерминалами.

5.7 Список литературы к главе 5

Синтаксически управляемые определения представляют собой вид индуктивных определений, в которых индукция проявляется в синтаксической структуре. Как таковые они давно неформально используются в математике. Их применение к языкам программирования началось с Algol 60.

Идея синтаксического анализатора, который вызывает семантические действия, может быть найдена у Сеймельсона (Samelson) и Бауэра (Bauer) [8], а также у Брукера (Brooker) и Морриса (Morris) [1]. Айронс (Irons) [2] создал один из первых синтаксически управляемых компиляторов с использованием синтезируемых атрибутов. Класс L-атрибутных определений введен в [6].

Наследуемые атрибуты, графы зависимостей и проверка цикличности СУО (т.е. выяснение, существует ли некоторое дерево разбора, для которого не существует порядка вычисления атрибутов) обязаны своим происхождением Кнуту

(Knuth) [5]. Джазаери (Jazayeri), Огден (Ogden) и Раундс (Rounds) [3] показали, что проверка цикличности требует времени, экспоненциально зависящего от размера СУО.

Генераторы синтаксических анализаторов, такие как `Yacc` [4] (см. также список литературы к главе 4), поддерживают вычисление атрибутов в процессе синтаксического анализа.

Обзор Паакки (Paakki) [7] может служить хорошей отправной точкой для литературного поиска по синтаксически управляемым определениям и трансляциям.

1. Brooker, R. A. and D. Morris, "A general translation program for phrase structure languages", *J. ACM* **9**:1 (1962), pp. 1–10.

2. Irons, E. T., "A syntax directed compiler for Algol 60", *Comm. ACM* **4**:1 (1961), pp. 51–55.

3. Jazayeri, M., W. F. Ogden, and W. C. Rounds, "The intrinsic exponential complexity of the circularity problem for attribute grammars", *Comm. ACM* **18**:12 (1975), pp. 697–706.

4. Johnson, S. C., "Yacc — Yet Another Compiler Compiler", Computing Science Technical Report 32, Bell Laboratories, Murray Hill, NJ, 1975. Доступно по адресу `http://dinosaur.compilertools.net/yacc/`.

5. Knuth, D. E., "Semantics of context-free languages", *Mathematical Systems Theory* **2**:2 (1968), pp. 127–145. См. также *Mathematical Systems Theory* **5**:1 (1971), pp. 95–96.

6. Lewis, P. M. II, D. J. Rosenkrantz, and R. E. Stearns, "Attributed translations", *J. Computer and System Sciences* **9**:3 (1974), pp. 279–307.

7. Paakki, J., "Attribute grammar paradigms — a high-level methodology in language implementation", *Computing Surveys* **27**:2 (1995), pp. 196–255.

8. Samelson, K. and F. L. Bauer, "Sequential formula translation", *Comm. ACM* **3**:2 (1960), pp. 76–83.

Глава 6

Генерация промежуточного кода

В модели анализа-синтеза компилятора на начальной стадии анализируется исходная программа и создается промежуточное представление, из которого на заключительной стадии генерируется целевой код. В идеальном случае детали исходного языка ограничены начальной стадией, а детали целевой машины — заключительной стадией компилятора. При наличии соответствующим образом определенного промежуточного представления компилятор для языка i и целевой машины j может быть построен путем комбинации начальной стадии для языка i и заключительной — для машины j. Этот подход к разработке набора компиляторов может сэкономить значительное количество работы: $m \times n$ компиляторов могут быть созданы путем написания m начальных стадий и n заключительных.

В этой главе будут рассмотрены вопросы промежуточного представления, статической проверки типов и генерации промежуточного кода. Для простоты мы полагаем, что начальная стадия компилятора организована так, как показано на рис. 6.1, где синтаксический анализ, статические проверки и генерация промежуточного кода выполняются последовательно; однако иногда они могут быть скомбинированы и включены в синтаксический анализ. Для определения проверок и трансляции мы воспользуемся синтаксически управляемым формализмом из глав 2 и 5. Многие схемы трансляции могут быть реализованы в процессе восходящего или нисходящего синтаксического анализа с применением методов из главы 5. Все схемы могут быть реализованы путем построения синтаксического дерева с его последующих обходом.

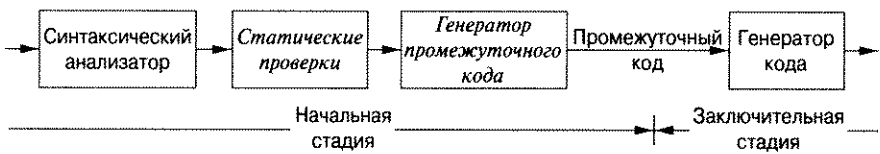


Рис. 6.1. Логическая структура начальной стадии компилятора

Статические проверки включают *проверку типов*, которая гарантирует применение операторов к совместимым операндам. Они также включают любые синтаксические проверки, оставшиеся после синтаксического анализа. Например, статическая проверка проверяет, что инструкция break в языке C находится

в охватывающей конструкции while, for или switch; если такой охватывающей конструкции нет, генерируется сообщение об ошибке.

Подход, описанный в этой главе, может использоваться для широкого диапазона промежуточных представлений, включая синтаксические деревья и трехадресный код, описанные в разделе 2.8. Термин "трехадресный код" происходит от команд общего вида $x = y \ op \ z$ с тремя адресами: два — операндов y и z и один — результата x.

В процессе трансляции программы на некотором исходном языке в код для заданной целевой машины компилятор может построить последовательность промежуточных представлений, как показано на рис. 6.2. Высокоуровневые представления близки к исходному языку, а низкоуровневые — к целевому коду. Синтаксические деревья представляют собой высокий уровень; они описывают естественную иерархическую структуру исходной программы и хорошо подходят для задач наподобие статической проверки типов.

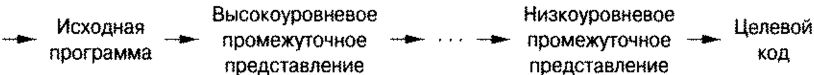


Рис. 6.2. Компилятор может использовать последовательность промежуточных представлений

Низкоуровневое представление подходит для машинно-зависимых задач, таких как распределение регистров и выбор команд. Трехадресный код может варьироваться в широких пределах — от высокоуровневого до низкоуровневого, в зависимости от выбора операторов. Как мы увидим в разделе 6.2.3, для выражений различие между синтаксическими деревьями и трехадресным кодом невелико. Но для конструкций циклов, например, синтаксическое дерево представляет компоненты конструкции, в то время как трехадресный код для представления потока управления использует метки и команды перехода — так же, как и машинный язык.

Выбор или дизайн промежуточного представления варьируется от компилятора к компилятору. Промежуточное представление может как использовать реальный язык, так и состоять из внутренних структур данных, совместно используемых фазами компилятора. C представляет собой язык программирования, но он очень часто используется в качестве промежуточного в силу его гибкости, возможности компиляции в эффективный машинный код и распространенности компиляторов. Например, первоначально компилятор C++ состоял из начальной стадии, которая генерировала код C; при этом компилятор C можно рассматривать как заключительную стадию.

6.1 Варианты синтаксических деревьев

Узлы синтаксического дерева представляют конструкции в исходной программе; их дочерние узлы представляют значимые компоненты конструкций. Ориентированный ациклический граф (directed acyclic graph) для выражения идентифицирует *общие подвыражения* (common subexpressions), т.е. подвыражения, встречающиеся в выражении более одного раза. Как мы увидим в этом разделе далее, ориентированный ациклический граф может быть построен с помощью тех же методов, которые использовались при построении синтаксических деревьев.

6.1.1 Ориентированные ациклические графы для выражений

Подобно синтаксическому дереву для выражения ориентированный ациклический граф имеет листья, соответствующие атомарным операндам, и внутренние узлы, соответствующие операторам. Отличие состоит в том, что узел N в ориентированном ациклическом графе имеет более одного родителя, если N представляет собой общее подвыражение; в синтаксическом дереве поддерево для общего подвыражения будет продублировано столько раз, сколько раз это подвыражение встречается в исходном выражении. Таким образом, ориентированный ациклический граф не только представляет выражение более кратко, но и дает генератору кода возможность генерации более эффективного кода для вычисления выражения.

Пример 6.1. На рис. 6.3 показан ориентированный ациклический граф для выражения `a+a*(b-c)+(b-c)*d`

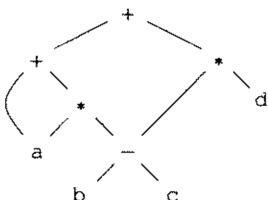


Рис. 6.3. Ориентированный ациклический граф для выражения `a+a*(b-c)+(b-c)*d`

Лист, представляющий `a`, имеет два родителя, поскольку `a` встречается в выражении дважды. Еще более интересно то, что два общих подвыражения `b-c` представлены одним узлом, помеченным −. Этот узел имеет два родительских узла, представляющих два использования упомянутого подвыражения в подвыра-

жениях `a*(b-c)` и `(b-c)*d`. Несмотря на то что `b` и `c` встречаются в полном выражении дважды, их узлы имеют один родительский узел, поскольку оба раза они встречаются в общем подвыражении `b-c`. □

СУО на рис. 6.4 может строить как синтаксические деревья, так и ориентированные ациклические графы. Оно использовалось для построения синтаксических деревьев в примере 5.11, в котором функции *Leaf* и *Node* создают новые узлы при каждом вызове. Те же функции позволяют построить ориентированный ациклический граф, если перед созданием нового узла эти функции будут сначала проверять, не существует ли уже идентичный узел, и если существует, то не создавать новый узел, а возвращать существующий. Например, перед созданием нового узла *Node* (*op*, *left*, *right*) мы проверяем, имеется ли узел с меткой *op* и дочерними узлами *left* и *right* (в указанном порядке). Если имеется, то функция *Node* возвратит существующий узел; в противном случае она создаст новый узел.

	Продукция	Семантическое правило
1)	$E \to E_1 + T$	$E.node = \textbf{new}\ Node\,('+', E_1.node, T.node)$
2)	$E \to E_1 - T$	$E.node = \textbf{new}\ Node\,('-', E_1.node, T.node)$
3)	$E \to T$	$E.node = T.node$
4)	$T \to (E)$	$T.node = E.node$
5)	$T \to \textbf{id}$	$T.node = \textbf{new}\ Leaf(\textbf{id}, \textbf{id}.entry)$
6)	$T \to \textbf{num}$	$T.node = \textbf{new}\ Leaf\,(\textbf{num}, \textbf{num}.val)$

Рис. 6.4. Синтаксически управляемое определение для получения синтаксических деревьев или ориентированных ациклических графов

Пример 6.2. Последовательность шагов, показанная на рис. 6.5, строит ориентированный ациклический граф, показанный на рис. 6.3, обеспечивая по возможности возврат функциями *Node* и *Leaf* существующих узлов, как рассматривалось ранее. Предполагается, что *entry-a* указывает на запись таблицы символов для `a`; то же выполняется и для других идентификаторов.

Когда вызов $Leaf(\textbf{id}, entry\text{-}a)$ повторяется на шаге 2, возвращается узел, созданный предыдущим вызовом, так что $p_2 = p_1$. Аналогично узлы, возвращаемые на шагах 8 и 9, те же, что и возвращаемые на шагах 3 и 4 (т.е. $p_8 = p_3$ и $p_9 = p_4$). Следовательно, узел, возвращаемый на шаге 10, должен быть тем же, что и возвращаемый на шаге 5, т.е. $p_{10} = p_5$. □

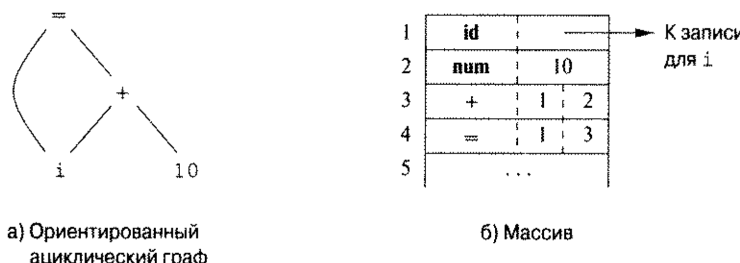
1) $p_1 = Leaf(\mathbf{id}, entry\text{-}a)$
2) $p_2 = Leaf(\mathbf{id}, entry\text{-}a) = p_1$
3) $p_3 = Leaf(\mathbf{id}, entry\text{-}b)$
4) $p_4 = Leaf(\mathbf{id}, entry\text{-}c)$
5) $p_5 = Node('-', p_3, p_4)$
6) $p_6 = Node('*', p_1, p_5)$
7) $p_7 = Node('+', p_1, p_6)$
8) $p_8 = Leaf(\mathbf{id}, entry\text{-}b) = p_3$
9) $p_9 = Leaf(\mathbf{id}, entry\text{-}c) = p_4$
10) $p_{10} = Node('-', p_3, p_4) = p_5$
11) $p_{11} = Leaf(\mathbf{id}, entry\text{-}d)$
12) $p_{12} = Node('*', p_5, p_{11})$
13) $p_{13} = Node('+', p_7, p_{12})$

Рис. 6.5. Шаги построения ориентированного ациклического графа, показанного на рис. 6.3

6.1.2 Метод номера значения для построения ориентированных ациклических графов

Зачастую узлы синтаксического дерева или ориентированного ациклического графа хранятся в массиве записей, как предложено на рис. 6.6. Каждая строка массива представляет одну запись, а следовательно, один узел. Первое поле каждой записи представляет собой код операции, указывая метку узла. На рис. 6.6, *б* у листьев имеется по одному дополнительному полю, в котором хранится лексическое значение (указатель на запись в таблице символов или константа), а внутренние узлы имеют по два дополнительных поля, указывающих левый и правый дочерние узлы.

Мы обращаемся к узлам с использованием целочисленного индекса соответствующей записи в массиве. Это целочисленное значение исторически носит название *номер значения* (value number) узла или выражения, представленного этим узлом. Например, на рис. 6.6 узел с меткой + имеет номер значения 3, а номера значений его левого и правого дочерних узлов равны соответственно 1 и 2. На практике вместо целочисленных индексов можно использовать указатели на записи или ссылки на объекты, но мы в любом случае будем говорить о ссылке на узел как о "номере значения". Будучи сохраненными с применением подходящей структуры данных, номера значений помогают эффективно строить ориентированные



а) Ориентированный ациклический граф

б) Массив

Рис. 6.6. Узлы ориентированного ациклического графа для $i = i + 10$, расположенные в массиве

ациклические графы выражений; как это делается, показано в приведенном ниже алгоритме.

Предположим, что узлы хранятся в массиве, как показано на рис. 6.6, и что обратиться к каждому узлу можно по его номеру. Назовем *сигнатурой* внутреннего узла тройку $\langle op, l, r\rangle$, где op — метка, l — номер значения левого, а r — правого дочернего узла. Унарный оператор можно рассматривать как имеющий значение $r = 0$.

Алгоритм 6.3. Метод номера значения построения узла ориентированного ациклического графа

ВХОД: метка op, узлы l и r.

ВЫХОД: номер значения узла с сигнатурой $\langle op, l, r\rangle$ в массиве.

МЕТОД: выполнить поиск в массиве узла M с меткой op, левым дочерним узлом l и правым дочерним узлом r. Если такой узел найден, вернуть номер значения M. Если нет, создать в массиве новый узел N с меткой op, левым дочерним узлом l и правым дочерним узлом r и вернуть его номер значения. □

Хотя алгоритм 6.3 и выдает на выходе требуемую информацию, сканирование всего массива всякий раз, когда нам требуется получить информацию об одном узле, — метод слишком расточительный, в особенности если в массиве хранятся выражения из всей программы. Более эффективный подход заключается в использовании хеш-таблицы, в которой узлы помещаются в блоки (buckets), в каждом из которых обычно хранится всего лишь несколько узлов. Хеш-таблица представляет собой одну из нескольких структур данных, эффективно поддерживающих словари.[1] Словарь представляет собой абстрактный тип данных, который позволяет добавлять и удалять элементы множества, а также определять, имеется ли

[1]Вопрос о структурах данных, поддерживающих словари, рассматривается в Aho, A.V., J.E. Hopcroft, and J.D. Ullman, *Data Structures and Algorithms*, Addison-Wesley, 1983 (русский перевод — А. Ахо, Д. Хопкрофт, Д. Ульман. *Структуры данных и алгоритмы*. — М.: Издательский дом "Вильямс", 2000).

некоторый элемент в настоящий момент в множестве. Хорошая структура данных для словаря, такая как хеш-таблица, выполняет каждую из указанных операций за константное или близкое к нему время, независимо от размера множества.

Для построения хеш-таблицы для узлов ориентированного ациклического графа требуется *хеш-функция* h, которая вычисляет индекс блока по сигнатуре $\langle op, l, r \rangle$ способом, который равномерно распределяет сигнатуры по блокам, так что маловероятна ситуация, когда в одном блоке находится существенно больше узлов, чем в другом. Индекс блока $h(op, l, r)$ детерминированно вычисляется на основании значений op, l и r, так что сколько бы мы ни повторяли вычисления, для заданной тройки $\langle op, l, r \rangle$ мы всегда будем получать один и тот же индекс.

Блоки могут быть реализованы в виде связанных списков, как показано на рис. 6.7. Массив, проиндексированный хеш-значениями, хранит *заголовки блоков*, каждый из которых указывает на первую ячейку списка. Внутри связанного списка блока каждая ячейка хранит номер значения одного из узлов, хешированных в данный блок, т.е. узел $\langle op, l, r \rangle$ может быть найден в списке, заголовок которого имеет в массиве индекс $h(op, l, r)$.

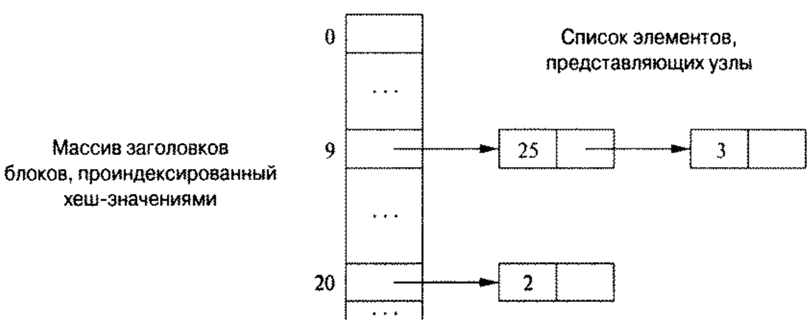


Рис. 6.7. Структура данных для поиска блоков

Таким образом, для данных op, l и r мы вычисляем индекс блока $h(op, l, r)$ и ищем заданный узел в списке ячеек этого блока. Обычно блоков достаточно для того, чтобы списки состояли всего лишь из нескольких узлов. Однако при поиске может потребоваться просканировать все ячейки блока, и для каждого номера значения v, найденного в ячейке списка, следует проверить соответствие сигнатуры искомого узла $\langle op, l, r \rangle$ узлу в списке (как показано на рис. 6.7). Если соответствие найдено, мы возвращаем v, если нет, то, поскольку нам известно, что в других блоках данный узел храниться не может, мы создаем новую ячейку, добавляем ее к списку ячеек блока с индексом $h(op, l, r)$ и возвращаем номер значения этой новой ячейки.

6.1.3 Упражнения к разделу 6.1

Упражнение 6.1.1. Постройте ориентированный ациклический граф для выражения

$$((x + y) - ((x + y) * (x - y))) + ((x + y) * (x - y))$$

Упражнение 6.1.2. Постройте ориентированный ациклический граф и идентифицируйте номера значений для подвыражений следующих выражений, считая + левоассоциативным оператором:

а) $a + b + (a + b)$;

б) $a + b + a + b$;

в) $a + a + (a + a + a + (a + a + a + a))$.

6.2 Трехадресный код

В трехадресном коде в правой части команды имеется не более одного оператора, т.е. не допускаются никакие встроенные арифметические выражения. Таким образом, выражение на исходном языке наподобие `x+y*z` может быть транслировано в трехадресные команды

$$\mathtt{t_1 = y * z}$$
$$\mathtt{t_2 = x + t_1}$$

Здесь t_1 и t_2 — временные имена, сгенерированные компилятором. Такая развертка многооператорных арифметических операций и вложенных инструкций управления потоком делает трехадресный код особенно подходящим для генерации целевого кода и оптимизации, как вы узнаете из глав 8 и 9. Использование имен для промежуточных значений, вычисляемых программой, позволяет легко выполнять перестановки в трехадресном коде.

Пример 6.4. Трехадресный код представляет собой линеаризованное представление синтаксического дерева или ориентированного ациклического графа, в котором явные имена соответствуют внутренним узлам графа. Ориентированный ациклический граф, показанный на рис. 6.3, повторен на рис. 6.8 вместе с соответствующей последовательностью команд трехадресного кода. □

6.2.1 Адреса и команды

Трехадресный код построен на основе двух концепций: адресов и команд. В объектно-ориентированных терминах эти концепции соответствуют классам, а различные виды адресов и команд — подклассам. В качестве альтернативы

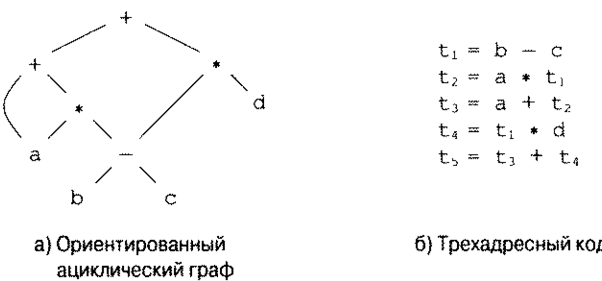


Рис. 6.8. Ориентированный ациклический граф и соответствующий ему трехадресный код

трехадресный код может быть реализован при помощи записей с полями для адресов; в разделе 6.2.2 вкратце обсуждаются записи, называющиеся четверками и тройками.

Адрес может быть одним из следующего списка.

- *Имя.* Для удобства мы позволяем именам исходной программы появляться в трехадресном коде в виде адресов. При реализации исходное имя заменяется указателем на запись в таблице символов, в которой хранится вся информация об имени.

- *Константа.* На практике компилятор должен работать со многими различными типами констант и переменных. Преобразования типов внутри выражений рассматриваются в разделе 6.5.2.

- *Генерируемые компилятором временные переменные.* Оказывается полезным, в особенности в оптимизирующих компиляторах, создание имен, отличающихся друг от друга, всякий раз, когда требуется временная переменная. Эти временные переменные могут комбинироваться при наличии такой возможности в процессе назначения переменным регистров и памяти.

Теперь рассмотрим распространенные трехадресные команды, использующиеся в оставшейся части этой книги. Символьные метки используются в командах, изменяющих поток управления. Символьная метка представляет индекс трехадресной команды в последовательности таких команд. Фактические индексы могут быть заменены метками либо при дополнительном проходе по коду, либо с использованием метода обратных поправок (backpatching), который рассматривается в разделе 6.7. Ниже перечислены наиболее распространенные виды трехадресных команд.

1. Команды присваивания вида $x = y \; op \; z$, где op — бинарная арифметическая или логическая операция, а x, y и z — адреса.

2. Присваивание вида $x = op\ y$, где op — унарная операция. Основные унарные операции включают унарный минус, логическое отрицание и операторы преобразования, которые, например, преобразуют целое число в число с плавающей точкой.

3. *Команды копирования* вида $x = y$, в которых x присваивается значение y.

4. Безусловный переход `goto` L. После этой команды будет выполнена трехадресная команда с меткой L.

5. Условный переход вида `if` x `goto` L и `ifFalse` x `goto` L. Эти команды приводят к тому, что следующей выполняется команда L, если значение x соответственно истинно или ложно. В противном случае, как обычно, выполняется команда, следующая за командой условного перехода.

6. Условные переходы вида `if` x *relop* y `goto` L, которые применяют оператор отношения (<, ==, >= и т.п.) к x и y, и следующей выполняется команда с меткой L, если отношение x *relop* y истинно. В противном случае выполняется команда, следующая за условным переходом.

7. Вызовы процедур и возврат из них реализуются при помощи команд `param` x для передачи параметров; *call* p, n и $y =$ *call* p, n для вызова соответственно процедур и функций; а также `return` y, где y — необязательное возвращаемое значение. Обычно они используются в виде приведенной ниже последовательности трехадресных команд:

   $$\begin{array}{l} \texttt{param}\ x_1 \\ \texttt{param}\ x_2 \\ \dots \\ \texttt{param}\ x_n \\ \texttt{call}\ p, n \end{array}$$

   Данная последовательность генерируется в качестве части вызова процедуры $p(x_1, x_2, \dots, x_n)$. Целое число n указывает количество фактических параметров в *call* p, n и не является излишним в силу того, что вызовы могут быть вложенными. Иначе говоря, некоторые из первых инструкций `param` могут быть параметрами для вызова, который будет выполнен после того, как p вернет значение; это значение станет еще одним параметром более позднего вызова. Реализация вызовов процедур описана в разделе 6.9.

8. Индексированные присваивания вида $x = y[i]$ и $x[i] = y$. Команда $x = y[i]$ присваивает x значение, находящееся в i-й ячейке памяти по отношению

к y. Команда $x[i] = y$ заносит в i-ю по отношению к x ячейку памяти значение y.

9. Присваивание адресов и указателей вида $x = \&y$, $x = *y$ и $*x = y$. Команда $x = \&y$ устанавливает r-значение x равным положению (l-значению) y в памяти[2]. Предположительно y представляет собой имя, возможно, временное, обозначающее выражение с l-значением наподобие $A[i][j]$, а x — имя указателя или временное имя. В команде $x = *y$ под y подразумевается указатель или временная переменная, r-значение которой представляет собой местоположение ячейки памяти. В результате r-значение x становится равным содержимому этой ячейки. И наконец, команда $*x = y$ устанавливает r-значение объекта, указываемого x, равным r-значению y.

Пример 6.5. Рассмотрим инструкцию

```
do i = i+1; while (a[i] < v);
```

На рис. 6.9 показаны две возможные трансляции этой инструкции. Трансляция *а* использует символьную метку L, связанную с первой командой. Трансляция *б* указывает номера позиций команд, произвольным образом начиная отсчет со 100. В обоих случаях последней командой является команда условного перехода к первой команде. Умножение i*8 требуется для массива элементов, каждый из которых занимает 8 единиц памяти. □

а) Символьные метки	б) Номера позиций
`L: t1 = i + 1`	`100: t1 = i + 1`
`   i = t1`	`101: i = t1`
`   t2 = i * 8`	`102: t2 = i * 8`
`   t3 = a [ t2 ]`	`103: t3 = a [ t2 ]`
`   if t3 < v goto L`	`104: if t3 < v goto 100`

Рис. 6.9. Два способа назначения меток трехадресным командам

Выбор доступных операторов представляет собой важный вопрос при проектировании промежуточного представления. Очевидно, что множество операторов должно быть достаточно богатым для реализации операций исходного языка. Операторы, близкие к машинным командам, облегчают реализацию промежуточного кода на целевой машине. Однако, если начальная стадия должна генерировать длинные последовательности команд для некоторых операций исходного языка,

[2] l- и r-значения, как указано в разделе 2.8.3, могут использоваться соответственно в левой и правой частях присваиваний.

то от оптимизатора и генератора кода потребуется большее количество работы для прояснения структуры и генерации эффективного кода для таких операций.

6.2.2 Четверки

Описание трехадресных команд определяет компоненты команд каждого вида, но не указывает, как эти команды должны быть представлены в структуре данных. В компиляторе эти команды могут быть реализованы как объекты или как записи с полями для операторов и операндов. Три такие представления называются четверками, тройками и косвенными тройками.

Четверка (quadruple) имеет четыре поля с именами *op*, arg_1, arg_2 и *result*. Поле *op* содержит внутренний код оператора. Например, трехадресная команда $x = y + z$ представима путем размещения + в *op*, y в arg_1, z в arg_2 и x в *result*. Из этого правила имеется несколько исключений.

1. Команды с унарными операторами наподобие $x =$ `minus` y или $x = y$ не используют arg_2. Заметим, что в случае команды копирования $x = y$ поле *op* равно =, в то время как для большинства других операций оператор присваивания подразумевается неявно.

2. Операторы наподобие `param` не используют ни arg_2, ни *result*.

3. Условные и безусловные переходы помещают в поле *result* целевую метку.

Пример 6.6. Трехадресный код для присваивания `a=b*-c+b*-c;` показан на рис. 6.10, *а*. Для отличия унарного оператора "минус" (как в случае `-c`) от бинарного оператора "минус" (как в случае `b-c`) используется специальный оператор `minus`. Заметим, что трехадресная команда унарного минуса использует только два адреса, как и команда копирования `a` = t_5.

```
t1 = minus c
t2 = b * t1
t3 = minus c
t4 = b * t3
t5 = t2 + t4
 a = t5
```

а) Трехадресный код

	op	arg_1	arg_2	*result*
0	minus	c		t_1
1	*	b	t_1	t_2
2	minus	c	t_3	
3	*	b	t_3	t_4
4	+	t_2	t_4	t_5
5	=	t_5		a
	...			

б) Четверки

Рис. 6.10. Трехадресный код и его представление с использованием четверок

Четверки на рис. 6.10, *б* реализуют трехадресный код, приведенный на рис. 6.10, *а*. □

Для удобочитаемости на рис. 6.10, *б* мы использовали в полях arg_1, arg_2 и *result* фактические идентификаторы `a`, `b` и `c`, а не указатели на соответствующие записи таблицы символов. Временные имена могут быть либо внесены в таблицу символов так же, как и имена, определенные программистом, либо реализованы в виде объектов класса *Temp* с собственными методами.

6.2.3 Тройки

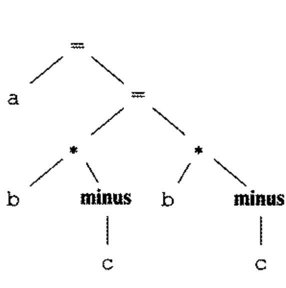
Тройки состоят только из трех полей — *op*, arg_1 и arg_2. Заметим, что на рис. 6.10, *б* поле *result* используется, в основном, для временных имен. Используя тройки, мы обращаемся к результату операции x *op* y по ее позиции, а не при помощи явного временного имени. Таким образом, вместо временной переменной t_1 на рис. 6.10, *б* представление с использованием троек будет обращаться к позиции (0). Числа в скобках представляют указатели в пределах структуры троек. В разделе 6.1.2 позиции или указатели на позиции получили название "номера значений".

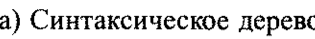
Тройки эквивалентны сигнатурам из алгоритма 6.3. Следовательно, ориентированный ациклический граф и представление с использованием троек эквивалентны. Эта эквивалентность ограничивается выражениями из-за существенного отличия представления потока управления вариантами синтаксических деревьев и трехадресным кодом.

Пример 6.7. Синтаксическое дерево и тройки на рис. 6.11 соответствуют трехадресному коду и четверкам на рис. 6.10. В представлении с использованием троек на рис. 6.11, *б* команда копирования `a = t`$_5$ закодирована в тройке путем размещения `a` в поле arg_1 и (4) в поле arg_2. □

Тернарные операции наподобие $x[i] = y$ требуют двух записей в структуре тройки; например, можно поместить x и i в одну тройку, а y — в другую. Аналогично $x = y[i]$ можно реализовать, рассматривая эту операцию так, как если бы это были две команды, $t = y[i]$ и $x = t$, где t — временная переменная, сгенерированная компилятором. Обратите внимание, что временная переменная t в действительности в тройке не появляется, поскольку обращения к временным значениям выполняются с использованием их позиций в структуре троек.

Преимущество четверок над тройками проявляется в оптимизирующих компиляторах, где зачастую применяется перемещение команд. В случае четверок при перемещении команды, которая вычисляет временное значение t, не требуется внесение изменений в команды, которые используют t. В случае троек обращение к результату операции выполняется с использованием ее позиции, так что перемещение команды может потребовать изменения всех ссылок на ее результат.




а) Синтаксическое дерево

	op	*arg_1*	*arg_2*
0	minus	c	
1	*	b	(0)
2	minus	c	
3	*	b	(2)
4	+	(1)	(3)
5	=	a	(4)
	. . .		

б) Тройки

Рис. 6.11. Представления $a = b * -c + b * -c$

Зачем нужны команды копирования

Простой алгоритм трансляции выражений генерирует для присваиваний команды копирования, как на рис. 6.10, *а*, где t_5 копируется в a вместо непосредственного присваивания t_2 + t_4 переменной a. Каждое подвыражение обычно получает собственную новую переменную для хранения его результата, и только при обработке оператора присваивания = становится известно, куда следует поместить значение полного выражения. Проход оптимизации, возможно, с использованием ориентированного ациклического графа из раздела 6.1.1 в качестве промежуточного представления может определить, что переменная t_5 может быть заменена на a.

Эта проблема не возникает при использовании косвенных троек, которые будут рассмотрены далее.

Косвенные тройки состоят не из списка самих троек, а из списка указателей на тройки. Например, используем массив *instruction* для перечисления указателей на тройки в требуемом порядке. Тогда тройки на рис. 6.11, *б* могут быть представлены так, как показано на рис. 6.12.

При использовании косвенных троек оптимизирующий компилятор может перемещать команды путем переупорядочивания списка *instruction*, никак не влияя на сами тройки. При реализации на Java массив объектов команд аналогичен представлению с косвенными тройками, поскольку Java рассматривает массив элементов как ссылки на объекты.

instruction	
35	(0)
36	(1)
37	(2)
38	(3)
39	(4)
40	(5)
	...

	op	arg_1	arg_2
0	minus	c	
1	*	b	(0)
2	minus	c	
3	*	b	(2)
4	+	(1)	(3)
5	=	a	(4)
	...		

Рис. 6.12. Представление трехадресного кода с применением косвенных троек

6.2.4 Представление в виде статических единственных присваиваний

Представление в виде статических единственных присваиваний (СЕП) является промежуточным представлением, которое облегчает некоторые из оптимизаций кода. СЕП отличается от трехадресного кода. Первое отличие заключается в том, что все присваивания в СЕП выполняются для переменных с различными именами; отсюда следует название *единственного присваивания*. На рис. 6.13 показана одна и та же промежуточная программа в трехадресном коде и в виде статических единственных присваиваний. Обратите внимание на индексы, которые отличают каждое определение переменных p и q в СЕП-представлении.

а) Трехадресный код	б) СЕП-представление
p = a + b	p_1 = a + b
q = p - c	q_1 = p_1 - c
p = q * d	p_2 = q_1 * d
p = e - p	p_3 = e - p_2
q = p + q	q_2 = p_3 + q_1

Рис. 6.13. Промежуточная программа в виде трехадресного кода и в виде СЕП-представления

Одна и та же переменная может быть определена в программе в двух разных путях потока управления. Например, исходная программа

```
if ( flag ) x = -1; else x = 1;
y = x * a;
```

содержит два пути потока управления, в которых определяется переменная x. Если воспользоваться различными именами для x в истинной и ложной частях условной конструкции, то какое имя должно использоваться в присваивании y = x * a? Здесь в игру вступает второе отличие СЕП. СЕП использует для комбинации двух определений x соглашение о записи, именуемой ϕ-функцией:

$$\texttt{if (flag) } x_1 = -1\texttt{; else } x_2 = 1\texttt{;}$$
$$x_3 = \phi(x_1, x_2)\texttt{;}$$

Здесь $\phi(x_1, x_2)$ имеет значение x_1, если поток управления проходит по истинной части условной конструкции, и x_2 — если по ложной части. Иначе говоря, ϕ-функция возвращает значение своего аргумента, соответствующее пути потока управления, который был пройден до команды присваивания с участием ϕ-функции.

6.2.5 Упражнения к разделу 6.2

Упражнение 6.2.1. Транслируйте выражение $a + -(b - c)$ в

а) синтаксическое дерево;

б) четверки;

в) тройки;

г) косвенные тройки.

Упражнение 6.2.2. Повторите упражнение 6.2.1 для следующих инструкций присваивания:

i) `a = b[i] + c[j]`

ii) `a[i] = b*c - b*d`

iii) `x = f(y+1) + 2`

iv) `x = *p + &y`

! **Упражнение 6.2.3.** Покажите, как преобразовать последовательность трехадресных кодов в последовательность, в которой каждая определяемая переменная получает уникальное имя.

6.3 Типы и объявления

Применение типов может быть разбито на две группы.

- *Проверка типов* использует логические правила для того, чтобы судить о поведении программы при ее выполнении. В частности, проверяется, что типы операндов соответствуют типам, требующимся для данного оператора. Например, оператор `&&` в Java требует, чтобы оба его операнда были булевыми; результат также должен принадлежать этому типу.

- *Применение в трансляции.* По типу переменной компилятор может определить, какое количество памяти требуется для данного имени в процессе выполнения программы. Информация о типе необходима, помимо прочего, также для вычисления адреса при обращении к элементу массива, для вставки явного преобразования типов и для выбора верной версии арифметического оператора.

В этом разделе мы рассмотрим типы и размещение в памяти имен, объявленных в процедуре или классе. Реальное распределение памяти для вызова процедуры или объекта происходит во время работы программы, когда вызывается процедура или создается объект. Однако при рассмотрении локальных объявлений во время компиляции можно определить *относительные адреса* имен или компонентов структур данных, которые представляют собой смещение от начала области данных.

6.3.1 Выражения типов

Типы имеют структуру, которую мы представим с использованием *выражений типов* (type expressions). Выражение типа либо представляет собой фундаментальный тип, либо образуется путем применения оператора, называющегося *конструктором типа*, к выражению типа. Множество фундаментальных типов и конструкторов зависит от конкретного языка.

Пример 6.8. Тип массива `int[2][3]` можно прочесть как "массив из 2 массивов из 3 целых чисел каждый" и записать как выражение типа $array(2, array(3, integer))$. Этот тип представлен деревом на рис. 6.14. Оператор $array$ принимает два параметра, число и тип. □

Мы будем использовать следующее определение выражений типа.

- Фундаментальный тип является выражением типа. Типичные фундаментальные типы языка включают *boolean*, *char*, *integer* и *void*; последний тип означает "отсутствие значения".

- Имя типа является выражением типа.

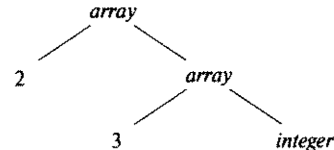


Рис. 6.14. Выражение типа для `int[2][3]`

- Выражение типа может быть образовано путем применения конструктора типа *array* к числу и выражению типа.

- Запись представляет собой структуру данных с именованными полями. Выражение типа может быть образовано путем применения конструктора типа *record* к именам полей и их типам. Типы записи будут реализованы в разделе 6.3.6 путем применения конструктора *record* к таблице символов, содержащей записи для полей.

- Выражение типа может быть образовано путем использования конструктора $\rightarrow$ для типов функций. Запись $s \rightarrow t$ означает "функция, принимающая тип s и возвращающая тип t". Типы функций будут полезны при проверке типов, которая рассматривается в разделе 6.5.

- Если s и t являются выражениями типов, то их декартово произведение $s \times t$ представляет собой выражение типа. Произведения введены для полноты; они могут использоваться для представления списка или кортежа типов (например, параметров функции). Будем считать, что оператор $\times$ левоассоциативен и что он имеет более высокий приоритет, чем $\rightarrow$.

- Выражения типов могут содержать переменные, значениями которых являются выражения типов. Переменные типов, сгенерированные компилятором, будут использоваться в разделе 6.5.4.

Удобным способом представления выражений типов является использование графов. Метод номера значения из раздела 6.1.2 может быть адаптирован для построения ориентированного ациклического графа для выражений типов, внутренние узлы которого соответствуют конструкторам типов, а листья — фундаментальным типам, именам типов и переменным типов; см., например, дерево на рис. 6.14[3].

[3]Поскольку имена типов обозначают выражения типов, они могут приводить к неявным циклам (см. врезку "Имена типов и рекурсивные типы"). Если ребра к именам типов перенаправлены к выражениям типов, обозначаемым этими именами, то получающийся в результате граф может содержать циклы из-за наличия рекурсивных типов.

Имена типов и рекурсивные типы

После определения класса его имя может использоваться в C++ или Java как имя типа; например, рассмотрим `Node` из следующего фрагмента программы:

```
public class Node { ··· }
...
public Node n;
```

Имена могут использоваться для определения рекурсивных типов, которые требуются для структур данных, таких как связанные списки. Псевдокод для элемента списка

```
class Cell { int info; Cell next; ··· }
```

определяет рекурсивный тип `Cell` как класс, который содержит поле `info` и поле `next` типа `Cell`. Подобные рекурсивные типы могут быть определены в C с использованием записей и указателей. Методы, описанные в этой главе, переносятся и на рекурсивные типы.

6.3.2 Эквивалентность типов

Когда два выражения типов эквивалентны? Многие правила проверки типов имеют вид "**если** два выражения типов эквивалентны, **то** вернуть некоторый тип, **иначе** сообщить об ошибке". Потенциальные неоднозначности возникают, когда имена, назначаемые выражениям типов, затем используются в последующих выражениях типов. Ключевой вопрос состоит в том, является ли имя в выражении типа именем или оно представляет собой сокращение для другого выражения типа.

Если выражения типов представлены графами, два типа *структурно эквивалентны* тогда и только тогда, когда выполняется одно из следующих условий.

- Они представляют собой один и тот же фундаментальный тип.
- Они образованы путем применения одного и того же конструктора к структурно эквивалентным типам.
- Один тип представляет собой имя, обозначающее другой тип.

Если имена типов рассматривать как обозначающие сами себя, то первые два условия в приведенном списке приводят к *именной эквивалентности* выражений типов.

При использовании алгоритма 6.3 выражения, эквивалентные в смысле имен, получают одинаковые номера значений. Структурная эквивалентность может быть проверена с использованием алгоритма унификации из раздела 6.5.5.

6.3.3 Объявления

Будем рассматривать типы и объявления с использованием упрощенной грамматики, которая объявляет имена по одному; объявления со списками имен могут обрабатываться так, как рассматривалось в примере 5.10. Грамматика выглядит следующим образом:

$$\begin{array}{lcl} D & \rightarrow & T\ \mathbf{id}\ ;\ D \mid \epsilon \\ T & \rightarrow & B\ C \mid \mathbf{record}\ '\{'\ D\ '\}' \\ B & \rightarrow & \mathbf{int} \mid \mathbf{float} \\ C & \rightarrow & \epsilon \mid [\ \mathbf{num}\ ]\ C \end{array}$$

Фрагмент приведенной выше грамматики, который работает с фундаментальными типами и массивами, использовался для иллюстрации наследуемых атрибутов в разделе 5.3.2. Отличие в этом разделе заключается в том, что мы рассматриваем как сами типы, так и размещение в памяти.

Нетерминал D генерирует последовательность объявлений. Нетерминал T генерирует фундаментальные типы, массивы и записи. Нетерминал B генерирует один из фундаментальных типов **int** и **float**. Нетерминал C (компонент) генерирует строки из нуля или нескольких целых чисел, каждое из которых размещено в квадратных скобках. Тип массива состоит из фундаментального типа, определяемого B, за которым следуют компоненты массива, определяемые нетерминалом C. Тип записи (вторая продукция для T) представляет собой последовательность объявлений полей записи, заключенную в фигурные скобки.

6.3.4 Размещение локальных имен в памяти

Исходя из типа имени можно определить количество памяти, требующееся для данного имени в процессе выполнения программы. Во время компиляции эти величины могут использоваться для назначения каждому имени относительного адреса. Тип и относительный адрес хранятся в записи таблицы символов для данного имени. Данные переменной длины, такие как строки, или данные, размер которых невозможно определить до начала работы программы, такие как динамические массивы, обрабатываются путем резервирования известного фиксированного количества памяти для указателя на данные. Управление памятью во время выполнения программы рассматривается в главе 7.

Предположим, что память состоит из блоков смежных байтов, где байт представляет собой наименьшую единицу адресуемой памяти. Обычно байт состоит

Выравнивание адресов

На размещение в памяти объектов данных сильно влияют ограничения на адреса, имеющиеся у целевой машины. Например, команда сложения целых чисел может ожидать, что эти целые числа *выровнены* (aligned), т.е. размещены в определенных позициях памяти, как, например, по адресу, делящемуся на 4. Хотя массив из 10 символов требует достаточного количества памяти для хранения 10 символов, компилятор может выделить для него 12 байт — очередное кратное 4 число, — оставляя неиспользованными 2 байта. Такая память, остающаяся неиспользованной из-за выравнивания, называется *заполнением* (padding). Когда вопросы экономии памяти выходят на первый план, компилятор может *упаковать* данные так, чтобы заполнения не было, однако при этом в процессе работы программы могут потребоваться дополнительные команды для позиционирования упакованных данных с тем, чтобы программа могла работать с ними так, как если бы они были корректно выровнены.

из восьми битов, а несколько байтов образуют машинное слово. Многобайтные объекты хранятся в памяти в последовательных байтах, и адрес объекта представляет собой адрес его первого байта.

Размер (width) типа равен количеству единиц памяти, необходимому для хранения объекта данного типа. Фундаментальные типы, такие как символы, целые числа и числа с плавающей точкой, требуют целого количества байтов. Для простоты обращения память для агрегатов, таких как массивы и классы, выделяется в виде одного непрерывного блока байтов[4].

Схема трансляции (СУТ) на рис. 6.15 вычисляет типы и их размеры для фундаментальных типов и массивов; записи будут рассматриваться в разделе 6.3.6. СУТ использует синтезируемые атрибуты *type* и *width* для каждого нетерминала и переменные t и w для передачи информации о типе и размере от узла B в дереве разбора к узлу продукции $C \to \epsilon$. В синтаксически управляемом определении t и w представляют собой наследуемые атрибуты C.

Тело T-продукции состоит из нетерминала B, действия и нетерминала C, который показан в следующей строке. Действие между B и C устанавливает t равным $B.type$, а w — равным $B.width$. Если $B \to$ **int**, то $B.type$ устанавливается равным *integer*, а $B.width$ — равным 4, размеру целого числа. Аналогично если $B \to$ **float**, то $B.type$ становится равным *float*, а $B.width$ — 8, размеру числа с плавающей точкой.

[4]Распределение памяти для указателей в C и C++ упрощается, если все указатели имеют одинаковые размеры. Причина этого в том, что память для указателей может быть выделена до того, как станет известен тип объекта, на который он может указывать.

$$
\begin{array}{lcll}
T & \rightarrow & B & \{\ t = B.type;\ w = B.width;\ \} \\
 & & C & \\
B & \rightarrow & \textbf{int} & \{\ B.type = integer;\ B.width = 4;\ \} \\
B & \rightarrow & \textbf{float} & \{\ B.type = float;\ B.width = 8;\ \} \\
C & \rightarrow & \epsilon & \{\ C.type = t;\ C.width = w;\ \} \\
C & \rightarrow & [\ \textbf{num}\ ]\ C_1 & \{\ array(\textbf{num}.value,\ C_1.type); \\
 & & & \quad C.width = \textbf{num}.value \times C_1.width;\ \}
\end{array}
$$

Рис. 6.15. Вычисление типов и их размеров

Продукции для C определяют, генерирует ли T базовый тип или тип массива. Если $C \rightarrow \epsilon$, то t становится равным $C.type$, а w – $C.width$.

В противном случае C определяет компонент массива. Действие для $C \rightarrow [\ \textbf{num}\ ]\ C_1$ образует $C.type$ путем применения конструктора типа *array* к операндам **num**.*value* и $C_1.type$. Например, результатом применения *array* может быть древовидная структура наподобие показанной на рис. 6.14.

Размер массива получается путем умножения размера элемента на количество элементов массива. Если адреса последовательных целых чисел отличаются на 4, то вычисление адреса для массива целых чисел будет включать умножения на 4. Такие умножения предоставляют широкие возможности для оптимизации, так что лучше, если начальная стадия компиляции делает их явными. В этой главе мы игнорируем машинно-зависимые вопросы, такие как выравнивание объектов данных на границу слова.

Пример 6.9. Дерево разбора для выражения `int[2][3]` показано на рис. 6.16 пунктиром. Сплошные линии показывают, каким образом тип и размер передаются от B вниз по цепочке C при помощи переменных t и w, а затем возвращаются вверх по цепочке в виде синтезируемых атрибутов *type* и *width*. Переменные t и w получают значения $B.type$ и $B.width$ перед тем, как будет исследовано поддерево с узлами C. Значения t и w используются в узле $C \rightarrow \epsilon$ для начала вычисления синтезируемых атрибутов вверх по цепочке из узлов C. □

6.3.5 Последовательности объявлений

Языки, такие как C и Java, позволяют обрабатывать все объявления в одной процедуре как группу. Эти объявления могут быть рассредоточены в пределах процедуры Java, но при этом все они обрабатываются в ходе анализа данной

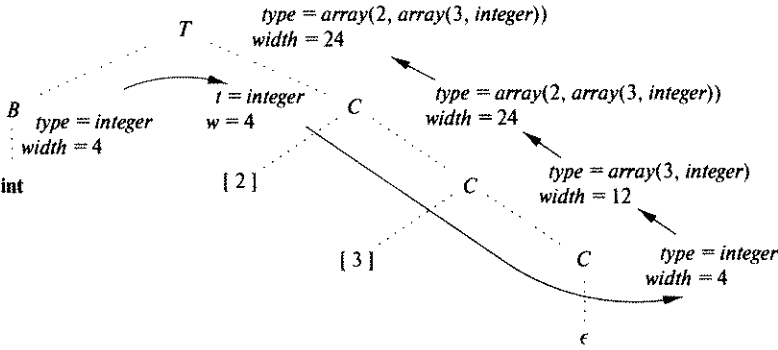


Рис. 6.16. Синтаксически управляемая трансляция типов массивов

процедуры. Поэтому можно воспользоваться переменной, скажем, *offset*, для отслеживания следующего доступного относительного адреса.

Схема трансляции на рис. 6.17 работает с последовательностями объявлений вида T **id**, где T генерирует тип, как на рис. 6.15. Перед рассмотрением первого объявления значение *offset* равно 0. Когда встречается новое имя x, оно вносится в таблицу символов со своим относительным адресом, равным текущему значению *offset*, которое после этого увеличивается на размер x.

$$
\begin{array}{lll}
P \to & & \{\ \mathit{offset} = 0;\ \} \\
 & D & \\
D \to & T\ \mathbf{id}\ ; & \{\ \mathit{top.put}(\mathbf{id}.\mathit{lexeme},\ T.\mathit{type},\ \mathit{offset}); \\
 & & \quad \mathit{offset} \ = \ \mathit{offset} + T.\mathit{width};\ \} \\
 & D_1 & \\
D \to & \epsilon &
\end{array}
$$

Рис. 6.17. Вычисление относительных адресов объявленных имен

Семантическое действие в продукции $D \to T$ **id** ; D_1 создает запись таблицы символов, выполняя *top.put* (**id**.*lexeme*, T.*type*, *offset*). Здесь *top* означает текущую таблицу символов. Метод *top.put* создает запись таблицы символов для **id**.*lexeme* с типом T.*type* и относительным адресом *offset* в области данных.

Инициализация *offset* на рис. 6.17 будет более очевидной, если переписать первую продукцию в одну строку как

$$P \ \to \ \{\mathit{offset} = 0;\}\ D \tag{6.1}$$

Чтобы все действия находились на правых концах продукций, можно использовать нетерминалы, генерирующие ϵ и называющиеся нетерминалами-маркерами (см.

раздел 5.5.4). Используя маркер M, можно записать (6.1) как

$$\begin{array}{lcl} P & \rightarrow & M\ D \\ M & \rightarrow & \epsilon\ \{\mathit{offset} = 0;\} \end{array}$$

6.3.6 Поля в записях и классах

Трансляция объявлений на рис. 6.17 переносится на поля в записях и классах. Типы записей могут быть добавлены к грамматике на рис. 6.15 путем добавления продукции

$$T \rightarrow \textbf{record}\ '\{'\ D\ '\}'$$

Поля в этом типе записи определяются последовательностью объявлений, генерируемых D. Подход, показанный на рис. 6.17, может использоваться и для определения типов и относительных адресов полей при соблюдении следующих условий:

- имена полей в записи должны быть различны, т.е. в объявлениях, генерируемых D, каждое имя может появляться не более одного раза;
- смещение, или относительный адрес, имени поля отсчитывается относительно начала области данных, выделенной для записи.

Пример 6.10. Использование имени x для поля внутри записи не конфликтует с другими применениями этого имени вне записи. Таким образом, три использования **x** в следующих объявлениях различны и не конфликтуют друг с другом:

```
float x;
record { float x; float y; } p;
record { int tag; float x; float y; } q;
```

Последующее присваивание `x = p.x + q.x;` устанавливает значение переменной **x** равным сумме полей **x** в записях `p` и `q`. Заметим, что относительный адрес **x** в `p` отличается от относительного адреса **x** в `q`. □

Для удобства типы записей будут кодироваться с применением как типов, так и относительных адресов их полей, используя для типа записи таблицу символов. Тип записи имеет вид *record* (t), где *record* – конструктор типа, а t – объект, представляющий собой таблицу символов, хранящую информацию о полях данного типа записи.

Схема трансляции на рис. 6.18 состоит из единственной продукции, которая добавляется к продукции на рис. 6.15. Эта продукция содержит два семантических действия. Действие перед D сохраняет существующую таблицу символов,

описываемую переменной *top*, и устанавливает значение *top* равным новой таблице символов. Кроме того, сохраняется текущее значение *offset* и переменной *offset* присваивается нулевое значение. Объявления, сгенерированные D, сохраняют типы и относительные адреса в новой таблице символов. Действие после D создает тип записи с использованием *top*, после чего восстанавливает сохраненные таблицу символов и смещение.

$$
\begin{array}{llll}
T & \rightarrow & \textbf{record}\ '\{' & \{\ Env.push(top);\ top = \textbf{new}\ Env();\\
 & & & \ \ Stack.push(offset);\ offset = 0;\ \}\\
 & & D\ '\}' & \{\ T.type = record(top);\ T.width = offset;\\
 & & & \ \ top = Env.pop();\ offset = Stack.pop();\ \}
\end{array}
$$

Рис. 6.18. Обработка имен полей в записях

Для конкретности действия на рис. 6.18 содержат псевдокод одного из вариантов реализации. Здесь класс *Env* реализует таблицы символов. Вызов *Env.push* (*top*) вносит на вершину стека текущую таблицу символов, обозначаемую переменной *top*. Точно так же переменная *offset* вносится на вершину стека с именем *Stack*. Затем переменная *offset* получает значение, равное 0.

После трансляции объявлений D таблица символов *top* содержит типы и относительные адреса полей в этой записи. Значение *offset* указывает количество памяти, необходимое для хранения всех полей записи. Второе действие устанавливает значение *T.type* равным *record* (*top*), а значение *T.width* – равным *offset*. Затем восстанавливаются значения переменных *top* и *offset*, которые были ранее сохранены в стеках, и этим завершается трансляция данного типа записи.

Данный метод распространяется и на классы, поскольку резервирования памяти для методов не требуется (см. упражнение 6.3.2).

6.3.7 Упражнения к разделу 6.3

Упражнение 6.3.1. Определите типы и относительные адреса идентификаторов в приведенной ниже последовательности объявлений:

```
float x;
record { float x; float y; } p;
record { int tag; float x; float y; } q;
```

! **Упражнение 6.3.2.** Распространите обработку имен полей на рис. 6.18 на классы и иерархии классов с единичным наследованием.

а) Приведите реализацию класса *Env*, которая поддерживает связанные таблицы символов, так что подкласс может либо переопределять имя поля, либо обращаться непосредственно к имени поля надкласса.

б) Приведите схему трансляции, которая выделяет непрерывную область данных для полей класса, включая унаследованные поля. Унаследованные поля должны поддерживать относительные адреса, которые они получают в схеме надкласса.

6.4 Трансляция выражений

Оставшаяся часть данной главы посвящена вопросам, возникающим при трансляции выражений и инструкций. Мы начнем этот раздел с трансляции выражений в трехадресный код. Выражения с более чем одним оператором, наподобие $a+b*c$, будут транслироваться в команды с не более чем одним оператором в команде. Обращения к массивам наподобие $A[i][j]$ будут разворачиваться в последовательность трехадресных команд, вычисляющих адрес, к которому будет выполняться обращение. Проверка типов будет рассмотрена в разделе 6.5, а использование логических выражений для управления потоком — в разделе 6.6.

6.4.1 Операции в выражениях

Синтаксически управляемое определение на рис. 6.19 создает трехадресный код для инструкции присваивания S с использованием атрибута *code* для S и атрибутов *addr* и *code* для выражения E. Атрибуты S.*code* и E.*code* означают трехадресный код S и E соответственно. Атрибут E.*addr* означает адрес, в котором сохраняется значение E. Вспомним из раздела 6.2.1, что адрес может быть именем, константой или временной переменной, сгенерированной компилятором.

Рассмотрим последнюю продукцию синтаксически управляемого определения на рис. 6.19, $E \to$ **id**. Когда выражение представляет собой единственный идентификатор, скажем, x, то x сам по себе хранит значение выражения. Семантические правила для этой продукции определяют атрибут E.*addr* как указывающий на запись в таблице символов для данного экземпляра **id**. Пусть *top* означает текущую таблицу символов. Функция *top.get*, будучи примененной к строковому представлению **id**.*lexeme* данного экземпляра **id**, выбирает из таблицы символов соответствующую запись. E.*code* устанавливается равным пустой строке.

В случае продукции $E \to (E_1)$ трансляция E та же, что и для подвыражения E_1. Следовательно, E.*addr* равно E_1.*addr*, а E.*code* равно E_1.*code*.

Операторы + и унарный - на рис. 6.19 являются представительными операторами типичных языков программирования. Семантические правила для $E \to E_1 + E_2$ генерируют код для вычисления значения E из значений E_1 и E_2. Вычисленные значения сохраняются во вновь создаваемых временных именах. Если значение E_1 сохраняется в E_1.*addr*, а E_2 — в E_2.*addr*, то $E_1 + E_2$ транслируется в $t = E_1.addr + E_2.addr$, где t — новое временное имя. E.*addr* устанавливает-

Продукция	Семантические правила
$S \rightarrow \mathbf{id} = E\ ;$	$S.code = E.code \mid\mid$ $gen(top.get(\mathbf{id}.lexeme)\ '='\ E.addr)$
$E \rightarrow E_1 + E_2$	$E.addr = \mathbf{new}\ Temp\,()$ $E.code = E_1.code \mid\mid E_2.code \mid\mid$ $gen(E.addr\ '='\ E_1.addr\ '+'\ E_2.addr)$
$\mid\ - E_1$	$E.addr = \mathbf{new}\ Temp\,()$ $E.code = E_1.code \mid\mid$ $gen(E.addr\ '='\ '\mathbf{minus}'\ E_1.addr)$
$\mid\ (\ E_1\ )$	$E.addr = E_1.addr$ $E.code = E_1.code$
$\mid\ \mathbf{id}$	$E.addr = top.get(\mathbf{id}.lexeme)$ $E.code = ''$

Рис. 6.19. Трехадресный код для выражений

ся равным t. Последовательность различных временных имен $t_1, t_2, \ldots$ создается последовательными вызовами **new** *Temp* ().

Для удобства мы используем запись $gen\,(x\ '='\ y\ '+'\ z)$ для представления трехадресной команды $x = y + z$. Выражения, находящиеся на местах переменных x, y и z, при передаче в *gen* вычисляются, а строки в кавычках наподобие $'='$ передаются буквально[5]. Другие трехадресные команды создаются аналогично путем применения *gen* к комбинации выражений и строк.

При трансляции продукции $E \rightarrow E_1 + E_2$ семантические правила на рис. 6.19 строят $E.code$ путем конкатенации $E_1.code$, $E_2.code$ и команды сложения значений E_1 и E_2. Команда помещает результат сложения в новое временное имя для E, обозначаемое при помощи $E.addr$.

Трансляция $E \rightarrow -E_1$ аналогична. Правила создают новое временное имя для E и генерируют команду для выполнения операции унарного минуса.

Наконец, продукция $S \rightarrow \mathbf{id} = E\ ;$ генерирует команды, которые присваивают значение выражения E идентификатору **id**. Семантическое правило для этой

[5]В синтаксически управляемых определениях *gen* создает и возвращает команду. В схеме трансляции *gen* создает команду и инкрементно помещает ее в поток генерируемых команд.

продукции использует функцию *top.get* для определения адреса идентификатора, представленного **id**, как и в правилах для продукции $E \to \mathbf{id}$. *S.code* состоит из команд для вычисления значения E в адрес *E.addr*, после чего следует присваивание с использованием адреса *top.get* (**id**.*lexeme*) для данного экземпляра **id**.

Пример 6.11. Синтаксически управляемое определение на рис. 6.19 транслирует инструкцию присваивания `a = b+-c;` в последовательность трехадресных команд

```
t1 = minus c
t2 = b + t1
a = t2
```

□

6.4.2 Инкрементная трансляция

Атрибуты-коды могут быть очень длинными строками, так что обычно они генерируются инкрементно, как обсуждалось в разделе 5.5.2. Таким образом, вместо построения *E.code* так, как показано на рис. 6.19, можно расположить действия таким образом, чтобы они генерировали только новые трехадресные команды, как это сделано в схеме трансляции на рис. 6.20. При инкрементном подходе *gen* не только конструирует трехадресную команду, но и добавляет ее к последовательности уже сгенерированных команд. Эта последовательность может как остаться в памяти для дальнейшей обработки, так и быть инкрементно выведена.

$$
\begin{array}{lll}
S \to \mathbf{id} = E\ ; & \{\ gen(\ top.get(\mathbf{id}.lexeme)\ '='\ E.addr);\ \} \\[2ex]
E \to E_1 + E_2 & \{\ E.addr = \mathbf{new}\ Temp\,(); \\
 & \quad gen(E.addr\ '='\ E_1.addr\ '+'\ E_2.addr);\ \} \\[2ex]
\quad\ \mid\ -E_1 & \{\ E.addr = \mathbf{new}\ Temp\,(); \\
 & \quad gen(E.addr\ '='\ '\mathbf{minus}'\ E_1.addr);\ \} \\[2ex]
\quad\ \mid\ (\ E_1\ ) & \{\ E.addr = E_1.addr;\ \} \\[2ex]
\quad\ \mid\ \mathbf{id} & \{\ E.addr = top.get(\mathbf{id}.lexeme);\ \}
\end{array}
$$

Рис. 6.20. Инкрементная генерация трехадресного кода для выражений

Схема трансляции на рис. 6.20 генерирует тот же код, что и синтаксически управляемое определение на рис. 6.19. В случае инкрементного подхода атрибут

code не используется, поскольку существует единственная последовательность команд, создаваемая последовательными вызовами *gen*. Например, семантическое правило для $E \to E_1 + E_2$ на рис. 6.20 просто вызывает *gen* для генерации команды сложения; команды вычисления E_1 в $E_1.addr$ и E_2 в $E_2.addr$ к этому моменту уже сгенерированы.

Подход, представленный на рис. 6.20, может использоваться также и для построения синтаксического дерева. Новое семантическое действие для $E \to E_1 + + E_2$ создает узел с использованием конструктора:

$$E \quad \to \quad E_1 + E_2 \ \{E.addr = \mathbf{new}\ Node\,('+', E_1.addr, E_2.addr)\,;\}$$

Здесь атрибут *addr* представляет адрес узла, а не переменной или константы.

6.4.3 Адресация элементов массива

Быстрое обращение к элементам массива легко осуществимо, когда они хранятся в блоке смежных ячеек памяти. В C и Java элементы массива из n элементов нумеруются как $0, 1, \ldots, n-1$. Если размер каждого элемента массива равен w, то i-й элемент массива A начинается в ячейке

$$base + i \times w \tag{6.2}$$

Здесь *base* — относительный адрес памяти, выделенной для массива, т.е. *base* — относительный адрес $A\,[0]$.

Формула (6.2) обобщается на два и более измерений. В случае двух измерений в C мы записываем элемент i_2 в строке i_1 как $A\,[i_1]\,[i_2]$. Пусть w_1 — размер строки, а w_2 — размер элемента в строке. Относительный адрес $A\,[i_1]\,[i_2]$ в таком случае может быть вычислен по формуле

$$base + i_1 \times w_1 + i_2 \times w_2 \tag{6.3}$$

В случае k размерностей формула принимает вид

$$base + i_1 \times w_1 + i_2 \times w_2 + \cdots + i_k \times w_k \tag{6.4}$$

Здесь w_j, $1 \leqslant j \leqslant k$, представляет собой обобщение w_1 и w_2 из (6.3).

Относительный адрес может быть вычислен и иначе, с использованием количества элементов n_j в каждой из размерностей j массива и размера $w = w_k$ одного элемента массива. В случае двух размерностей (т.е. $k = 2$ и $w = w_2$) начальная ячейка $A\,[i_1]\,[i_2]$ вычисляется как

$$base + (i_1 \times n_2 + i_2) \times w \tag{6.5}$$

В случае k размерностей тот же адрес, что и в формуле (6.4), вычисляется как

$$base + ((\cdots((i_1 \times n_2 + i_2) \times n_3 + i_3)\cdots) \times n_k + i_k) \times w \tag{6.6}$$

Вообще говоря, массивы не обязаны нумероваться начиная с нулевого элемента. Например, в одномерном массиве элементы могут иметь номера $low, low + 1, \ldots, high$, а $base$ может быть относительным адресом элемента $A[low]$. Формула (6.2) для адреса $A[i]$ при этом заменяется следующей:

$$base + (i - low) \times w \tag{6.7}$$

Выражения (6.2) и (6.7) могут быть переписаны в виде $i \times w + c$, где подвыражение $c = base - low \times w$ может быть предвычислено в процессе компиляции. Заметим, что $c = base$ при $low = 0$. Будем считать, что c хранится в записи таблицы символов для A, так что относительный адрес $A[i]$ получается путем простого добавления $i \times w$ к c.

Предвычисления в процессе компиляции могут применяться и для адресов элементов многомерных массивов; см. упражнение 6.4.5. Однако существует одна ситуация, в которой не может использоваться предвычисление в процессе компиляции: в случае динамического размера массива. Если мы не знаем значений low и $high$ (или их многомерных обобщений) в процессе компиляции, то не можем вычислить и такие константы, как c. В таком случае формула типа (6.7) должна вычисляться в процессе выполнения программы так, как она записана.

Приведенные выше вычисления адресов основаны на построчной схеме размещения массивов, используемой в C. Двумерный массив обычно хранится одним из двух способов, либо *построчно* (row-major), либо *постолбцово* (column-major). На рис. 6.21 показано построчное (*а*) и постолбцовое (*б*) размещение массива A размером 2×3. Постолбцовое размещение используется в языках семейства Fortran.

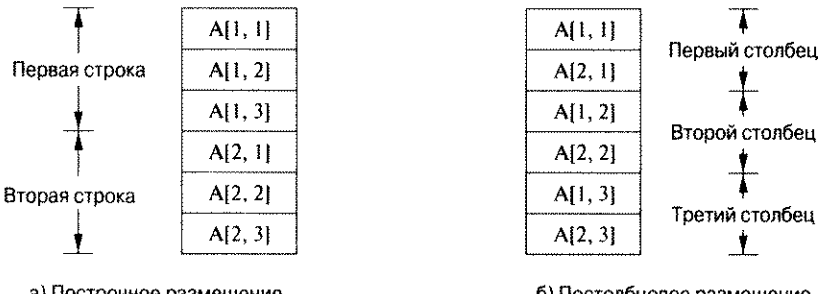


Рис. 6.21. Схемы размещения двумерного массива

Можно обобщить построчное и постолбцовое размещение для многих измерений. Обобщение построчной формы заключается в хранении элементов таким образом, что при сканировании блока памяти правый индекс меняется наиболее быстро; постолбцовое размещение, напротив, приводит к наиболее быстрому изменению левого индекса.

6.4.4 Трансляция обращений к массиву

Главная проблема при генерации кода для обращения к массивам состоит в связывании формул вычисления адресов из раздела 6.4.3 с грамматикой для обращения к массивам. Пусть нетерминал L генерирует имя массива с последовательностью индексных выражений:

$$L \to L\ [E] \mid \mathbf{id}\ [E]$$

Как в C и Java, будем считать, что наименьший номер элемента массива равен 0. Будем вычислять адрес, исходя из размеров с использованием формулы (6.4), а не из количества элементов, как в (6.6). Схема трансляции на рис. 6.22 генерирует трехадресный код для выражений с обращениями к массивам. Она состоит из продукций и семантических действий, представленных на рис. 6.20, к которым добавлены продукции, включающие нетерминал L.

Нетерминал L имеет три синтезируемых атрибута.

1. $L.addr$ обозначает временную переменную, использующуюся при вычислении смещения для обращения к массиву путем суммирования членов $i_j \times w_j$, как в (6.4).

2. $L.array$ — это указатель на запись таблицы символов для имени массива. Базовый адрес массива $L.array.base$ используется для определения фактического l-значения ссылки на массив после анализа всех индексных выражений.

3. $L.type$ представляет собой тип подмассива, сгенерированного L. Считаем, что размер любого типа t равен $t.width$. В качестве атрибутов используются типы, а не размеры, поскольку в любом случае потребуется обеспечить проверку типов. Считаем также, что для любого типа t значение $t.elem$ дает тип элемента.

Продукция $S \to \mathbf{id} = E$; представляет присваивание переменной, не являющейся элементом массива и обрабатываемой, как обычно. Семантические действия для $S \to L = E$; генерируют команду индексированного копирования, которая присваивает значение выражения E ячейке памяти, указываемой ссылкой на массив L. Вспомним, что атрибут $L.array$ дает нам запись таблицы символов для указанного массива. Базовый адрес массива — адрес его нулевого элемента — равен $L.array.base$. Атрибут $L.addr$ обозначает временную переменную, хранящую смещение для обращения к массиву, генерируемому L. Таким образом, искомая ячейка памяти для обращения к массиву — $L.array.base\,[L.addr]$. Сгенерированная команда копирует r-значение из адреса $E.addr$ в ячейку памяти для L.

Продукции $E \to E_1 + E_2$ и $E \to \mathbf{id}$ те же, что и ранее. Семантическое действие для новой продукции $E \to L$ генерирует код для копирования значения

$$
\begin{array}{llll}
S & \rightarrow & \mathbf{id} = E \ ; & \{ \ gen(\ top.get(\mathbf{id}.lexeme) \ '=' \ E.addr); \ \} \\
 & | & L = E \ ; & \{ \ gen(L.addr.base \ '[' \ L.addr \ ']' \ '=' \ E.addr); \ \} \\
E & \rightarrow & E_1 + E_2 & \{ \ E.addr = \mathbf{new} \ Temp\,(); \\
 & & & \ \ gen(E.addr \ '=' \ E_1.addr \ '+' \ E_2.addr); \ \} \\
 & | & \mathbf{id} & \{ \ E.addr = top.get(\mathbf{id}.lexeme); \ \} \\
 & | & L & \{ \ E.addr = \mathbf{new} \ Temp\,(); \\
 & & & \ \ gen(E.addr \ '=' \ L.array.base \ '[' \ L.addr \ ']'); \ \} \\
L & \rightarrow & \mathbf{id} \ [\ E \ ] & \{ \ L.array = top.get(\mathbf{id}.lexeme); \\
 & & & \ \ L.type = L.array.type.elem; \\
 & & & \ \ L.addr = \mathbf{new} \ Temp\,(); \\
 & & & \ \ gen(L.addr \ '=' \ E.addr \ '*' \ L.type.width); \ \} \\
 & | & L_1 \ [\ E \ ] & \{ \ L.array = L_1.array; \\
 & & & \ \ L.type = L_1.type.elem; \\
 & & & \ \ t = \mathbf{new} \ Temp\,(); \\
 & & & \ \ L.addr = \mathbf{new} \ Temp\,(); \\
 & & & \ \ gen(t \ '=' \ E.addr \ '*' \ L.type.width); \ \} \\
 & & & \ \ gen(L.addr \ '=' \ L_1.addr \ '+' \ t); \ \}
\end{array}
$$

Рис. 6.22. Семантические действия для работы с массивами

из ячейки памяти, описываемой L, в новую временную переменную. Эта ячейка памяти, как указывалось при рассмотрении продукции $S \rightarrow L = E;$, представляет собой $L.array.base\,[L.addr]$. Как и ранее, атрибут $L.array$ дает имя массива, а $L.array.base$ — его базовый адрес. Атрибут $L.addr$ обозначает переменную, в которой хранится значение смещения. Код для обращения к массиву помещает r-значение из ячейки памяти, которая вычисляется по заданным базовому адресу и смещению, в новую временную переменную, представленную атрибутом $E.addr$.

Пример 6.12. Пусть a — массив целых чисел размером 2×3 и пусть c, i и j — целые числа. Тип a — $array\,(2, array\,(3, integer))$. Его размер w равен 24 в предпо-

ложении, что размер целого числа равен 4. Тип `a[i]` — *array* $(3, integer)$, а размер $w_1 = 12$. Тип `a[i][j]` — *integer*.

Аннотированное дерево разбора для выражения `c+a[i][j]` показано на рис. 6.23. Выражение транслируется в последовательность трехадресных команд на рис. 6.24. Как обычно, имя каждого идентификатора используется для обращения к его записи в таблице символов. □

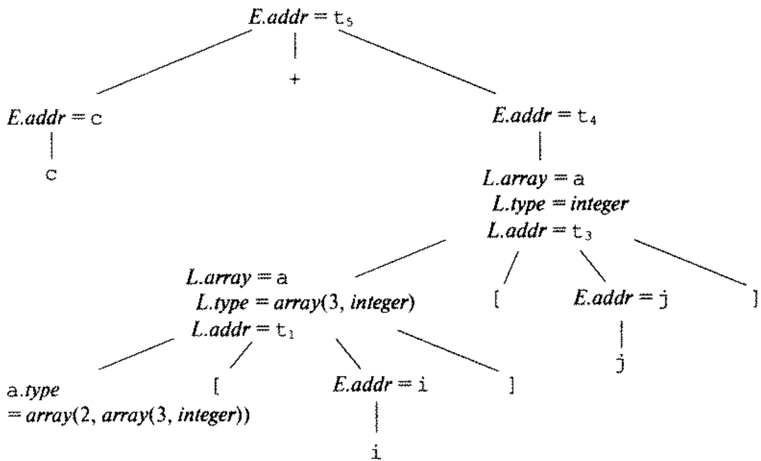


Рис. 6.23. Аннотированное дерево разбора для выражения `c+a[i][j]`

$$
\begin{aligned}
t_1 &= i * 12\\
t_2 &= j * 4\\
t_3 &= t_1 + t_2\\
t_4 &= a\ [\ t_3\ ]\\
t_5 &= c + t_4
\end{aligned}
$$

Рис. 6.24. Трехадресный код для выражения `c+a[i][j]`

6.4.5 Упражнения к разделу 6.4

Упражнение 6.4.1. Добавьте к трансляции на рис. 6.19 правила для следующих продукций:

а) $E \to E_1 * E_2$;

б) $E \to +E_1$ (унарный плюс).

Упражнение 6.4.2. Повторите упражнение 6.4.1 для инкрементной трансляции на рис. 6.20.

Упражнение 6.4.3. Воспользуйтесь трансляцией на рис. 6.22 для следующих присваиваний:

а) `x = a[i] + b[j]`

б) `x = a[i][j] + b[i][j]`

! в) `x = a[b[i][j]][c[k]]`

! **Упражнение 6.4.4.** Преобразуйте трансляцию на рис. 6.22 для обращения к массивам в стиле Fortran, т.е. к **id** $[E_1, E_2, \ldots, E_n]$ в n-мерном случае.

Упражнение 6.4.5. Обобщите формулу (6.7) для многомерных массивов и укажите, какие значения могут храниться в таблице символов и использоваться для вычисления смещений. Рассмотрите следующие случаи.

а) Двумерный массив A, хранящийся построчно. Первое измерение имеет индексы от l_1 до h_1, второе — от l_2 до h_2. Размер одного элемента массива равен w.

б) То же, что и в п. a, но массив хранится постолбцово.

! в) k-мерный массив A, хранящийся построчно. Размер одного элемента массива равен w. j-е измерение имеет индексы от l_j до h_j.

! г) То же, что и в п. a, но массив хранится постолбцово.

Упражнение 6.4.6. Массив целых чисел $A[i,j]$ имеет индекс i со значениями в диапазоне от 1 до 10 и индекс j со значениями от 1 до 20. Размер каждого целого числа — 4 байта. Предположим, что массив A хранится построчно, начиная с байта 0. Найдите положения следующих элементов массива:

а) $A[5,5]$; б) $A[10,8]$; в) $A[3,17]$.

Упражнение 6.4.7. Выполните упражнение 6.4.6, считая, что массив A размещается в памяти постолбцово.

Упражнение 6.4.8. Массив действительных чисел $A[i,j,k]$ имеет индекс i со значениями в диапазоне от 1 до 4, индекс j со значениями от 0 до 4 и индекс k со значениями от 5 до 10. Размер каждого действительного числа — 8 байт. Предположим, что массив A хранится построчно, начиная с байта 0. Найдите положения следующих элементов массива:

а) $A[3,4,5]$; б) $A[1,2,7]$; в) $A[4,3,9]$.

Упражнение 6.4.9. Выполните упражнение 6.4.8, считая, что массив A размещается в памяти постолбцово.

Символьные размеры типов

Промежуточный код должен быть относительно независимым от целевой машины, так что оптимизатор не должен сильно изменяться при замене генератора кода генератором для другой целевой машины. Однако, как показывают вычисления размеров типов, схема трансляции зависит от встроенных фундаментальных типов. Так, в примере 6.12 предполагается, что размер каждого элемента целочисленного массива — 4 байт. Некоторые промежуточные коды, например P-код для Pascal, предоставляют генератору кода определять размер элемента массива, так что промежуточный код не зависит от размера машинного слова. В нашей схеме трансляции мы можем добиться того же эффекта, заменяя значение 4 (размер целого числа) символьной константой.

6.5 Проверка типов

Для выполнения *проверки типов* (type checking) компилятор должен назначить каждому компоненту исходной программы выражение типа. Затем компилятор должен определить, удовлетворяют ли эти выражения типов набору логических правил, которые называются *системой типов* исходного языка программирования.

Проверка типов потенциально способна находить ошибки в программах. В принципе, любая проверка может выполняться динамически, если целевой код хранит не только значение элемента, но и его тип. *Надежная* система типов (sound type system) устраняет необходимость динамической проверки типов во избежание связанных с ними ошибок, так как позволяет нам статически определить, что такие ошибки не могут возникнуть в процессе работы программы. Реализация языка является *строго типизированной*, если компилятор гарантирует, что скомпилированная программа будет выполняться без ошибок, связанных с типами.

Идеи проверки типов используются не только для компиляции, но и для повышения безопасности систем, допускающих импорт и выполнение программных модулей. Программы Java компилируются в машинно-независимый байт-код, который включает детальную информацию о типах операций. Импортируемый код перед выполнением проходит проверку, чтобы избежать как случайных ошибок, так и предумышленных злонамеренных действий.

6.5.1 Правила проверки типов

Проверка типов может принимать два вида: синтеза и выведения. *Синтез типа* (type synthesis) строит тип выражения из типов его подвыражений. Он требует, чтобы имена были объявлены до их использования. Тип $E_1 + E_2$ определяется

в терминах типов E_1 и E_2. Типичное правило синтеза типа имеет вид

$$\begin{array}{c}\textbf{if } f \text{ имеет тип } s \to t \textbf{ and } x \text{ имеет тип } s,\\ \textbf{then} \text{ выражение } f\,(x) \text{ имеет тип } t\end{array} \tag{6.8}$$

Здесь x и f означают выражения, а $s \to t$ – функцию от s, возвращающую t. Это правило для функций с одним аргументом переносится на функции с несколькими аргументами. Правило (6.8) может быть распространено на $E_1 + E_2$ путем рассмотрения сложения как применения функции $add\,(E_1, E_2)$[6].

Выведение типа (type inference) определяет тип языковой конструкции из способа ее использования. Забегая вперед, к примерам из раздела 6.5.4, рассмотрим функцию *null*, предназначенную для проверки, является ли список пустым. Тогда исходя из использования этой функции $null\,(x)$ можно сказать, что x должно быть списком. Тип элементов x неизвестен; все, что мы знаем, – это то, что x должно быть списком элементов некоторого типа, в настоящий момент неизвестного.

Говорить о неизвестных типах нам позволяют переменные, представляющие выражения типов. Будем использовать греческие буквы $\alpha, \beta, \ldots$ для переменных типа в выражениях типов.

Типичное правило для вывода типа имеет вид

$$\begin{array}{l}\textbf{if } f\,(x) \text{ является выражением},\\ \quad\textbf{then } f \text{ имеет тип } \alpha \to \beta \text{ для некоторых } \alpha \text{ и } \beta \textbf{ and } x \text{ имеет тип } \alpha\end{array} \tag{6.9}$$

Выведение типов необходимо в языках наподобие ML, которые проверяют типы, но не требуют объявления имен.

В этом разделе мы рассмотрим проверку типов выражений. Правила для проверки инструкций аналогичны правилам для выражений. Например, мы рассматриваем условную инструкцию "**if** (E) S;" как если бы это было применение функции *if* к E и S. Используем для обозначения отсутствия значения специальный тип *void*. Тогда функция *if* должна применяться к аргументам типа *boolean* и *void*; результат ее применения имеет тип *void*.

6.5.2 Преобразования типов

Рассмотрим выражение наподобие $i + t$, где x имеет тип числа с плавающей точкой, а i – целочисленный тип. Поскольку представление целых чисел и чисел с плавающей точкой в компьютере различно и для операций с целыми числами и числами с плавающей точкой используются различные машинные команды, компилятору может потребоваться преобразовать один из операндов оператора +,

---

[6]Термин "синтез" будет применяться, даже если при определении типов используется некоторая контекстная информация. В случае перегруженных функций, когда одно и то же имя относится более чем к одной функции, в некоторых языках программирования может потребоваться рассмотрение контекста $E_1 + E_2$.

чтобы гарантировать, что оба операнда при выполнении сложения имеют один и тот же тип.

Предположим, что целые числа при необходимости преобразуются в числа с плавающей точкой с использованием унарного оператора `(float)`. Например, в приведенном ниже коде для выражения `2*3.14` целое число 2 преобразуется в число с плавающей точкой:

```
t1 = (float) 2
t2 = t1 * 3.14
```

Мы можем расширить такие примеры, рассматривая разные версии операторов для целых чисел и для чисел с плавающей точкой, например `int*` для умножения целых чисел и `float*` для умножения чисел с плавающей точкой.

Синтез типов будет проиллюстрирован путем расширения схемы из раздела 6.4.2 для трансляции выражений. Введем дополнительный атрибут *E.type*, который может принимать значение *integer* или *float*. Правило, связанное с продукцией $E \to E_1 + E_2$, строит псевдокод

if $(E_1.type = integer$ **and** $E_2.type = integer)$ $E.type = integer$;
else if $(E_1.type = float$ **and** $E_2.type = integer)$ $\cdots$
...

При увеличении количества типов, которые могут быть преобразованы, резко растет количество случаев, которые должны быть рассмотрены. Следовательно, для большого количества типов становится важной аккуратная организация семантических действий.

Правила преобразования типов варьируются от языка к языку. Правила языка программирования Java, показанные на рис. 6.25, различают *расширяющие преобразования*, предназначенные для сохранения информации, и *сужающие преобразования*, которые могут приводить к потере информации. Расширяющие преобразования показаны в иерархии на рис. 6.25, *а*: любой тип в иерархии может быть расширен до типа, находящегося в иерархии выше. Таким образом, *char* может быть расширен до *int* или *float*, но не может быть расширен до *short*. Сужающие правила представлены графом на рис. 6.25, *б*: тип s может быть сужен до типа t, если в графе имеется путь от s до t. Заметим, что *char*, *short* и *byte* попарно преобразуемы друг в друга.

Преобразование из одного типа в другой называется *неявным*, если оно выполняется компилятором автоматически. Неявные преобразования типов во многих языках ограничены расширяющими преобразованиями. Преобразование называется *явным*, если программист должен что-то написать для того, чтобы такое преобразование было выполнено.

Семантические действия для проверки $E \to E_1 + E_2$ используют две функции.

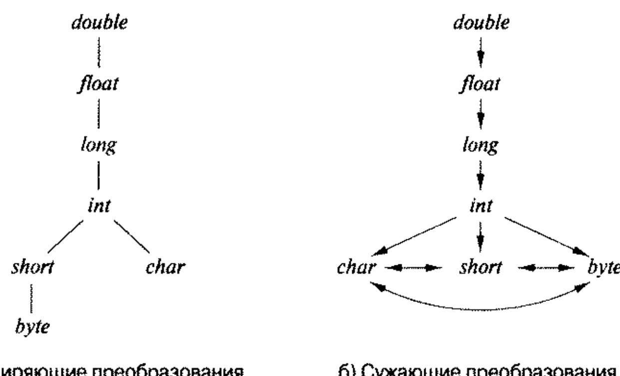


Рис. 6.25. Преобразования примитивных типов Java

1. Функция $\max(t_1, t_2)$ принимает два типа, t_1 и t_2, и возвращает максимальный из этих двух типов (или их наименьшую верхнюю границу) в иерархии расширения. Она объявляет об ошибке, если либо t_1, либо t_2 нет в иерархии, например если тип является массивом или указателем.

2. Функция $widen(a, t, w)$ генерирует преобразование типов при необходимости расширения значения по адресу a типа t в значение типа w. Если типы t и w совпадают, просто возвращается значение, хранящееся по адресу a. В противном случае генерируется команда, выполняющая преобразование и помещающая результат во временную переменную, которая и возвращается функцией. Псевдокод функции *widen* в предположении, что единственные имеющиеся типы — *integer* и *float*, показан на рис. 6.26.

```
Addr widen(Addr a, Type t, Type w)
    if ( t = w ) return a;
    else if ( t = integer and w = float ) {
        temp = new Temp();
        gen(temp '=' '(float)' a);
        return temp;
    }
    else error;
}
```

Рис. 6.26. Псевдокод функции *widen*

Семантическое действие для $E \rightarrow E_1 + E_2$ на рис. 6.27 иллюстрирует, каким образом преобразования типов могут быть добавлены в схему трансляции выражений на рис. 6.20. В семантическом действии временная переменная a_1

представляет собой либо E_1.*addr* (если тип E_1 не требуется преобразовывать в тип E), либо новую временную переменную, возвращаемую функцией *widen*, если такое преобразование необходимо. Аналогично a_2 представляет собой либо E_2.*addr*, либо новую временную переменную, хранящую преобразованное значение E_2. Если оба типа равны *integer* или оба типа — *float*, то преобразование не требуется. Однако в общем случае может оказаться, что единственным способом сложить значения двух разных типов является их преобразование в третий тип.

$$
\begin{array}{lll}
E \rightarrow E_1 + E_2 & \{ & E.type = max(E_1.type, E_2.type); \\
& & a_1 = widen(E_1.addr, E_1.type, E.type); \\
& & a_2 = widen(E_2.addr, E_2.type, E.type); \\
& & E.addr = \mathbf{new}\ Temp\,(); \\
& & gen(E.addr\ '='\ a_1\ '+'\ a_2); \ \}
\end{array}
$$

Рис. 6.27. Добавление преобразований типов в вычисление выражения

6.5.3 Перегрузка функций и операторов

Перегруженный (overloaded) символ имеет различные значения в зависимости от контекста. Перегрузка *разрешается* (resolved), когда для каждого появления имени однозначно определяется его единственное значение. В этом разделе мы ограничимся перегрузкой, которая может быть разрешена путем рассмотрения аргументов функции, как в языке программирования Java.

Пример 6.13. Оператор + в Java означает либо конкатенацию строк, либо сложение в зависимости от типов его операндов. Перегруженными могут быть и пользовательские функции, как в случае

```
void err() { ··· }
void err(String s) { ··· }
```

Обратите внимание, что выбрать требующуюся функцию `err` из двух ее вариантов можно на основании передаваемых функции аргументов. □

Ниже приведено правило синтеза типа для перегруженных функций:

$$
\begin{array}{ll}
\textbf{if } f \text{ может иметь тип } s_i \rightarrow t_i,\ 1 \leqslant i \leqslant n, \text{ где } s_i \neq s_j \text{ при } i \neq j & \\
\textbf{and } x \text{ имеет тип } s_k \text{ для некоторого } 1 \leqslant k \leqslant n, & (6.10) \\
\textbf{then} \text{ выражение } f\,(x) \text{ имеет тип } t_k &
\end{array}
$$

Для разрешения перегрузки на основании типов аргументов к выражениям типа может быть применен метод номера значения из раздела 6.1.2. В ориентированном ациклическом графе, представляющем выражение типа, каждому узлу

назначается целочисленный индекс, называющийся номером значения. Применяя алгоритм 6.3, строим сигнатуру узла, состоящую из его метки и номеров значений его дочерних узлов в порядке слева направо. Сигнатура функции состоит из имени функции и типов ее аргументов. Утверждение, что можно выполнить разрешение перегрузки на основании типов аргументов, эквивалентно утверждению, что можно выполнить разрешение перегрузки на основании сигнатур.

Не всегда возможно выполнить разрешение перегрузки, рассматривая только аргументы функции. В языке программирования Ada отдельное подвыражение может иметь множество возможных типов, для выбора единственного среди которых контекст должен предоставлять достаточное количество информации (см. упражнение 6.5.2).

6.5.4 Выведение типа и полиморфные функции

Выведение типа полезно в языках наподобие ML, которые строго типизированы, но не требуют объявления имен до их использования. Выведение типов гарантирует согласованность использования имен.

Термин "полиморфность" относится к любому фрагменту кода, который может быть выполнен с аргументами разных типов. В этом разделе мы рассмотрим *параметрический полиморфизм*, характеризующийся параметрами или переменными типа. На рис. 6.28 приведен работающий фрагмент программы на языке программирования ML, в котором определяется функция *length*. Тип *length* может быть описан как "для любого типа α *length* отображает список элементов типа α на целое число".

fun *length*(x) = **if** *null*(x) **then** 0 **else** *length*(*tl*(x)) + 1

Рис. 6.28. Программа ML для определения длины списка

Пример 6.14. На рис. 6.28 ключевое слово **fun** указывает определение функции; функции могут быть рекурсивными. Приведенный фрагмент определяет функцию *length* с одним параметром x. Тело функции состоит из условного выражения. Предопределенная функция *null* проверяет, является ли список пустым, а предопределенная функция *tl* (от "tail" – "хвост") возвращает остаток списка после удаления из него первого элемента.

Функция *length* определяет длину, или количество элементов, списка x. Все элементы списка должны иметь один и тот же тип, но функция *length* может быть применена к спискам с разными типами элементов. В приведенном далее выражении *length* применяется к спискам двух различных типов (элементы списка заключены в квадратные скобки):

$$length\,([\texttt{"sun"}, \texttt{"mon"}, \texttt{"tue"}]) + length\,([10, 9, 8, 7]) \qquad (6.11)$$

Список строк имеет длину 3, а список целых чисел — длину 4, так что вычисление выражения (6.11) дает 7. □

Использование символа $\forall$ (читается как "для любого типа") и конструктора типа *list* дает возможность записать тип функции *length* как

$$\forall \alpha . list(\alpha) \to integer \tag{6.12}$$

Символ $\forall$ представляет собой *квантор всеобщности* (universal qualifier), а переменная, к которой он применяется, называется связанной (bound) с ним. Такие переменные могут быть переименованы, если только переименованию подвергаются все вхождения данной переменной. Таким образом, выражение типа

$$\forall \beta . list(\beta) \to integer$$

эквивалентно выражению (6.12). О выражении типа с символом $\forall$ мы будем неформально говорить как о "полиморфном типе".

Всякий раз, когда применяется полиморфная функция, ее связанная переменная типа может описывать иной тип. В процессе проверки типов при каждом применении полиморфного типа мы заменяем связанную переменную новой переменной и убираем квантор общности. В следующем примере неформально выводится тип *length* с неявным применением правил вывода наподобие (6.9), которые повторены ниже:

if $f(x)$ является выражением,
then f имеет тип $\alpha \to \beta$ для некоторых α и β **and** x имеет тип α

Пример 6.15. Абстрактное синтаксическое дерево на рис. 6.29 представляет определение *length* на рис. 6.28. Корень дерева с меткой **fun** представляет определение функции. Прочие узлы, не являющиеся листьями, можно рассматривать как применения функции. Узел, помеченный +, представляет применение оператора + к паре дочерних узлов. Аналогично узел с меткой **if** представляет применение оператора **if** к тройке, образованной дочерними узлами (для проверки типов не имеет значения, что будет вычислена только одна из частей, **then** или **else**, но не обе).

Из тела функции *length* мы можем вывести ее тип. Рассмотрим узлы, дочерние по отношению к узлу **if**, слева направо. Поскольку *null* применяется к спискам, x должно быть таковым. Используем переменную α в качестве заменителя типа элемента списка, т.е. x имеет тип "список α".

Если $null(x)$ истинно, то $length(x)$ равно 0. Таким образом, тип *length* должен быть "функция от списка α, возвращающая целое число". Этот выведенный тип согласуется с использованием функции *length* в части **else**, $length(tl(x)) + 1$. □

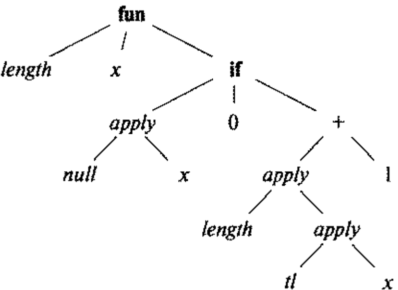


Рис. 6.29. Абстрактное синтаксическое дерево для определения функции на рис. 6.28

Подстановки, экземпляры и унификация

Если t представляет собой выражение типа, а S — подстановку (отображение переменной типа на выражение типа), то результат последовательной замены всех вхождений каждой переменной типа α в t на $S(\alpha)$ будем записывать как $S(t)$. $S(t)$ называется *экземпляром* t. Например, *list* (*integer*) представляет собой экземпляр *list* (α), поскольку это выражение представляет собой результат подстановки *integer* вместо α в *list* (α). Заметим, однако, что *integer* $\rightarrow$ *float* экземпляром $\alpha \rightarrow \alpha$ не является, поскольку подстановка должна заменять все вхождения α одним и тем же выражением типа.

Подстановка S является *унификатором* выражений типа t_1 и t_2, если $S(t_1) = S(t_2)$. S является *наиболее общим унификатором* t_1 и t_2, если для любого другого унификатора t_1 и t_2, скажем, S', $S'(t)$ является экземпляром $S(t)$. Другими словами, S' накладывает на t бóльшие ограничения, чем S.

Поскольку в выражениях типа могут участвовать переменные, мы должны пересмотреть понятие эквивалентности типов. Предположим, что E_1 типа $s \rightarrow s'$ применяется к E_2 типа t. Вместо простого определения эквивалентности s и t мы должны их "унифицировать". Неформально мы определяем, могут ли s и t быть сделаны структурно эквивалентными путем замены переменных типов в s и t выражениями типов.

Подстановка представляет собой отображение переменных типов на выражения типов. Мы записываем результат применения подстановки S к переменным в выражении типа t как $S(t)$; см. врезку "Подстановки, экземпляры и унификация". Два выражения типов t_1 и t_2 *унифицированы*, если существует некоторая подстановка S, такая, что $S(t_1) = S(t_2)$. На практике нас интересует наиболь-

ший общий унификатор, представляющий собой подстановку, которая налагает наименьшие ограничения на переменные в выражениях. См. алгоритм унификации в разделе 6.5.5.

Алгоритм 6.16. Выведение типа для полиморфных функций

ВХОД: программа, состоящая из последовательности определений функций, за которыми следуют вычисляемые выражения. Выражение состоит из применений функций и имен, где имена могут иметь предопределенные полиморфные типы.

ВЫХОД: выведенные типы для имен в программе.

МЕТОД: для простоты будем работать только с унарными функциями. Тип функции $f(x_1, x_2)$ с двумя параметрами может быть представлен выражением типа $s_1 \times \times s_2 \to t$, где s_1 и s_2 — типы x_1 и x_2 соответственно, а t — тип результата $f(x_1, x_2)$. Выражение $f(a, b)$ может быть проверено путем сопоставления типа a с s_1, а типа b — с s_2.

Проверяем определения функций и выражения во входной последовательности. Если функция впоследствии используется в выражении, используем выводимый тип функции.

- Для определения функции **fun** $\mathbf{id}_1\,(\mathbf{id}_2) = E$ создадим новые переменные типов α и β. Свяжем тип $\alpha \to \beta$ с функцией $\mathbf{id}_1$, а тип α — с параметром $\mathbf{id}_2$. Затем выведем тип для выражения E. Предположим, что после выведения типа E α обозначает тип s, а β обозначает тип t. Выведенный тип функции $\mathbf{id}_1$ — $s \to t$. Свяжем квантором $\forall$ все переменные типов, оставшиеся в $s \to t$ без ограничений.

- Дли применения функции $E_1\,(E_2)$ выведем типы E_1 и E_2. Поскольку E_1 используется в качестве функции, соответствующий тип должен иметь вид $s \to s'$. (Технически тип E_1 должен быть унифицирован с $\beta \to \gamma$, где β и γ — новые переменные типа.) Пусть t — выведенный тип E_2. Унифицируем s и t. Если унификация завершилась неудачей, выражение содержит ошибку типа. В противном случае выведенный тип $E_1\,(E_2)$ — s'.

- Для каждого вхождения полиморфной функции заменим в их типах связанные переменные различными новыми переменными и удалим кванторы $\forall$ Получившееся выражение типа является выведенным типом данного вхождения.

- Для впервые встречающегося имени введем новую переменную, обозначающую его тип. □

Пример 6.17. На рис. 6.30 выводится тип функции *length*. Корень синтаксического дерева на рис. 6.29 соответствует определению функции, так что мы вводим

переменные β и γ и связываем тип $\beta \to \gamma$ с функцией *length*, а тип β — с x (см. строки 1–2 на рис. 6.30).

Строка	Выражение : Тип	Унификация
1)	$length : \beta \to \gamma$	
2)	$x : \beta$	
3)	$\textbf{if} : boolean \times \alpha_i \times \alpha_i \to \alpha_i$	
4)	$null : list(\alpha_n) \to boolean$	
5)	$null(x) : boolean$	$list(\alpha_n) = \beta$
6)	$0 : integer$	$\alpha_i = integer$
7)	$+ : integer \times integer \to integer$	
8)	$tl : list(\alpha_t) \to list(\alpha_t)$	
9)	$tl(x) : list(\alpha_t)$	$list(\alpha_t) = list(\alpha_n)$
10)	$length(tl(x)) : \gamma$	$\gamma = integer$
11)	$1 : integer$	
12)	$length(tl(x)) + 1 : integer$	
13)	$\textbf{if}(\cdots) : integer$	

Рис. 6.30. Выведение типа для функции length на рис. 6.28

В правом дочернем узле корня мы рассматриваем **if** как полиморфную функцию, которая применяется к тройке, состоящей из логического значения и двух выражений, представляющих части **then** и **else**. Тип **if** — $\forall\alpha.boolean \times \alpha \times \alpha \to \alpha$.

Каждое применение полиморфной функции может иметь свой тип, так что мы вводим новую переменную α_i (где i указывает на **if**) и удаляем $\forall$ (см. строку 3 на рис. 6.30). Тип левого дочернего узла **if** должен быть унифицирован с *boolean*, а типы двух других дочерних узлов — с α_i.

Предопределенная функция *null* имеет тип $\forall\alpha.list(\alpha) \to boolean$. Мы используем новую переменную типа α_n (где n указывает на *null*) вместо связанной переменной α (см. строку 4). Из применения *null* к x мы выводим, что тип x β должен соответствовать $list(\alpha_n)$ (см. строку 5).

В первом дочернем узле **if** тип *boolean* для $null(x)$ соответствует типу, ожидаемому конструкцией **if**. Во втором дочернем узле тип α_i унифицируется с *integer* (см. строку 6).

Теперь рассмотрим подвыражение $length(tl(x)) + 1$. Мы создаем новую переменную α_t (где t указывает на tl) для связанной переменной α в типе tl (см. строку 8). Исходя из применения $tl(x)$ мы выводим $list(\alpha_t) = \beta = list(\alpha_n)$ (см. строку 9).

Поскольку *length* $(tl\,(x))$ является операндом +, его тип γ должен быть унифицирован с *integer* (см. строку 10). Отсюда следует, что тип *length* — *list* $(\alpha_n) \rightarrow$ *integer*. После проверки определения функции в типе *length* остается переменная типа α_n. Поскольку об α_n не было сделано никаких предположений, при применении функции вместо нее может быть подставлен любой тип. Поэтому мы делаем ее связанной переменной и записываем тип *length* как $\forall\alpha_n.list\,(\alpha_n) \rightarrow integer$. □

6.5.5 Алгоритм унификации

Неформально унификация представляет собой определение, можно ли выражения s и t сделать идентичными путем подстановки выражений вместо переменных в s и t. Проверка эквивалентности выражений является частным случаем унификации; если s и t содержат константы, но не переменные, то s и t унифицируются тогда и только тогда, когда они идентичны. Алгоритм унификации в этом разделе распространяется на графы с циклами, так что он может использоваться для проверки структурной эквивалентности циклических типов[7].

Мы реализуем графотеоретическую формулировку унификации, в которой типы представлены графами. Переменные типов представлены листьями, а конструкторы типов — внутренними узлами. Узлы группируются в классы эквивалентности; если два узла принадлежат одному и тому же классу эквивалентности, то выражения типов, которые они представляют, должны унифицироваться. Таким образом, все внутренние узлы одного и того же класса должны иметь один и тот же конструктор типа, а их соответствующие дочерние узлы должны быть эквивалентны.

Пример 6.18. Рассмотрим два выражения типа

$$((\alpha_1 \rightarrow \alpha_2) \times list\,(\alpha_3)) \rightarrow list\,(\alpha_2)$$
$$((\alpha_3 \rightarrow \alpha_4) \times list\,(\alpha_3)) \rightarrow \alpha_5$$

Приведенная далее подстановка S представляет собой наиболее общую унификацию для этих выражений:

x	$S(x)$
α_1	α_1
α_2	α_2
α_3	α_1
α_4	α_2
α_5	$list\,(\alpha_2)$

[7] В некоторых приложениях является ошибкой унификация переменной с выражением, содержащим эту переменную. Алгоритм 6.19 допускает такую подстановку.

Данная подстановка отображает два приведенных выше выражения типа на выражение

$$((\alpha_1 \to \alpha_2) \times list\,(\alpha_1)) \to list\,(\alpha_2)$$

Эти два выражения представлены двумя узлами с метками $\to$: 1 на рис. 6.31. Целые числа в узлах указывают классы эквивалентности, которым принадлежат узлы после унификации узлов с номером 1. □

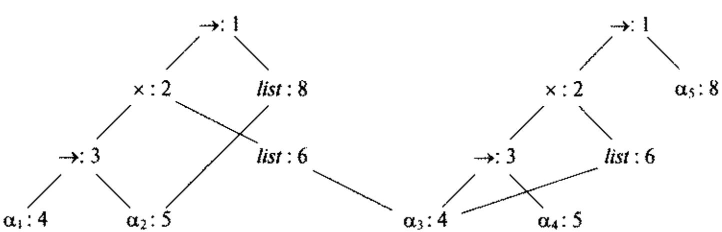


Рис. 6.31. Классы эквивалентности после унификации

Алгоритм 6.19. Унификация пар узлов в графе типа

ВХОД: граф, представляющий тип, и пара узлов m и n, которые должны быть унифицированы.

ВЫХОД: логическое значение true, если выражения, представленные узлами m и n, унифицированы; false в противном случае.

МЕТОД: узел реализуется записью с полями для бинарного оператора и указателей на левый и правый дочерние узлы. Множества эквивалентных узлов поддерживаются с использованием поля *set*. Один узел в каждом классе эквивалентности выбирается в качестве уникального представителя класса эквивалентности, что делается путем установки его поля *set* равным нулевому указателю. Поля *set* остальных узлов класса эквивалентности указывают (возможно, косвенно, через другие узлы множества) на представительный узел. Изначально каждый узел n принадлежит классу эквивалентности, состоящему из одного этого узла.

Алгоритм унификации, показанный на рис. 6.32, использует две следующие операции над узлами.

- *find* (n) возвращает представительный узел класса эквивалентности, содержащего узел n.
- *union* (m, n) объединяет классы эквивалентности, содержащие узлы m и n. Если один из представителей классов эквивалентности m и n является не переменной, то операция *union* делает представителем нового класса эквивалентности именно его; в противном случае операция делает представителем любого из представителей исходных классов. Эта асимметрия

```
boolean unify(Node m, Node n) {
    s = find(m); t = find(n);
    if ( s = t ) return true;
    else if ( Узлы s и t представляют один и тот же фундаментальный тип )
        return true;
    else if ( s является узлом операции с дочерними узлами s1 и s2,
                а t является узлом операции с дочерними узлами t1 и t2 ) {
        union(s, t);
        return unify(s1, t1) and unify(s2, t2);
    }
    else if (s или t представляет собой переменную) {
        union(s, t);
        return true;
    }
    else return false;
}
```

Рис. 6.32. Алгоритм унификации

операции *union* важна в связи с тем, что переменная не может использоваться как представитель класса эквивалентности для выражений, содержащих конструктор типа или фундаментальный тип: в противном случае два неэквивалентных выражения могут быть унифицированы посредством этой переменной.

Операция *union* над множествами реализуется простым изменением поля *set* представителя одного класса эквивалентности таким образом, чтобы оно указывало на представителя другого класса. Чтобы найти класс эквивалентности, которому принадлежит узел, надо проследовать по цепочке указателей *set* до достижения представителя (узла с нулевым указателем *set*).

Обратите внимание, что алгоритм на рис. 6.32 использует $s = find(m)$ и $t = find(n)$ вместо самих узлов m и n. Представительные узлы s и t равны, если m и n принадлежат одному классу эквивалентности. Если s и t представляют один и тот же фундаментальный тип, вызов $unify(s, t)$ вернет *true*. Если и s, и t являются внутренними узлами для бинарного конструктора типа, мы объединяем их классы эквивалентности и рекурсивно проверяем эквивалентность их соответствующих дочерних узлов. Поскольку сначала выполняется объединение, количество классов эквивалентности перед рекурсивной проверкой уменьшается, что гарантирует завершение работы алгоритма.

Подстановка выражения вместо переменной реализуется путем добавления листа переменной в класс эквивалентности, содержащий узел этого выражения.

Предположим, что либо m, либо n представляет собой лист для переменной. Предположим также, что этот лист помещен в класс эквивалентности с узлом, представляющим выражение с конструктором типа или фундаментальным типом. Тогда *find* вернет представительный узел, который отражает этот конструктор типа или фундаментальный тип, так что переменная не может быть унифицирована с двумя разными выражениями. □

Пример 6.20. Предположим, что два выражения в примере 6.18 изначально представлены графом на рис. 6.33, где каждый узел принадлежит собственному классу эквивалентности. При вычислении алгоритмом 6.19 *unify* $(1, 9)$ выясняется, что узлы 1 и 9 представляют один и тот же оператор. Следовательно, выполняется объединение 1 и 9 в один класс эквивалентности и вызываются *unify* $(2, 10)$ и *unify* $(8, 14)$. Результатом вычисления *unify* $(1, 9)$ оказывается граф, ранее показанный на рис. 6.31. □

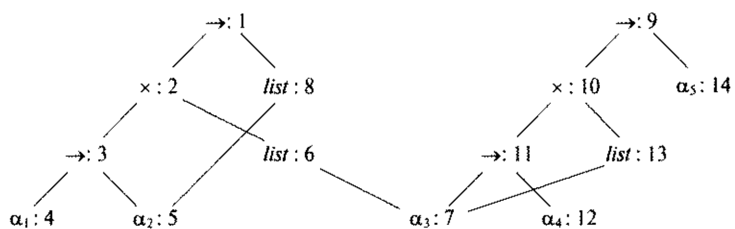


Рис. 6.33. Начальный граф, в котором каждый узел принадлежит собственному классу эквивалентности

Если алгоритм 6.19 возвращает *true*, подстановку S, которая выступает в роли унификатора, можно построить следующим образом. Для каждой переменной α *find* (α) дает узел n, который является представителем класса эквивалентности α. Выражение, представленное n, и есть $S(\alpha)$. Например, на рис. 6.31 видно, что представителем α_3 является узел 4, который представляет α_1. Представителем α_5 является узел 8, представляющий *list* (α_2). Окончательная подстановка приведена в примере 6.18.

6.5.6 Упражнения к разделу 6.5

Упражнение 6.5.1. В предположении, что функция *widen* на рис. 6.26 может работать с любым типом в иерархии на рис. 6.25, *а*, транслируйте приведенные ниже выражения (считаем, что `c` и `d` имеют тип `char`, `s` и `t` — `short`, `i` и `j` — `int`, а `x` — `float`):

а) `x = s + c`

б) `i = s + c`

в) `x = (s + c) * (t + d)`

Упражнение 6.5.2. Предположим, как в языке программирования Ada, что каждое выражение должно иметь единственный тип, но все, что можно вывести из подвыражений, — это множество возможных типов. Иначе говоря, применение функции E_1 к аргументу E_2, представленное продукцией $E \to E_1\,(E_2)$, имеет связанное с ней правило

$$E.type = \{t \mid \text{для некоторого } s \text{ из } E_2.type \ s \to t \text{ принадлежит } E_1.type\}$$

Опишите СУО, которое определяет единственный тип каждого подвыражения путем использования атрибута *type* для восходящего синтеза множества возможных типов и по определении единственного типа всего выражения выполняет нисходящее определение атрибута *unique* для типа каждого из подвыражений.

6.6 Поток управления

Трансляция таких инструкций, как конструкции if-else и while, связана с трансляцией булевых выражений. В языках программирования булевы выражения часто используются для следующих целей.

1. *Изменение потока управления.* Булевы выражения используются в качестве условных выражений в инструкциях, которые изменяют поток управления. Значения таких булевых выражений неявно присутствуют в позиции, достигнутой программой. Например, в **if** $(E)\,S$ выражение E должно быть равно *true*, если достигнута инструкция S.

2. *Вычисление логических значений.* Булево выражение может представлять *true* или *false* в качестве значений. Такие булевы выражения могут вычисляться по аналогии с арифметическими выражениями с использованием трехадресных команд с логическими операторами.

Запланированное использование булевых выражений определяется их синтаксическим контекстом. Например, выражение, следующее за ключевым словом **if**, используется для изменения потока управления, в то время как выражение в правой части присваивания используется для обозначения логического значения. Такой синтаксический контекст может определяться несколькими способами: можно использовать два разных нетерминала, воспользоваться наследуемыми атрибутами или устанавливать флаг в процессе синтаксического анализа. В качестве альтернативы можно построить синтаксическое дерево и вызывать различные процедуры для двух различных использований булевых выражений.

В этом разделе мы остановимся на использовании булевых выражений для изменения потока управления. Для ясности для этой цели мы введем новый нетерминал B. В разделе 6.6.6 будет рассмотрено, как компилятор может разрешить булевым выражениям представлять логические значения.

6.6.1 Булевы выражения

Булевы выражения составлены из булевых операторов (которые будут обозначаться &&, || и ! с помощью обозначений языка программирования C для операторов И, ИЛИ и НЕ соответственно), примененных к элементам, представляющим собой булевы переменные или выражения отношений. Выражение отношения имеет вид E_1 **rel** E_2, где E_1 и E_2 — арифметические выражения. В этом разделе будут рассматриваться булевы выражения, генерируемые грамматикой

$$B \rightarrow B || B \mid B \;\&\&\; B \mid !B \mid (B) \mid E \; \mathbf{rel} \; E \mid \mathbf{true} \mid \mathbf{false}$$

Для указания, какой из шести операторов сравнения (<, <=, =, !=, > или >=) представлен при помощи **rel**, будет использоваться атрибут **rel**.*op*. По привычке мы полагаем, что операторы || и && левоассоциативны и что у оператора || приоритет меньше, чем у оператора &&, приоритет которого, в свою очередь, меньше приоритета оператора !.

Если для данного выражения $B_1 || B_2$ определено, что B_1 равно *true*, то можно заключить без вычисления B_2, что все выражение равно *true*. Аналогично, если в B_1 && B_2 B_1 равно *false*, то значение всего выражения также равно *false*.

Семантическое определение языка программирования указывает, должны ли вычисляться все части булева выражения. Если определение языка программирования допускает (или требует) оставить часть логического выражения не вычисленной, то компилятор может оптимизировать вычисления булевых выражений путем вычисления только той части, которая достаточна для определения значения выражения в целом. Таким образом, в выражении наподобие $B_1 || B_2$ ни B_1, ни B_2 не обязаны вычисляться полностью. Если либо B_1, либо B_2 представляет собой выражение с побочным действием (например, содержит вызов функции, которая изменяет глобальную переменную), то может быть получен не тот результат, которого вы ожидаете.

6.6.2 Код сокращенного вычисления

При *сокращенных* (short-circuit) вычислениях булевы операторы &&, || и ! транслируются в переходы. Сами операторы в коде отсутствуют; вместо этого значение булева выражения представлено в виде позиции в последовательности команд.

Пример 6.21. Инструкция

```
if ( x < 100 || x > 200 && x != y ) x = 0;
```

может быть транслирована в код, приведенный на рис. 6.34. В этой трансляции булево выражение истинно, если управление достигает метки L_2. Если выражение ложно, управление передается непосредственно к метке L_1, минуя L_2 и присваивание `x = 0`. □

```
      if x < 100 goto L2
      ifFalse x > 200 goto L1
      ifFalse x != y goto L1
L2:   x = 0
L1:
```

Рис. 6.34. Код сокращенных вычислений

6.6.3 Инструкции потока управления

Рассмотрим теперь трансляцию булевых выражений в трехадресный код в контексте таких инструкций, как генерируемые следующей грамматикой:

$$
\begin{array}{lcl}
S & \rightarrow & \textbf{if}\ (B)\ S_1 \\
S & \rightarrow & \textbf{if}\ (B)\ S_1\ \textbf{else}\ S_2 \\
S & \rightarrow & \textbf{while}\ (B)\ S_1
\end{array}
$$

В этих продукциях нетерминал B представляет булево выражение, а нетерминал S – инструкцию.

Приведенная грамматика обобщает рабочий пример выражения **while** из примера 5.19. В этом примере и B, и S имели синтезируемый атрибут *code*, который давал трансляцию в трехадресные команды. Для простоты мы строим трансляции $B.code$ и $S.code$ с использованием синтаксически управляемых определений в виде строк. Семантические правила, определяющие атрибуты *code*, могут быть реализованы также посредством построения синтаксических деревьев и генерацией кода в процессе обхода дерева, а также с использованием иных методов, рассмотренных в разделе 5.5.

Трансляция **if** (B) S_1 состоит из $B.code$, за которым следует $S_1.code$, как показано на рис. 6.35, *а*. В $B.code$ имеются переходы, основанные на значении B. Если B истинно, управление переходит к первой команде $S_1.code$, а если B ложно, то управление переходит к команде, следующей непосредственно за $S_1.code$.

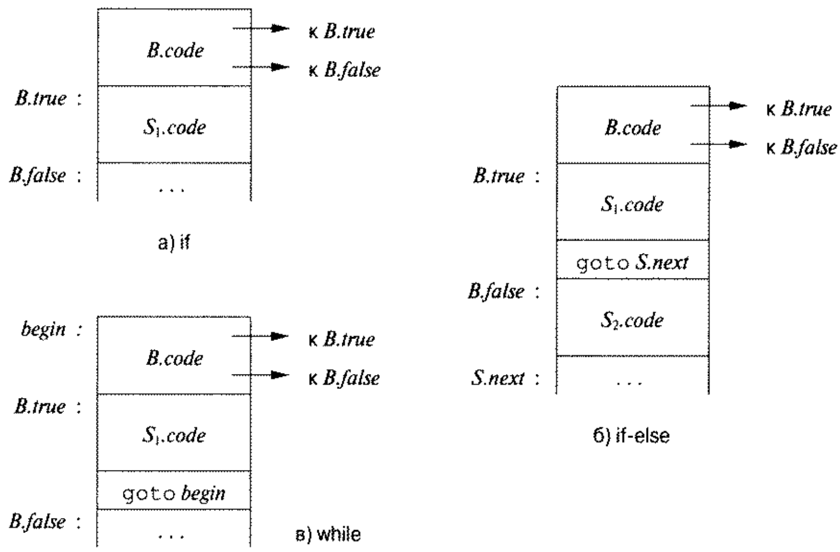


Рис. 6.35. Коды для инструкций if, if-else и while

Работа с метками для переходов в *B.code* и *S.code* выполняется с использованием наследуемых атрибутов. С булевым выражением B связаны две метки: *B.true*, метка, которой передается управление в случае истинности B, и *B.false*, метка, управление которой передается в случае ложности B. Что касается инструкции S, то с ней связана метка *S.next*, обозначающая команду, следующую непосредственно за кодом S. В некоторых случаях командой, непосредственно следующей за *S.code*, оказывается команда перехода к некоторой метке L. Перехода к переходу к метке L из кода *S.code* можно избежать, используя *S.next*.

Синтаксически управляемое определение на рис. 6.36 и 6.37 генерирует трехадресный код для булевых выражений в контексте **if**, **if-else** и **while** инструкций.

Мы считаем, что каждый вызов *newlabel* () создает новую метку и что вызов *label* (L) назначает метку L очередной генерируемой трехадресной команде[8].

Программа состоит из инструкции, генерируемой продукцией $P \to S$. Семантические правила, связанные с этой продукцией, инициализируют атрибут *S.next* новой меткой. *P.code* состоит из *S.code*, за которым следует новая метка *S.next*. Токен **assign** в продукции $S \to$ **assign** представляет собой "заполнитель" для инструкций присваивания. Трансляция присваиваний выполняется так, как рас-

[8]В случае буквальной реализации семантические правила будут генерировать множество меток и могут назначать одной трехадресной команде больше одной метки. Метод обратных поправок из раздела 6.7 создает метки только тогда, когда это необходимо. Другой подход может состоять в удалении лишних меток на стадии оптимизации.

Продукция	Семантические правила
$P \to S$	$S.next = newlabel()$ $P.code = S.code \parallel label(S.next)$
$S \to$ **assign**	$S.code =$ **assign**$.code$
$S \to$ **if** $(B) S_1$	$B.true = newlabel()$ $B.false = S_1.next = S.next$ $S.code = B.code \parallel label(B.true) \parallel S_1.code$
$S \to$ **if** $(B) S_1$ **else** S_2	$B.true = newlabel()$ $B.false = newlabel()$ $S_1.next = S_2.next = S.next$ $S.code = B.code$ $\parallel label(B.true) \parallel S_1.code$ $\parallel gen(\text{'goto'}\ S.next)$ $\parallel label(B.false) \parallel S_2.code$
$S \to$ **while** $(B) S_1$	$begin = newlabel()$ $B.true = newlabel()$ $B.false = S.next$ $S_1.next = begin$ $S.code = label(begin) \parallel B.code$ $\parallel label(B.true) \parallel S_1.code$ $\parallel gen(\text{'goto'}\ begin)$
$S \to S_1\ S_2$	$S_1.next = newlabel()$ $S_2.next = S.next$ $S.code = S_1.code \parallel label(S_1.next) \parallel S_2.code$

Рис. 6.36. Синтаксически управляемое определение для инструкций потока управления

сматривалось в разделе 6.4; здесь мы рассматриваем поток управления, так что просто полагаем $S.code$ равным **assign**$.code$.

При трансляции $S \rightarrow \textbf{if}\ (B)\ S_1$ семантические правила, показанные на рис. 6.36, создают новую метку *B.true* и назначают ее первой трехадресной команде, генерируемой для инструкции S_1, как показано на рис. 6.35, *а*. Таким образом, переход к *B.true* в коде B приводит к переходу к коду S_1. Далее, путем присваивания атрибуту *B.false* значения *S.next* мы гарантируем, что в случае, если вычисление B даст значение *false*, код S_1 будет пропущен.

При трансляции $S \rightarrow \textbf{if}\ (B)\, S_1\ \textbf{else}\ S_2$ код для булева выражения B содержит переходы к первой команде кода S_1 при истинном B и к первой команде кода S_2 при ложном B, как показано на рис. 6.35, *б*. Затем управление как из S_1, так и из S_2 передается трехадресной команде, непосредственно следующей за кодом S, – ее метка хранится в наследуемом атрибуте *S.next*. За кодом S_1 следует явная команда безусловного перехода *goto S.next*, которая позволяет пропустить код S_2. После S_2 команда безусловного перехода не нужна, поскольку S_2*.next* совпадает с *S.next*.

Код для $S \rightarrow \textbf{while}\ (B)\ S_1$ образуется из *B.code* и S_1*.code* так, как показано на рис. 6.35, *в*. Мы используем локальную переменную *begin* для хранения новой метки, назначенной первой команде инструкции **while**, которая одновременно является первой командой B. Мы используем переменную, а не атрибут, поскольку переменная *begin* локальна для семантических правил данной продукции. Наследуемая метка *S.next* маркирует команду, управление которой передается в случае, когда B ложно; следовательно, *B.false* устанавливается равным *S.next*. Новая метка *B.true* назначается первой команде S_1; код для B генерирует переход к этой метке, если B истинно. После кода S_1 помещается команда безусловного перехода `goto` *begin*, которая передает управление в начало кода булева выражения. Обратите внимание, что для того, чтобы переходы из кода S_1*.code* выполнялись к метке *begin*, значение S_1*.next* устанавливается равным *begin*.

Код для продукции $S \rightarrow S_1\ S_2$ состоит из кода S_1, за которым следует код S_2. Семантические правила управляют метками; первой командой после кода S_1 является начальная команда S_2, а командой после кода S_2 является команда, следующая за кодом S.

В разделе 6.7 мы продолжим рассмотрение инструкций потока управления и познакомимся с методом "обратных поправок" (backpatching), который позволяет сгенерировать код инструкции за один проход.

6.6.4 Трансляция логических выражений с помощью потока управления

Семантические правила для булевых выражений на рис. 6.37 дополняют семантические правила для инструкций на рис. 6.36. Как и в коде на рис. 6.35, булево выражение B транслируется в трехадресные команды, которые вычисляют

B с использованием условных и безусловных переходов к одной из двух меток: $B.true$, если B истинно, и $B.false$, если B ложно.

Продукция	Семантические правила
$B \rightarrow B_1 \ \vert\vert \ B_2$	$B_1.true = B.true$ $B_1.false = newlabel()$ $B_2.true = B.true$ $B_2.false = B.false$ $B.code = B_1.code \ \Vert \ label(B_1.false) \ \Vert \ B_2.code$
$B \rightarrow B_1 \ \&\& \ B_2$	$B_1.true = newlabel()$ $B_1.false = B.false$ $B_2.true = B.true$ $B_2.false = B.false$ $B.code = B_1.code \ \Vert \ label(B_1.true) \ \Vert \ B_2.code$
$B \rightarrow \ ! \ B_1$	$B_1.true = B.false$ $B_1.false = B.true$ $B.code = B_1.code$
$B \rightarrow E_1 \ \mathbf{rel} \ E_2$	$B.code = E_1.code \ \Vert \ E_2.code$ $\Vert \ gen('\texttt{if}' \ E_1.addr \ \mathbf{rel}.op \ E_2.addr \ '\texttt{goto}' \ B.true)$ $\Vert \ gen('\texttt{goto}' \ B.false)$
$B \rightarrow \mathbf{true}$	$B.code = gen('\texttt{goto}' \ B.true)$
$B \rightarrow \mathbf{false}$	$B.code = gen('\texttt{goto}' \ B.false)$

Рис. 6.37. Генерация трехадресного кода для булевых выражений

Четвертая продукция на рис. 6.37, $B \rightarrow E_1$ **rel** E_2, транслируется непосредственно в трехадресную команду сравнения с переходами к соответствующим меткам. Например, B вида $a < b$ транслируется в

$$\texttt{if a < b goto } B.true$$
$$\texttt{goto } B.false$$

Остальные продукции B транслируются следующим образом.

1. Предположим, что B имеет вид $B_1 \| B_2$. Если B_1 истинно, то мы знаем, что B также истинно, так что $B_1.true$ равно $B.true$. Если же B_1 ложно, то должно быть вычислено B_2, так что $B_1.false$ должно быть меткой первой команды кода B_2. Истинный и ложный выходы B_2 те же, что и для B.

2. Трансляция B_1 && B_2 аналогична.

3. Для вычисления B вида $!B_1$ не требуется никакого кода: просто истинный и ложный выходы B становятся соответственно ложным и истинным выходами B_1.

4. Константы **true** и **false** транслируются в переходы к $B.true$ и $B.false$ соответственно.

Пример 6.22. Вновь обратимся к выражению из примера 6.21:

```
if ( x < 100 || x > 200 && x != y ) x = 0;
```
(6.13)

Используя синтаксически управляемые определения, показанные на рис. 6.36, и рис. 6.37, мы получаем код, приведенный на рис. 6.38.

```
      if x < 100 goto L2
      goto L3
L3:   if x > 200 goto L4
      goto L1
L4:   if x != y goto L2
      goto L1
L2:   x = 0
L1
```

Рис. 6.38. Трансляция простой инструкции if с помощью потока управления

Инструкция (6.13) составляет программу, генерируемую продукцией $P \to S$ на рис. 6.36. Семантические правила для продукции генерируют новую метку L_1 для команды после кода S. Инструкция S имеет вид **if** (B) S_1, где S_1 представляет собой `x = 0;`, так что правила на рис. 6.36 генерируют новую метку L_2 и назначают ее первой (и единственной в данном случае) команде в $S_1.code$, которая имеет вид `x = 0`.

Поскольку оператор || имеет меньший приоритет, чем &&, булево выражение в (6.13) имеет вид $B_1 \| B_2$, где B_1 представляет собой `x < 100`. В соответствии с правилами на рис. 6.37, $B_1.true$ равно L_2, метке присваивания `x = 0;`. $B_1.false$ представляет собой новую метку L_3, назначенную первой команде кода B_2.

Заметим, что сгенерированный код не оптимален и содержит на три команды безусловного перехода больше, чем код из примера 6.21. Команда *goto* L_3 излишня, поскольку L_3 представляет собой метку команды, следующей за командой безусловного перехода. Две команды *goto* L_1 могут быть устранены, если вместо команды `if` использовать команду `ifFalse`, как это сделано в примере 6.21. □

6.6.5 Устранение излишних команд перехода

В примере 6.22 сравнение `x > 200` транслируется в следующий фрагмент кода:

```
        if x > 200 goto L4
        goto L1
L4:     ...
```

Рассмотрим вместо этого команду

```
        ifFalse x > 200 goto L1
L4:     ...
```

Такая команда `ifFalse` использует преимущества естественного потока от одной команды в последовательности к другой, так что управление просто "проваливается" к метке L_4, если $x > 200$, тем самым позволяя избежать явного использования команды безусловного перехода.

В схеме кода для конструкций **if** и **while** на рис. 6.35 код для инструкции S_1 следует непосредственно за кодом булева выражения B. При помощи специальной метки *fall* ("не генерировать никакой команды перехода") можно так адаптировать семантические правила, представленные на рис. 6.36 и 6.37, чтобы управление переходило сквозь код B к коду S_1. Новые правила для продукции $S \to$ **if** (B) S_1 на рис. 6.36 устанавливают *B.true* равным *fall*:

$$B.true = fall$$
$$B.false = S_1.next = S.next$$
$$S.code = B.code \| S_1.code$$

Аналогично устанавливают *B.true* равным *fall* и правила для инструкций if-else и while.

Можно адаптировать семантические правила для булевых выражений таким образом, чтобы позволить управлению "проваливаться" там, где это возможно. Новые правила для продукции $B \to E_1$ **rel** E_2 на рис. 6.39 генерируют две команды, как на рис. 6.37, если *B.true* и *B.false* являются явными метками, т.е. ни одна из них не равна *fall*. В противном случае, если *B.true* представляет собой явную метку, метка *B.false* должна быть равна *fall*, так что она генерирует команду `if`,

которая позволяет управлению "провалиться", если условие ложно. И наоборот, если явную метку содержит $B.false$, то она генерирует команду `ifFalse`. В оставшемся случае, когда и $B.true$, и $B.false$ равны $fall$, никаких команд перехода не генерируется.[9]

$$
\begin{aligned}
& test = E_1.addr\ \mathbf{rel}.op\ E_2.addr \\
& \quad s = \mathbf{if}\ B.true \neq fall\ \mathbf{and}\ B.false \neq fall\ \mathbf{then} \\
& \qquad\qquad gen\,('if'\ test\ '\texttt{goto}'\ B.true)\,\|gen\,('\texttt{goto}'\ B.true) \\
& \qquad \mathbf{else\ if}\ B.true \neq fall\ \mathbf{then}\ gen\,('\texttt{if}'\ test\ '\texttt{goto}'\ B.true) \\
& \qquad \mathbf{else\ if}\ B.false \neq fall\ \mathbf{then}\ gen\,('\texttt{ifFalse}'\ test\ '\texttt{goto}'\ B.false) \\
& \qquad \mathbf{else}\ '\ \ ' \\
& B.code = E_1.code\|E_2.code\|s
\end{aligned}
$$

Рис. 6.39. Семантические правила для $B \to E_1$ **rel** E_2

В новых правилах для $B \to B_1 \| B_2$ на рис. 6.40 обратите внимание на то, что смысл метки $fall$ для B отличается от смысла для B_1. Предположим, что $B.true$ равно $fall$, т.е. управление "проваливается" через B, если вычисленное значение B равно $true$. Хотя значение B равно $true$, если это же значение принимает B_1, метка $B_1.true$ должна гарантировать переход управления через код B_2 к команде, следующей за кодом B.

$$
\begin{aligned}
& B_1.true = \mathbf{if}\ B.true \neq fall\ \mathbf{then}\ B.true\ \mathbf{else}\ newlabel\,() \\
& B_1.false = fall \\
& B_2.true = B.true \\
& B_2.false = B.false \\
& B.code = \mathbf{if}\ B.true \neq fall\ \mathbf{then}\ B_1.code\|B_2.code \\
& \qquad\qquad \mathbf{else}\ B_1.code\|B_2.code\|label\,(B_1.true)
\end{aligned}
$$

Рис. 6.40. Семантические правила для $B \to B_1 \| B_2$

С другой стороны, если вычисление B_1 дает $false$, то истинность значения B определяется значением B_2, так что правила на рис. 6.40 гарантируют, что $B_1.false$ соответствует "проваливанию" управления от B_1 к коду B_2.

Семантические правила для $B \to B_1$ && B_2 аналогичны правилам, приведенным на рис. 6.40, и оставлены читателю в качестве упражнения.

[9] В C и Java выражения могут включать в себя присваивания, так что код для подвыражений E_1 и E_2 должен быть сгенерирован, даже если и $B.true$, и $B.false$ равны $fall$. Если требуется, "мертвый" код может быть удален на стадии оптимизации.

Пример 6.23. При новых правилах, использующих специальную метку *fall*, программа (6.13) из примера 6.21

```
if ( x < 100 || x > 200 && x != y ) x = 0;
```

транслируется в код, приведенный на рис. 6.41.

```
      if x < 100 goto L2
      ifFalse x > 200 goto L1
      ifFalse x != y goto L1
L2:   x = 0
L1:
```

Рис. 6.41. Трансляция инструкции `if` с применением метода "проваливания"

Как и в примере 6.22, правила для продукции $P \to S$ создают метку L_1. Отличие от примера 6.22 заключается в том, что наследуемый атрибут $B.true$ равен *fall* при применении семантических правил для $B \to B_1 \parallel B_2$ ($B.false$ равно L_1). Правила на рис. 6.40 создают новую метку L_2, позволяющую выполнить переход через код B_2, если вычисление B_1 дает значение *true*. Таким образом, $B_1.true$ равно L_2, а $B_1.false$ равно *fall*, поскольку, если B_1 равно *false*, должно вычисляться B_2.

Продукция $B \to E_1$ **rel** E_2, которая генерирует $x < 100$, достигается, таким образом, с $B.true = L_2$ и $B.false = fall$. С этими наследуемыми метками правила на рис. 6.39 генерируют одну команду `if x < 100 goto L2`. □

6.6.6 Булевы значения и код с переходами

В этом разделе мы остановимся на использовании булевых выражений для изменения потока управления в инструкциях. Булево выражение может также вычисляться для получения его значения, как в инструкциях присваивания наподобие `x = true;` и `x = a<b;`.

Очевидный способ работы с обеими ролями булевых выражений состоит в построении синтаксического дерева для выражений с использованием одного из следующих подходов.

1. *Использование двух проходов.* Строим полное синтаксическое дерево для входных данных, а затем выполняем его обход в глубину, вычисляя трансляции, определяемые семантическими правилами.

2. *Использование одного прохода для инструкций и двух — для выражений.* При таком подходе в **while** (E) S_1 сначала транслируется E, а затем рассматривается S_1. Трансляция же E выполняется путем построения синтаксического дерева с последующим его обходом.

Приведенная далее грамматика использует для выражений один нетерминал E:

$$S \rightarrow \textbf{id} = E\ ;\ |\ \textbf{if}\ (E)\ S\ |\ \textbf{while}\ (E)\ S\ |\ S\ S$$
$$E \rightarrow E \,\|\, E\ |\ E\ \&\&\ E\ |\ E\ \textbf{rel}\ E\ |\ E + E\ |\ (E)\ |\ \textbf{id}\ |\ \textbf{true}\ |\ \textbf{false}$$

Нетерминал E регулирует поток управления в $S \rightarrow \textbf{while}\ (E)\ S_1$; тот же нетерминал E является значением в $S \rightarrow \textbf{id} = E\ ;$ и $E \rightarrow E + E$.

Справиться с этими двумя ролями выражений можно при помощи отдельных функций генерации кода. Предположим, что атрибут $E.n$ обозначает узел синтаксического дерева для выражения E и что узлы представляют собой объекты. Пусть метод *перехода* (jump) генерирует в узле выражения код с переходами, а метод *r-значения* (rvalue) генерирует код для вычисления значения узла во временную переменную.

Когда E встречается в $S \rightarrow \textbf{while}\ (E)\ S_1$, в узле $E.n$ вызывается метод *jump*. Реализация перехода основана на правилах для булевых выражений, представленных на рис. 6.37. В частности, код с переходами генерируется путем вызова $E.n.jump\,(t, f)$, где t — новая метка для первой команды $S_1.code$, а f — метка $S.next$.

Если же E встречается в $S \rightarrow \textbf{id}\ =\ E\ ;$, в узле $E.n$ используется метод *rvalue*. Если E имеет вид $E_1 + E_2$, вызывается метод $E.n.rvalue\,()$, генерирующий код так, как описано в разделе 6.4. Если E имеет вид $E_1\ \&\&\ E_2$, то сначала генерируется код с переходами для E и новой временной переменной `t` присваивается значение *true* или *false* в соответствующих выходах из кода с переходами.

Например, присваивание `x = a<b && c<d` может быть реализовано кодом, представленным на рис. 6.42.

```
     ifFalse a < b goto L1
     ifFalse c < d goto L1
     t = true
     goto L2
L1:  t = false
L2:  x = t
```

Рис. 6.42. Трансляция булева присваивания

6.6.7 Упражнения к разделу 6.6

Упражнение 6.6.1. Добавьте правила к синтаксически управляемому определению на рис. 6.36 для следующих конструкций:

а) **repeat** S **while** B

! б) **for** $(S_1; B; S_2)\ S_3$

Упражнение 6.6.2. Современные машины пытаются выполнять одновременно несколько команд, включая команды ветвления. При этом на производительности крайне отрицательно сказываются ситуации, когда машина предполагает, что ветвление пойдет по одному пути, в то время как на самом деле оно идет другим путем (вся работа оказывается выполненной зря). Следовательно, желательно минимизировать количество ветвлений. Заметим, что реализация цикла **while** на рис. 6.35, *в* имеет два ветвления на итерацию: одно — для входа в тело цикла из условия B, второе — для возврата к коду B. В результате обычно более предпочтительной является реализация **while** $(B)\ S$, как если бы это была конструкция **if** (B) $\{$**repeat** S **until** $!\,(B)\}$. Приведите схему кода для такой трансляции и измените соответствующим образом правила на рис. 6.36.

! **Упражнение 6.6.3.** Предположим наличие оператора "исключающего ИЛИ" (который возвращает значение *true* тогда и только тогда, когда ровно один из двух его аргументов равен *true*). Напишите для такого оператора правила в стиле рис. 6.37.

Упражнение 6.6.4. Транслируйте следующие выражения с использованием схемы трансляции с устранением переходов из раздела 6.6.5:

а) `if (a==b && c==d || e==f) x == 1;`

б) `if (a==b || c==d || e==f) x == 1;`

в) `if (a==b && c==d && e==f) x == 1;`

Упражнение 6.6.5. Разработайте схему трансляции на основе синтаксически управляемых определений, приведенных на рис. 6.36 и 6.37.

Упражнение 6.6.6. Адаптируйте семантические правила, представленные на рис. 6.36 и 6.37, так, чтобы они допускали "проваливание" управления, используя правила, аналогичные представленным на рис. 6.39 и 6.40.

! **Упражнение 6.6.7.** Семантические правила для инструкций в упражнении 6.6.6 генерируют излишние метки. Модифицируйте правила, представленные на рис. 6.36, так, чтобы метки создавались по мере необходимости, с использованием специальной метки *deferred*, означающей, что метка пока что не была создана. Ваши правила должны генерировать код, подобный коду из примера 6.21.

!! **Упражнение 6.6.8.** В разделе 6.6.5 говорилось об использовании кода с "проваливанием" для минимизации переходов в генерируемом промежуточном коде. Однако при этом не использовались возможности замены условий на обратные, например замены `if a < b goto` L_1`; goto` L_2 фрагментом `if a >=`

`b goto L2; goto L1`. Разработайте СУО, которое при необходимости использует преимущества такой замены.

6.7 Обратные поправки

Ключевая проблема при генерации кода для булевых выражений и инструкций потока управления заключается в соответствии команд перехода целевым меткам. Например, трансляция булева выражения B в `if (`B`)` S содержит переход в случае ложного значения B к команде, следующей за кодом S. При однопроходной трансляции B должно быть транслировано до того, как будет рассматриваться S. Что в таком случае должно быть указано в качестве целевой метки команды `goto`, которая выполняет переход через весь код S? В разделе 6.6 мы решали эту задачу путем передачи меток в качестве наследуемых атрибутов в место, где генерируются соответствующие команды переходов. Однако в таком случае для связывания меток с адресами требуется дополнительный проход.

В этом разделе рассматривается дополнительный метод *обратных поправок* (backpatching), в котором списки переходов передаются как синтезируемые атрибуты. В частности, при генерации перехода целевая метка временно остается неизвестной. Каждый такой переход помещается в список переходов, метки которых будут указаны после того, как эти метки смогут быть определены. Все переходы в одном списке имеют одну и ту же целевую метку.

6.7.1 Однопроходная генерация кода с использованием обратных поправок

Метод обратных поправок может использоваться для генерации кода булевых выражений и инструкций потока управления за один проход. Генерируемые нами трансляции будут иметь тот же вид, что и в разделе 6.6, за исключением работы с метками.

В этом разделе для работы с метками кода с переходами для булевых выражений используются синтезируемые атрибуты *truelist* и *falselist*. В частности, *B.truelist* представляет собой список команд условного и безусловного перехода, в которые должна быть вставлена метка, которой передается управление при истинности B. Аналогично *B.falselist* представляет собой список команд перехода, которые в конечном итоге получат метку для передачи управления при ложности B. После генерации кода B переходы к истинному и ложному выходам остаются незавершенными, с незаполненными полями меток. Эти незавершенные команды перехода помещаются в списки, на которые указывают *B.truelist* и *B.falselist*. Аналогично инструкция S имеет синтезируемый атрибут *S.nextlist*, представляющий собой список переходов к команде, идущей непосредственно за кодом S.

Для определенности мы генерируем команды в виде массива команд, и метки представляют собой индексы этого массива. При работе со списками переходов используются три функции.

1. *makelist* (i) создает новый список, состоящий только из i, индекса команды в массиве. Функция возвращает указатель на вновь созданный список.

2. *merge* (p_1, p_2) объединяет списки, на которые указывают p_1 и p_2, и возвращает указатель на объединенный список.

3. *backpatch* (p, i) вставляет i в качестве целевой метки в каждую команду списка, на который указывает p.

6.7.2 Обратные поправки для булевых выражений

Теперь построим схему трансляции, пригодную для генерации кода для булевых выражений в процессе восходящего синтаксического анализа. Маркер-нетерминал M заставляет семантическое действие получить в необходимый момент индекс очередной генерируемой команды. Грамматика выглядит следующим образом:

$$\begin{array}{lcl} B & \rightarrow & B_1 \| M \; B_2 \mid B_1 \;\&\&\; M \; B_2 \mid !B_1 \mid (B_1) \mid E_1 \; \textbf{rel} \; E_2 \mid \textbf{true} \mid \textbf{false} \\ M & \rightarrow & \epsilon \end{array}$$

Схема трансляции представлена на рис. 6.43.

Рассмотрим семантическое действие 1 продукции $B \rightarrow B_1 \| M \; B_2$. Если B_1 истинно, то истинно также и значение B в целом, так что переходы B_1.*truelist* становятся частью списка B.*truelist*. Если B_1 ложно, следует выполнить проверку B_2, так что целевой меткой переходов B_1.*falselist* должно быть начало кода B_2. Эта целевая метка получается с помощью маркера-нетерминала M. Этот нетерминал в качестве синтезируемого атрибута M.*instr* дает индекс очередной команды непосредственно перед тем, как начнется генерация кода для B_2.

Чтобы получить этот индекс, мы связываем с продукцией $M \rightarrow \epsilon$ семантическое действие $\{M.instr = nextinstr;\}$. Переменная *nextinstr* хранит индекс очередной команды. Это значение будет вставлено в команды перехода из B_1.*falselist* (т.е. каждая команда из списка B_1.*falselist* получит в качестве целевой метки значение M.*instr*), когда мы достигнем конца продукции $B \rightarrow B_1 \| M \; B_2$.

Семантическое действие 2 продукции $B \rightarrow B_1 \&\& M \; B_2$ аналогично действию 1. Действие 3 для $B \rightarrow !B_1$ меняет местами списки для ложного и истинного значений. Действие 4 просто игнорирует скобки.

Для простоты семантическое действие 5 генерирует две команды, условного и безусловного переходов. Ни у одной из них не заполняется поле целевой метки. Эти команды помещаются в новые списки, на которые указывают B.*truelist* и B.*falselist* соответственно.

1)	$B \to B_1$ `||` $M\ B_2$	{ $backpatch(B_1.falselist, M.instr)$; $B.truelist = merge(B_1.truelist, B_2.truelist)$; $B.falselist = B_2.falselist$; }
2)	$B \to B_1$ && $M\ B_2$	{ $backpatch(B_1.truelist, M.instr)$; $B.truelist = B_2.truelist$; $B.falselist = merge(B_1.falselist, B_2.falselist)$; }
3)	$B \to\ !\ B_1$	{ $B.truelist = B_1.falselist$; $B.falselist = B_1.truelist$; }
4)	$B \to (\ B_1\ )$	{ $B.truelist = B_1.truelist$; $B.falselist = B_1.falselist$; }
5)	$B \to E_1$ **rel** E_2	{ $B.truelist = makelist(nextinstr)$; $B.falselist = makelist(nextinstr + 1)$; *emit*('`if`' $E_1.addr$ **rel**.*op* $E_2.addr$ '`goto _`'); *emit*('`goto _`'); }
6)	$B \to$ **true**	{ $B.truelist = makelist(nextinstr)$; *emit*('`goto _`'); }
7)	$B \to$ **false**	{ $B.falselist = makelist(nextinstr)$; *emit*('`goto _`'); }
8)	$M \to \epsilon$	{ $M.instr = nextinstr$; }

Рис. 6.43. Схема трансляции для булевых выражений

Пример 6.24. Вновь рассмотрим выражение

```
x < 100 || x > 200 && x != y
```

Аннотированное дерево разбора для этого выражения показано на рис. 6.44. Для удобочитаемости атрибуты *truelist*, *falselist* и *instr* представлены их начальными буквами. Действия выполняются в процессе обхода дерева в глубину. Поскольку все действия находятся на правых концах продукций, они могут выполняться вместе со свертками в процессе восходящего разбора. При свертке `x < 100` в B в соответствии с продукцией 5 генерируются две команды:

```
100 : if x < 100 goto _
101 : goto _
```

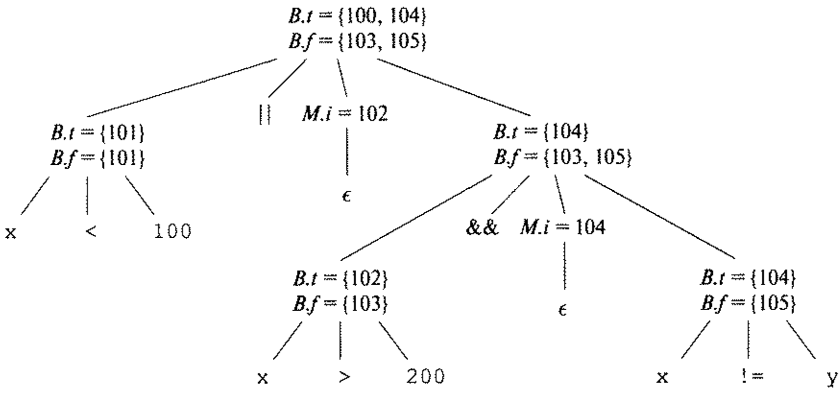


Рис. 6.44. Аннотированное дерево разбора для $x < 100 \parallel x > 200 \&\& x! = y$

(Команды нумеруются начиная с произвольно выбранного индекса 100.) Маркер-нетерминал M в продукции $B \to B_1 \parallel M\ B_2$ записывает значение *nextinstr*, которое в настоящий момент равно 102. Свертка `x > 200` в B в соответствии с продукцией 5 генерирует команды

```
102 : if x > 200 goto _
103 : goto _
```

Подвыражение `x > 200` соответствует B_1 в продукции $B \to B_1\ \&\&\ M\ B_2$. Маркер-нетерминал M записывает текущее значение *nextinstr*, которое в этот момент равно 104. Свертка `x != y` в B в соответствии с продукцией 5 генерирует

```
104 : if x!=y goto _
105 : goto _
```

Теперь можно выполнить свертку в соответствии с продукцией $B \to B_1\ \&\&\ M\ B_2$. Соответствующее семантическое действие вызывает *backpatch*(B_1.*truelist*, M.*instr*) для связывания выхода из B_1 по значению "истинно" с первой командой B_2. Поскольку список B_1.*truelist* равен $\{102\}$, а M.*instr* равно 104, этот вызов *backpatch* вносит метку 104 в команду 102. Сгенерированные таким образом шесть команд показаны на рис. 6.45, *а*.

Семантическое действие, связанное с последней сверткой в соответствии с $B \to B_1 \parallel M\ B_2$, вызывает *backpatch* $(\{101\}, 102)$, после чего сгенерированные команды принимают вид, показанный на рис. 6.45, *б*.

Значение всего выражения истинно тогда и только тогда, когда достигаются безусловные переходы в командах 100 и 104, и ложно тогда и только тогда, когда достигаются команды перехода 103 и 105. Поля целевых меток этих команд будут

```
100:  if x < 100 goto _
101:  goto _
102:  if x > 200 goto 104
103:  goto _
104:  if x != y goto _
105:  goto _
```

а) После обратной поправки 104 в команде 102

```
100:  if x < 100 goto _
101:  goto 102
102:  if x > 200 goto 104
103:  goto _
104:  if x != y goto _
105:  goto _
```

б) После обратной поправки 102 в команде 101

Рис. 6.45. Шаги обратных поправок

заполнены позже, когда станет известно, какие действия следует предпринять в зависимости от истинности или ложности всего выражения. □

6.7.3 Инструкции потока управления

Теперь воспользуемся методом обратных поправок для однопроходной трансляции инструкций потока управления. Рассмотрим инструкции, генерируемые следующей грамматикой:

$$\begin{array}{lcl} S & \rightarrow & \textbf{if}\ (B)\ S \mid \textbf{if}\ (B)\ S\ \textbf{else}\ S \mid \textbf{while}\ (B)\ S \mid \{L\} \mid A\ ; \\ L & \rightarrow & L\,S \mid S \end{array}$$

Здесь S – инструкция, L – список инструкций, A – инструкция присваивания, а B – булево выражение. Заметим, что должны быть и другие продукции, такие как и в случае инструкций присваивания. Однако приведенных выше продукций вполне достаточно для иллюстрации методов, используемых при трансляции инструкций потока управления.

Схема кода инструкций **if**, **if-else** и **while** такая же, как и в разделе 6.6. Делается неявное предположение о том, что последовательность в массиве команд отражает естественный поток управления от одной команды к следующей. Если это не

так, для реализации естественного последовательного потока управления должны быть вставлены явные команды переходов.

Схема трансляции на рис. 6.46 поддерживает список команд переходов, которые будут заполнены после выяснения их целевых меток. Как и на рис. 6.43, булевы выражения, генерируемые нетерминалом B, имеют два списка переходов, *B.truelist* и *B.falselist*, соответствующие выходам из кода B по значению *true* и *false* соответственно. Инструкции, генерируемые нетерминалами S и L, имеют списки незаполненных команд переходов (задаваемые атрибутами *nextlist*), которые в конечном итоге будут заполнены в результате обратных поправок. *S.nextlist* представляет собой список всех условных и безусловных переходов к команде, следующей за кодом S в порядке выполнения. *L.nextlist* определяется аналогично.

Рассмотрим семантическое действие 3 на рис. 6.46. Схема кода для продукции $S \rightarrow \textbf{while}\ (B)\ S_1$ выглядит так же, как и на рис. 6.35, *в*. Два маркера-нетерминала M в продукции

$$S \rightarrow \textbf{while}\ M_1\ (B)\ M_2\ S_1$$

записывают номера команд начала кода B и начала кода S_1. Соответствующие метки на рис. 6.35, *в* — *begin* и *B.true*.

Единственная продукция для M — $M \rightarrow \epsilon$. Действие 6 на рис. 6.46 устанавливает атрибут *M.instr* равным номеру следующей команды. После выполнения тела инструкции **while** S_1 управление передается в начало. Таким образом, при свертке **while** $M_1\ (B)\ M_2\ S_1$ в S мы выполняем обратную поправку S_1*.nextlist*, делая все целевые метки в этом списке равными M_1*.instr*. По окончании кода S_1 добавляется явный переход в начало кода B, иначе управление может "выпасть на дно". В команды *B.truelist* вносятся обратные поправки для перехода к началу S_1 путем установки целевых меток равными M_2*.instr*.

Более серьезным аргументом в пользу использования *S.nextlist* и *L.nextlist* является генерация кода для условной инструкции **if** $(B)\ S_1$ **else** S_2. Если управление "выпадает на дно" S_1, как, например, в случае присваивания, необходимо добавить в конце кода S_1 переход через весь код S_2. Для генерации такого перехода после S_1 надо использовать еще один маркер-нетерминал. Пусть этим маркером будет нетерминал N с продукцией $N \rightarrow \epsilon$. N имеет атрибут *N.nextlist*, который будет представлять собой список, состоящий из номера команды перехода `goto _`, генерируемой семантическим действием 7 для N.

Семантическое действие 2 на рис. 6.46 работает с инструкциями **if-else** с синтаксисом

$$S \rightarrow \textbf{if}\ (B)\ M_1\ S_1\ N\ \textbf{else}\ M_2\ S_2$$

Обратная поправка для перехода при истинном B состоит во внесении целевой метки M_1*.instr*, которая представляет собой начало кода S_1. Аналогично поправка перехода при ложном B назначает ему целевую метку, приводящую к началу кода S_2. Список *S.nextlist* включает как все переходы из S_1 и S_2, так и переход,

1) $S \rightarrow \textbf{if} \; (\; B \;) \; M \; S_1$ $\{ \; backpatch(B.truelist, \; M.instr);$
$S.nextlist \; = \; merge(B.falselist, \; S_1.nextlist); \; \}$

2) $S \rightarrow \; \textbf{if} \; (\; B \;) \; M_1 \; S_1 \; N \; \textbf{else} \; M_2 \; S_2$
$\{ \; backpatch(B.truelist, \; M_1.instr);$
$backpatch(B.falselist, \; M_2.instr);$
$temp \; = \; merge(S_1.nextlist, \; N.nextlist);$
$S.nextlist \; = \; merge(temp, \; S_2.nextlist); \; \}$

3) $S \rightarrow \; \textbf{while} \; M_1 \; (\; B \;) \; M_2 \; S_1$
$\{ \; backpatch(S_1.nextlist, \; M_1.instr);$
$backpatch(B.truelist, \; M_2.instr);$
$S.nextlist \; = \; B.falselist;$
$emit('\texttt{goto}' \; M_1.instr); \; \}$

4) $S \rightarrow \{ \; L \; \}$ $\{ \; S.nextlist \; = \; L.nextlist; \; \}$

5) $S \rightarrow A \; ;$ $\{ \; S.nextlist \; = \; \textbf{null}; \; \}$

6) $M \rightarrow \epsilon$ $\{ \; M.instr \; = \; nextinstr; \; \}$

7) $N \rightarrow \epsilon$ $\{ \; N.nextlist \; = \; makelist(nextinstr);$
$emit('\texttt{goto _}'); \; \}$

8) $L \rightarrow L_1 \; M \; S$ $\{ \; backpatch(L_1.nextlist, \; M.instr);$
$L.nextlist \; = \; S.nextlist; \; \}$

9) $L \rightarrow S$ $\{ \; L.nextlist \; = \; S.nextlist; \; \}$

Рис. 6.46. Трансляция инструкций

генерируемый маркером N. (Переменная *temp* представляет собой временную переменную, использующуюся только для объединения списков.)

Семантические действия 8 и 9 обрабатывают последовательности инструкций. В случае

$$L \rightarrow L_1 \;\; M \;\; S$$

команда, следующая в порядке выполнения за кодом L_1, представляет собой начало кода S. Таким образом, в команды списка $L_1.nextlist$ вносится целевая метка, соответствующая началу кода S, которую мы получаем при помощи $M.instr$. В случае $L \rightarrow S$ значение $L.nextlist$ совпадает с $S.nextlist$.

Заметим, что в этих семантических правилах нигде не генерируются новые команды, за исключением правил 3 и 7. Весь прочий код генерируется семантическими действиями, связанными с инструкциями присваивания и выражениями. Поток управления обеспечивает правильность обратных поправок, так что присваивания и вычисления булевых выражений оказываются корректно соединенными.

6.7.4 Инструкции break, continue и goto

Наиболее элементарной конструкцией языков программирования для изменения потока управления является инструкция **goto**. В С инструкция вида `goto L` передает управление инструкции с меткой `L` (в области видимости должна находиться ровно одна инструкция с меткой `L`). Инструкции безусловного перехода могут быть реализованы при помощи списка незаполненных команд перехода для каждой метки с последующим внесением обратных поправок, когда положение метки становится известным.

Язык программирования Java обходится без инструкций безусловного перехода. Однако в этом языке имеются переходы, обеспечиваемые инструкциями `break` (передающей управление за пределы охватывающей конструкции цикла) и `continue` (запускающей новую итерацию охватывающего цикла). Приведенный далее фрагмент кода лексического анализатора иллюстрирует действие этих инструкций:

```
1)  for( ; ; readch() ) {
2)      if ( peek == ' ' || peek == '\t' ) continue;
3)      else if ( peek == '\n' ) line = line + 1;
4)      else break;
5)  }
```

Из строки 4 управление передается команде, следующей за охватывающим циклом **for**. Команда `continue` в строке 2 передает управление коду вычисления функции `readch()` с последующим переходом к инструкции **if** в строке 2.

Если S представляет собой охватывающий цикл, то инструкция `break` приводит к переходу к первой команде после S. Код для `break` можно сгенерировать путем 1) отслеживания охватывающего цикла S, 2) генерации незаполненных переходов для инструкции `break` и 3) помещения этих незаполненных команд в список $S.nextlist$, где $nextlist$ имеет тот же смысл, что и в разделе 6.7.3.

В случае двухпроходной начальной стадии компилятора, строящей синтаксические деревья, $S.nextlist$ можно реализовать в виде поля в узле S. Отслеживать S можно с помощью таблицы символов, чтобы отображать специальный идентификатор **break** на узел охватывающей инструкции цикла S. Такой подход пригоден также и для обработки помеченных инструкций break в Java, поскольку таблица

символов может использоваться и для отображения метки на узел синтаксического дерева помеченной конструкции.

Вместо использования таблицы символов для обращения к узлу S можно поместить указатель на S.*nextlist* в саму таблицу символов. В этом случае по достижении инструкции break мы генерируем незаполненную команду перехода, ищем в таблице символов *nextlist* и добавляем этот переход в список, в котором позже он будет исправлен так, как описывалось в разделе 6.7.3.

Инструкции continue можно обработать аналогично инструкциям break. Основное отличие между ними — в различных целевых метках генерируемых команд перехода.

6.7.5 Упражнения к разделу 6.7

Упражнение 6.7.1. Используя схему на рис. 6.43, выполните трансляцию каждого из следующих выражений. Приведите списки для истинного и ложного значений для каждого подвыражения. Для определенности можно считать, что адрес первой генерируемой команды равен 100.

а) `a==b && (c==d || e==f)`

б) `(a==b || c==d) || e==f`

в) `(a==b && c==d) && e==f`

Упражнение 6.7.2. На рис. 6.47, *а* приведен набросок программы, а на рис. 6.47, *б* — структура генерируемого трехадресного кода при использовании трансляции с обратными поправками на рис. 6.46. Здесь $i_1 \ldots i_8$ — метки генерируемых команд, с которых начинаются фрагменты “Код для...”. При реализации данной трансляции для каждого булева выражения поддерживаются два списка мест в коде E; это списки E.*true* и E.*false*. Места в списке E.*true* — это те места, куда в конечном счете будет помещена метка, в которую передается управление при истинности E; аналогично в списке E.*false* перечисляются места, в которые надо поместить метку, в которую передается управление при ложности E. Кроме того, для каждой инструкции S поддерживается список мест, в которые надо поместить метку, соответствующую потоку управления по завершении вычисления S. Для каждого из приведенных списков укажите значение (одно из $i_1 \ldots i_8$), которое помещается в команды из этого списка.

а) E_3.*false*

б) S_2.*next*

в) E_4.*false*

г) $S_1.next$

д) $E_2.true$

```
while (E1) {                 i1: Код для E1
    if (E2)                  i2: Код для E2
        while (E3)           i3: Код для E3
            S1;              i4: Код для S1
    else {                   i5: Код для E4
        if (E4)              i6: Код для S2
            S2;              i7: Код для S3
        S3                   i8: ...
    }
}
              а)                          б)
```

Рис. 6.47. Структура управления потоком к упражнению 6.7.2

Упражнение 6.7.3. При выполнении трансляции приведенного на рис. 6.47 фрагмента с использованием схемы на рис. 6.46, для каждой инструкции создается список *S.next*, начиная с инструкций присваивания S_1, S_2 и S_3 и переходя ко все бóльшим инструкциям if, if-else, while. Всего на рис. 6.47 имеется пять инструкций такого вида:

S_4 : **while** (E_3) S_1

S_5 : **if** (E_4) S_2

S_6 : Блок, состоящий из S_5 и S_3

S_7 : Инструкция **if** (E_2) S_4 **else** S_6

S_8 : Вся программа

Для каждой из указанных инструкций существует правило, позволяющее построить $S_i.next$ из других списков $S_j.next$ и списков $E_k.true$ и $E_k.false$ для выражений в программе. Приведите эти правила для

а) $S_4.next$;

б) $S_5.next$;

в) $S_6.next$;

г) $S_7.next$;

д) $S_8.next$.

6.8 Инструкции выбора

Инструкции выбора (switch) доступны во многих языках. Рассматриваемый нами синтаксис инструкции выбора показан на рис. 6.48. Имеется вычисляемое выражение-селектор E, за которым следуют n константных значений $V_1, V_2, \ldots,$ V_n, которые может принимать выражение, а также, возможно, "значение" *по умолчанию* (default), которое считается всегда соответствующим значению E в том случае, когда оно не равно ни одному другому значению.

switch (E) {
  case V_1 : S_1
  case V_2 : S_2
  ...
  case V_{n-1} : S_{n-1}
  default : S_n
}

Рис. 6.48. Синтаксис инструкции выбора

6.8.1 Трансляция инструкций выбора

Трансляция инструкции выбора должна состоять в следующем.

1. Вычисление выражения E.

2. Поиск в списке вариантов значения V_j, которое равно значению выражения. Вспомните, что значение по умолчанию соответствует выражению, если ему не соответствует ни одно явно указанное значение V_j.

3. Выполнение инструкции S_j, связанной с найденным значением.

Шаг 2 представляет собой n-путевое ветвление, которое может быть реализовано одним из нескольких способов. Если количество вариантов невелико, скажем, не более 10, то имеет смысл воспользоваться последовательностью условных переходов, каждый из которых выполняет проверку на равенство одному из значений и передает управление соответствующему коду при совпадении.

Компактный способ реализации такой последовательности условных переходов состоит в создании таблицы пар, каждая из которых состоит из значения и метки кода соответствующей инструкции. Вычисленное значение самого выражения в паре с меткой для инструкции по умолчанию помещается в конце таблицы во время выполнения программы. Затем компилятором генерируется простой цикл, который сравнивает значение выражения с каждым значением из таблицы,

которая гарантирует, что если не найдется другого соответствующего значения, то будет выполнена инструкция по умолчанию.

Если количество значений превышает 10 или около того, более эффективным способом является построение хеш-таблицы значений с метками для разных значений в качестве записей. Если в таблице не находится запись для вычисленного значения, генерируется переход к инструкции по умолчанию.

Существует распространенный частный случай, который может быть реализован еще более эффективно, чем n-путевое ветвление. Если все возможные значения лежат в некотором небольшом диапазоне, скажем, от min до max, и количество различных значений представляет собой существенную долю от $max - min$, то можно построить массив из $max - min$ блоков, где блок $j - min$ содержит метку инструкции для значения j; все блоки, которые остаются незаполненными при этом процессе, заполняются метками инструкции по умолчанию.

Для выполнения выбора вычисляется значение выражения j. Затем убеждаемся в том, что это значение находится в диапазоне от min до max, и выполняем косвенный переход с использованием записи таблицы со смещением $j - min$. Например, если выражение имеет символьный тип, может быть создана таблица из, скажем, 128 записей (зависит от используемого множества символов), и переход может выполняться даже без проверки вычисленного значения на принадлежность диапазону.

6.8.2 Синтаксически управляемая трансляция инструкций выбора

Промежуточный код на рис. 6.49 представляет собой трансляцию инструкции выбора на рис. 6.48. Все проверки вынесены в конец, так что простой генератор кода в состоянии распознать многопутевое ветвление и сгенерировать для него эффективный код с использованием наиболее подходящей реализации, предложенной в начале данного раздела.

Более прямой способ, показанный на рис. 6.50, требует от компилятора проведения интенсивного анализа для поиска наиболее эффективной реализации. Заметим, что в случае однопроходного компилятора размещение инструкций ветвления в начале оказывается неудобным, поскольку компилятор не может генерировать код для каждой инструкции S_i сразу же, как только встретит ее.

Для трансляции в вид, показанный на рис. 6.49, встретив ключевое слово **switch**, следует сгенерировать две новые метки, `test` и `next`, и новую временную переменную `t`. Затем при синтаксическом анализе E генерируются код для вычисления значения E во временную переменную `t` и команда безусловного перехода `goto test`.

После этого, когда встречается каждое из ключевых слов **case**, создается новая метка L_i, которая вносится в таблицу символов. В очередь, использующуюся

```
          Код для вычисления E в t
          goto test
L1:       Код S1
          goto next
L2:       Код S2
          goto next
          ...
Ln-1:     Код Sn-1
          goto next
Ln:       Код Sn
          goto next
test:     if t = V1 goto L1
          if t = V2 goto L2
          ...
          if t = Vn-1 goto Ln-1
          goto Ln
next:
```

Рис. 6.49. Трансляция инструкции выбора

```
          Код для вычисления E в t
          if t != V1 goto L1
          Код S1
          goto next
L1:       if t != V2 goto L2
          Код S2
          goto next
L2:
          ...
Ln-2:     if t != Vn-1 goto Ln-1
          Код Sn-1
          goto next
Ln-1:     Код Sn
next:
```

Рис. 6.50. Еще одна трансляция инструкции выбора

```
case t V1 L1
case t V2 L2
...
case t Vn-1 Ln-1
case t t Ln
next:
```

Рис. 6.51. Трехадресные команды case, использующиеся при трансляции инструкции выбора

для хранения вариантов case, помещается пара "значение – метка", состоящая из значения константы варианта V_i и L_i (или указателя на запись для L_i в таблице символов). Обработка каждой инструкции **case** V_i : S_i дает метку L_i, прикрепленную к коду S_i, за которым следует команда безусловного перехода `goto next`.

Когда мы добираемся таким образом до конца инструкции switch, у нас все готово для генерации кода n-путевого ветвления. Считывая из очереди пары "значение – метка", можно сгенерировать последовательность трехадресных команд, показанных на рис. 6.51. Здесь `t` – временная переменная, хранящая значение выражения-селектора E, а L_n – метка для инструкции по умолчанию.

Команда `case t` V_i L_i представляет собой синоним для `if t =` V_i `goto` L_i на рис. 6.49, но при использовании команд `case` генератору кода проще определить, что перед ним кандидат для обработки особым способом. На стадии генерации кода такая последовательность инструкций `case` может быть транслирована в n-путевое ветвление наиболее эффективным способом в зависимости от количества значений и их диапазона.

6.8.3 Упражнения к разделу 6.8

! **Упражнение 6.8.1.** Для трансляции инструкции выбора в последовательность команд `case`, как показано на рис. 6.51, транслятор должен создать список пар "значение – метка" при обработке исходного текста инструкции выбора. Это можно сделать с помощью дополнительной трансляции, которая накапливает такие пары. Набросайте синтаксически управляемое определение, которое создает список пар, генерируя при этом код инструкций S_i, представляющих собой действия для каждого из вариантов case.

6.9 Промежуточный код процедур

Процедуры и их реализация будут подробно рассматриваться в главе 7 вместе с управлением памятью для имен в процессе выполнения программы. В этом разделе для процедуры, которая возвращает значение, будет использоваться термин

"функция". Здесь мы вкратце рассмотрим объявления функций и трехадресный код для их вызовов. В трехадресном коде вызов функции раскрывается в вычисление параметров при подготовке к последующему вызову функции. Для простоты считаем, что все параметры передаются по значению (методы передачи параметров рассматриваются в разделе 1.6.6).

Пример 6.25. Предположим, что a — массив целых чисел и что f — функция от целочисленного значения, возвращающая целое число. Тогда присваивание

```
n = f(a[i]);
```

может быть транслировано в следующий трехадресный код:

1) t_1 = i * 4
2) t_2 = a [t_1]
3) param t_2
4) t_3 = call f, 1
5) n = t_3

Две первые строки вычисляют значение выражения a[i] во временную переменную t_2, как рассматривалось в разделе 6.4. Строка 3 делает переменную t_2 фактическим параметром вызова функции f с одним параметром в строке 4. Строка 4 также присваивает значение, возвращаемое функцией, переменной t_3, а в строке 5 это значение присваивается переменной n. □

Продукции на рис. 6.52 делают возможными определения функций и их вызовы. (Приведенный синтаксис генерирует лишнюю запятую после последнего параметра, но для дидактических целей вполне пригоден.) Нетерминалы D и T генерируют соответственно объявления и типы, как в разделе 6.3. Определение функции, генерируемое нетерминалом D, состоит из ключевого слова **define**, возвращаемого типа, имени функции, формальных параметров в скобках и тела функции, представляющего собой инструкцию в фигурных скобках. Нетерминал F генерирует несколько формальных параметров (возможно, нуль); каждый формальный параметр состоит из типа, за которым следует идентификатор. Нетерминалы S и E генерируют соответственно инструкции и выражения. Продукция для S добавляет инструкцию, возвращающую значение выражения, а продукция для E — вызов функции с фактическим параметром, генерируемым A. Фактический параметр представляет собой выражение.

Определения и вызовы функций могут транслироваться с использованием концепций, которые уже рассматривались в данной главе.

- *Типы функций.* Тип функции должен кодировать возвращаемый тип и типы формальных параметров. Обозначим через *void* специальный тип, означающий отсутствие параметра или возвращаемого значения. Таким образом,

$$\begin{array}{lcl} D & \rightarrow & \textbf{define}\ T\ \textbf{id}\ (F)\ \{S\} \\ F & \rightarrow & \epsilon \mid T\ \textbf{id}\ ,\ F \\ S & \rightarrow & \textbf{return}\ E\ ; \\ E & \rightarrow & \textbf{id}\ (A) \\ A & \rightarrow & \epsilon \mid E\ ,\ A \end{array}$$

Рис. 6.52. Добавление функций в исходный язык программирования

тип функции *pop* (), которая возвращает целое число, — "функция от *void*, возвращающая *integer*". Типы функций могут быть представлены с применением конструктора *fun* к возвращаемому типу и упорядоченному списку типов параметров.

- *Таблицы символов.* Пусть s — текущая таблица символов при достижении определения функции. Имя функции вносится в таблицу символов s для использования в оставшейся части программы. Формальные параметры функции могут быть обработаны точно так же, как и имена полей записи (см. рис. 6.18). В продукции D после **define** и имени функции мы заносим s в стек и создаем новую таблицу символов

  $$Env.push\,(top)\,; \quad top = \textbf{new}\ Env\,(top)\,;$$

  Назовем новую таблицу символов t. Обратите внимание, что *top* передается в качестве параметра в вызов **new** *Env* (*top*), так что новая таблица символов t может быть связана с предыдущей — s. Новая таблица символов t используется для трансляции тела функции. После трансляции тела функции мы вернемся к предыдущей таблице символов s.

- *Проверка типов.* В выражениях функция рассматривается как и любой другой оператор. Таким образом, материал из раздела 6.5.2, включая неявные преобразования типов, распространяется и на функции. Например, если f — функция с параметром, представляющим собой действительное число, то при вызове $f\,(2)$ целое число 2 преобразуется в действительное число.

- *Вызовы функций.* При генерации трехадресных команд для вызова функции **id** $(E, E, \ldots, E)$ достаточно сгенерировать трехадресные команды для вычисления или преобразования параметров E в адреса, за которыми следуют команды `param` для каждого из параметров. Если мы не хотим смешивать команды вычисления параметров и команды `param`, то для каждого выражения E можно сохранить атрибут $E.addr$ в структуре данных вроде очереди. После того, как все выражения транслированы, генерируются команды `param` до полного опустошения очереди.

Процедуры – настолько важная и часто используемая программная конструкция, что от компилятора, безусловно, требуется генерация эффективного кода для вызова процедур и возврата из них. Подпрограммы времени выполнения, обрабатывающие передачу параметров, вызовы и возвраты, являются частью пакета поддержки времени выполнения. Механизмы поддержки времени выполнения рассматриваются в главе 7.

6.10 Резюме к главе 6

Методы, описанные в этой главе, могут быть объединены для построения простой начальной стадии компилятора наподобие приведенной в приложении А. Начальная стадия компилятора может строиться инкрементно.

- *Выбор промежуточного представления.* Промежуточное представление обычно является некоторой комбинацией графических обозначений и трехадресного кода. В графическом представлении синтаксических деревьев узлы представляют конструкции языка; дочерние узлы представляют их подконструкции. Трехадресный код получил свое название от команд вида $x = y$ **op** z, с не более чем одним оператором в команде. Имеются также дополнительные команды для потока управления.

- *Трансляция выражений.* Выражения со встроенными операциями могут быть развернуты в последовательность отдельных операций путем связывания действий с каждой продукцией вида $E \to E_1$ **op** E_2. Действие либо создает узел для E с дочерними узлами E_1 и E_2, либо генерирует трехадресную команду, которая применяет **op** к адресам E_1 и E_2 и помещает результат в новое временное имя, которое становится адресом E.

- *Проверка типов.* Тип выражения E_1 **op** E_2 определяется оператором **op** и типами E_1 и E_2. Зачастую применяется неявное преобразование типов, такое как преобразование *integer* в *float*. Промежуточный код содержит явные преобразования типов, обеспечивающие точное соответствие типов операндов типам, ожидаемым оператором.

- *Использование таблицы символов для реализации объявлений.* Объявление определяет тип имени. Размер типа равен количеству памяти, необходимой для имени с данным типом. Использование размеров позволяет вычислять относительные адреса имен времени выполнения как смещения от начала области данных. Тип и относительный адрес имени помещаются в таблицу символов объявлением, так что в дальнейшем транслятор может использовать их всякий раз, когда это имя встречается в выражении.

- *Массивы.* Для быстрого обращения элементы массивов хранятся в последовательных ячейках памяти. Массивы массивов располагаются таким образом, что могут рассматриваться как одномерные массивы отдельных элементов. Тип массива используется для вычисления адресов элементов массива относительно его базы.

- *Генерация кода с переходами для булевых выражений.* При сокращенных вычислениях или в коде с переходами значение булева выражения неявно определяется достигнутой позицией кода. Применимость кода с переходами обусловлена тем, что обычно булевы выражения используются для потока управления, как в случае **if** (B) S. Булево выражение может быть вычислено путем перехода к команде `t = true` или `t = false`, где `t` — временное имя. При использовании меток переходов булево выражение может быть транслировано при помощи наследуемых меток, соответствующих переходам по истинному и ложному значениям. Константы *true* и *false* транслируются в переходы к соответствующим меткам.

- *Реализация инструкций с использованием потока управления.* Инструкции могут транслироваться с использованием наследуемой метки *next*, которая маркирует первую команду после кода данной инструкции. Условная инструкция $S \rightarrow$ **if** (B) S_1 может транслироваться путем назначения новой метки, маркирующей начало кода S_1, и использования ее и $S.next$ в качестве переходов для истинного и ложного значений B соответственно.

- *Использование обратных поправок.* Обратные поправки представляют собой метод однопроходной генерации кода булевых выражений и инструкций. Идея заключается в поддержании списков незавершенных команд переходов, так что все команды в списке имеют одну и ту же целевую метку. Когда эта метка становится известной, выполняется завершение команд из списка путем заполнения их полей целевой метки.

- *Реализация записей.* Имена полей в записи или классе могут рассматриваться как последовательности объявлений. Тип записи кодирует типы и относительные адреса полей. Для этой цели может использоваться таблица символов.

6.11 Список литературы к главе 6

Большинство описанных в этой главе методов берут свое начало из бурной деятельности, сопровождавшей появление Algol 60. Синтаксически управляемая трансляция в промежуточный код твердо устоялась во времена создания Pascal [11] и C [6 и 9].

В середине 1950-х годов появился UNCOL (Universal Compiler Oriented Language – универсальный компиляторно-ориентированный язык), мифический универсальный промежуточный язык, при использовании которого компилятор мог строиться путем объединения начальной стадии для данного исходного языка программирования и заключительной стадии для данного целевого языка [10]. Методы раскрутки (bootstrapping) описаны в сообщении [10] и регулярно используются для переноса компиляторов.

Идеал UNCOL, заключающийся в связывании начальных и заключительных стадий компилятора, может быть достигнут различными способами. Переносимый компилятор состоит из одной начальной стадии, которая может объединяться с несколькими заключительными стадиями для реализации данного языка программирования на нескольких разных машинах. Ранним примером языка с переносимым компилятором, написанным на самом этом языке, является Neliac [5]. Другой подход заключается в модификации начальной стадии нового языка для использования в существующем компиляторе. Фельдман (Feldman) [2] описал добавление начальной стадии Fortran 77 к компиляторам C [6 и 9]. GCC (GNU Compiler Collection – набор компиляторов GNU) [3] поддерживает начальные стадии для C, C++, Objective-C, Fortran, Java и Ada.

Номера значений и их реализация путем хеширования разработаны Ершовым [1].

Использование информации о типах для повышения безопасности байт-кода Java описано Гослингом (Gosling) [4].

Выведение типов путем применения унификации для решения множеств уравнений было открыто несколько раз; его применение к ML описано Милнером (Milner) [7]. Всестороннее рассмотрение типов можно найти у Пирса (Pierce) [8].

1. Ershov, A. P., "On programming of arithmetic operations", *Comm. ACM* **1**:8 (1958), pp. 3–6. См. также *Comm. ACM* **1**:9 (1958), p. 16.

2. Feldman, S. I., "Implementation of a portable Fortran 77 compiler using modern tools", *ACM SIGPLAN Notices* **14**:8 (1979), pp. 98–106.

3. Начальная страница GCC `http://gcc.gnu.org/`, Free Software Foundation.

4. Gosling, J., "Java intermediate bytecodes", *Proc. ACM SIGPLAN Workshop on Intermediate Representations* (1995), pp. 111–118.

5. Huskey, H. D., M. H. Halstead, and R. McArthur, "Neliac – a dialect of Algol", *Comm. ACM* **3**:8 (1960), pp. 463–468.

6. Johnson, S. C., "A tour through the portable C compiler", Bell Telephone Laboratories, Inc., Murray Hill, N. J., 1979.

7. Milner, R., "A theory of type polymorphism in programming", *J. Computer and System Sciences* **17**:3 (1978), pp. 348–375.

8. Pierce, B. C., *Types and Programming Languages*, MIT Press, Cambridge, Mass., 2002.

9. Ritchie, D. M., "A tour through the UNIX C compiler", Bell Telephone Laboratories, Inc., Murray Hill, N. J., 1979.

10. Strong, J., J. Wegstein, A. Tritter, J. Olsztyn, O. Mock, and T. Steel, "The problem of programming communication with changing machines: a proposed solution", *Comm. ACM* **1**:8 (1958), pp. 12–18. Part 2: **1**:9 (1958), pp. 9–15. Report of the SHARE Ad-Hoc Committee on Universal Languages.

11. Wirth, N. "The design of a Pascal compiler", *Software — Practice and Experience* **1**:4 (1971), pp. 309–333.

Глава 7

Среды времени выполнения

Компилятор должен точно реализовывать абстракции, воплощенные в определении исходного языка программирования. Обычно эти абстракции включают рассмотренные в разделе 1.6 концепции, такие как имена, области видимости, связывание, типы данных, операторы, процедуры, параметры и конструкции потока управления. Компилятор должен сотрудничать с операционной системой и другим программным обеспечением для поддержки этих абстракций на целевой машине.

Для этого компилятор создает среду времени выполнения (run-time environment), в которой, как предполагается, будет выполняться целевая программа, и управляет ею. Эта среда решает множество вопросов, таких как схема размещения и память для именованных объектов исходной программы, механизмы, используемые целевой программой для доступа к переменным, связи между процедурами, механизмы передачи параметров, взаимодействие с операционной системой, устройствами ввода-вывода и другими программами.

Две главные темы данной главы — это выделение памяти и доступ к переменным и данным. Мы более детально рассмотрим управление памятью, включая такие вопросы, как распределение стека, управление кучей и сборка мусора. В следующей главе будут рассмотрены методы генерации целевого кода для многих распространенных языковых конструкций.

7.1 Организация памяти

С точки зрения разработчика компилятора, целевая программа работает в собственном адресном пространстве, в котором у каждого программного значения есть свое местоположение в памяти. Управление этим логическим адресным пространством и его организация разделяются между компилятором, операционной системой и целевой машиной. Операционная система отображает логические адреса на физические, которые обычно разбросаны в памяти.

Представление времени выполнения объектной программы в пространстве логических адресов состоит из областей данных и программы, как показано на рис. 7.1. Компилятор для языка наподобие C++ в операционной системе типа Linux может подразделять память указанным образом.

Рис. 7.1. Типичное разделение памяти времени выполнения на области кода и данных

В этой книге мы полагаем, что память времени выполнения имеет вид блоков смежных байтов, где байт — наименьшая адресуемая единица памяти. Байт состоит из восьми битов, а четыре байта образуют машинное слово. Многобайтные объекты хранятся в последовательных байтах, а их адреса определяются адресами их первых байтов.

Как говорилось в главе 6, количество памяти, необходимое для конкретного имени, определяется его типом. Элементарные типы данных, такие как символы, целые и действительные числа, могут храниться в целом количестве байтов. Память, выделенная для составных типов, таких как массивы или структуры, должна быть достаточно велика для хранения всех их компонентов.

На схему размещения объектов данных в памяти сильное влияние оказывают ограничения адресации на целевой машине. На многих машинах команды сложения целых чисел могут требовать *выравнивания* (align) целых чисел, т.е. их размещения в адресах, кратных 4. Хотя массив из десяти символов требует только десяти байтов для хранения своего содержимого[1], компилятор может выделить ему для достижения корректного выравнивания 12 байт, оставляя 2 байта неиспользуемыми. Память, остающаяся неиспользуемой из-за выравнивания, известна как *заполнение* (padding). В случаях, когда вопросы экономии памяти выходят на первое место, компилятор может *паковать* (pack) данные так, чтобы заполнения не было; однако при этом могут потребоваться дополнительные команды времени выполнения для позиционирования упакованных данных таким образом, чтобы они могли быть обработаны так, как если бы они были корректно выровнены.

Размер сгенерированного целевого кода фиксируется во время компиляции, так что компилятор может разместить выполнимый целевой код в статически

[1] В языках, где размер символа — 1 байт. — *Прим. ред.*

определенной области (*Код* на рис. 7.1), обычно в нижних адресах памяти. Аналогично в процессе компиляции может быть известен размер некоторых объектов данных программы, таких как глобальные константы, и данных, сгенерированных компилятором, таких как информация для поддержки сборки мусора; такие данные также могут быть размещены в другой статически определяемой области (*Статические данные* на рис. 7.1). Одна из причин статического выделения памяти для максимально возможного количества объектов данных заключается в том, что адреса этих объектов могут быть встроены в целевой код. В ранних версиях Fortran память для всех объектов данных могла выделяться только статически.

Для максимального использования памяти во время выполнения программы две другие области, *Стек* и *Куча*, находятся по разные стороны оставшегося адресного пространства. Это динамические области — их размеры могут изменяться в процессе работы программы, причем области при необходимости растут одна навстречу другой. Стек используется для хранения структур данных, называющихся записями активации и генерируемых при вызовах процедур.

На практике стек растет по направлению к меньшим адресам, а куча — по направлению к бóльшим. Однако в этой и следующей главах мы считаем, что стек также растет в направлении бóльших адресов, что позволяет использовать во всех наших примерах положительные значения смещений.

Как будет видно из следующего раздела, для хранения при вызове процедуры информации о состоянии машины, такой как счетчик команд и регистры машины, используется запись активации. Когда управление возвращается из вызова, активация вызывающей процедуры может быть возобновлена после восстановления значений регистров и установки счетчика команд в точку, следующую непосредственно за вызовом. Объекты данных, время жизни которых находится в пределах времени жизни активации, могут располагаться в стеке вместе с другой информацией, связанной с активацией.

Многие языки программирования позволяют программисту выделять память для данных и освобождать ее под управлением программы. Например, C имеет функции `malloc` и `free`, которые могут использоваться для получения и освобождения блоков памяти требуемого размера. Для управления данными такого вида используется куча. В разделе 7.4 будут рассматриваться различные алгоритмы управления памятью, которые могут использоваться для работы с кучей.

7.1.1 Статическое и динамическое распределение памяти

Выделение памяти для данных и схема их размещения в памяти в среде времени выполнения являются ключевыми вопросами управления памятью. Это достаточно сложные вопросы, поскольку одно и то же имя в исходном тексте программы может обозначать несколько разных мест в памяти во время работы про-

граммы. Два прилагательных — *статическое* и *динамическое* — предназначены для того, чтобы различать действия во время компиляции и во время выполнения программы. Мы говорим, что решение о выделении памяти статическое, если оно принимается во время компиляции, когда известен только исходный текст программы, но не то, что именно программа делала в процессе работы. И наоборот, решение является динамическим, если оно может быть принято исключительно при выполнении программы. Многие компиляторы используют для динамического выделения памяти некоторую комбинацию двух следующих стратегий.

1. *Хранение в стеке.* Память для локальных имен процедуры выделяется в стеке. Стек времени выполнения мы будем рассматривать начиная с раздела 7.2. Стек поддерживает обычную политику вызова процедур и возврата из них.

2. *Хранение в куче.* Память для данных, которые должны "пережить" вызов создающей их процедуры, обычно выделяется в куче, которая представляет собой многократно используемую память. Работу с кучей мы будем рассматривать начиная с раздела 7.4. Куча представляет собой область виртуальной памяти, которая позволяет объектам или другим элементам данных получать место для хранения при создании и освобождать его, когда эти объекты становятся недействительными.

Одно из средств поддержки управления кучей — "сборка мусора" (garbage collection) — позволяет системе времени выполнения находить ненужные элементы данных и повторно использовать занимаемую ими память, даже если программист явно не освободил ее. Автоматическая сборка мусора представляет собой существенный элемент многих современных языков программирования, несмотря на трудность эффективного выполнения данной операции; в ряде языков применение такой технологии попросту невозможно.

7.2 Выделение памяти в стеке

Почти все компиляторы языков программирования, в которых применяются пользовательские функции, процедуры или методы, как минимум часть своей памяти времени выполнения используют в качестве стека. Всякий раз при вызове процедуры[2] в стеке выделяется пространство для ее локальных переменных, а по завершении процедуры это пространство снимается со стека. Как мы увидим, такой метод не только позволяет совместно использовать память не перекрывающимся по времени процедурам, но и обеспечивает возможность компиляции кода

[2]Напомним, что "процедура" представляет собой обобщенный термин для процедур, функций, методов или подпрограмм.

процедуры таким образом, что относительные адреса ее нелокальных переменных всегда остаются одними и теми же, независимо от последовательности вызовов процедур.

7.2.1 Деревья активации

Выделение памяти в стеке не имело бы столько преимуществ, если бы вызовы, или *активации* (activation), не могли бы быть вложенными во времени. Приведенный далее пример иллюстрирует вложенность вызовов процедур.

Пример 7.1. На рис. 7.2 приведен набросок программы, которая считывает девять целых чисел в массив a и сортирует их с помощью рекурсивного алгоритма быстрой сортировки.

```
int a[11];
void readArray() { /* Считываем 9 целых чисел в a[1], ..., a[9]. */
    int i;
    ...
}
int partition(int m, int n) {
    /* Выбор разделителя v и разделение a[m..n]
       так, что a[m..p-1] меньше v, a[p] = v,
       а a[p+1..n] не меньше v. Возврат p. */
    ...
}
void quicksort(int m, int n) {
    int i;
    if (n > m) {
        i = partition(m, n);
        quicksort(m, i-1);
        quicksort(i+1, n);
    }
}
main() {
    readArray();
    a[0] = -9999;
    a[10] = 9999;
    quicksort(1,9);
}
```

Рис. 7.2. Набросок программы быстрой сортировки

Функция *main* решает три задачи. Она вызывает функцию *readArray*, устанавливает ограничители, а потом вызывает функцию *quicksort* для всего массива данных. На рис. 7.3 показана последовательность вызовов, которая может получиться в результате работы программы. При этом вызов *partition* $(1, 9)$ возвращает 4, так что элементы массива с $a[1]$ по $a[3]$ хранят значения, меньшие выбранного разделителя v, в то время как значения, бóльшие v, размещаются в элементах от $a[5]$ до $a[9]$. □

```
Вход в main()
    Вход в readArray()
    Выход из readArray()
    Вход в quicksort(1,9)
        Вход в partition(1,9)
        Выход из partition(1,9)
        Вход в quicksort(1,3)
            ...
        Выход из quicksort(1,3)
        Вход в quicksort(5,9)
            ...
        Выход из quicksort(5,9)
    Выход из quicksort(1,9)
Выход из main()
```

Рис. 7.3. Возможная последовательность активаций программы, представленной на рис. 7.2

В данном примере, как и в общем случае, активации процедур оказываются вложенными во времени. Если активация процедуры p вызывает процедуру q, то активация q должна завершиться раньше, чем завершится активация p. Обычно складывается одна из трех ситуаций.

1. Активация q завершается нормально. Тогда, по сути, в любом языке программирования управление возвращается в точку p, следующую непосредственно за точкой, в которой был сделан вызов q.

2. Активация q или некоторой процедуры, вызванной q прямо или косвенно, завершается аварийно, т.е. продолжение выполнения программы невозможно. В этом случае p завершается одновременно с q.

3. Активация q завершается из-за сгенерированного исключения, которое q обработать не в состоянии. Процедура p может обработать исключение; в этом случае активация q завершается, в то время как активация p продолжается, хотя и не обязательно с точки, в которой была вызвана q. Если p

Версия быстрой сортировки

Набросок программы быстрой сортировки на рис. 7.2 использует две вспомогательные функции: *readArray* и *partition*. Функция *readArray* используется только для загрузки данных в массив a. Первый и последний элементы a не используются для хранения данных, представляя собой "ограничители", устанавливаемые функцией `main`. Считаем, что $a[0]$ имеет значение, меньшее любого возможного значения данных, а $a[10]$ — большее.

Функция *partition* разбивает часть массива, определяемую аргументами m и n, таким образом, что меньшие элементы подмножества от $a[m]$ до $a[n]$ находятся в начале, а бóльшие — в конце, хотя эти подгруппы элементов не обязаны находиться в отсортированном порядке. Мы не вдаемся в подробности того, как работает функция *partition*, за исключением того, что она может воспользоваться существованием ограничителей. Код одного из возможных алгоритмов *partition* представлен на рис. 9.1.

Рекурсивная процедура *quicksort* сначала выясняет, требуется ли сортировать более одного массива. Поскольку один элемент всегда "отсортирован", *quicksort* в этом случае не делает ничего. Если же сортировать надо несколько элементов, *quicksort* сначала вызывает *partition*, которая возвращает индекс i элемента, разделяющего меньшие и бóльшие элементы. Эти две группы элементов сортируются двумя рекурсивными вызовами *quicksort*.

не может обработать исключение, то данная активация p завершается одновременно с активацией q и, скорее всего, исключение будет обработано некоторой другой открытой активацией процедуры.

Следовательно, активации процедур в процессе выполнения всей программы можно представить в виде дерева, называемого *деревом активаций* (activation tree). Каждый узел соответствует одной активации, а корень представляет собой активацию "главной" процедуры, которая инициирует выполнение программы. В узле активации процедуры p дочерние узлы соответствуют активациям процедур, вызываемых данной активацией p. Эти активации указываются слева направо в порядке их вызовов. Заметим, что активация, соответствующая дочернему узлу, должна быть завершена до того, как начнется активация его соседа справа.

Пример 7.2. Одно из возможных деревьев активации, содержащее полную последовательность вызовов процедур и возвратов из них, показано на рис. 7.4. Функции на рисунке представлены первыми буквами их имен. Не забывайте, что данное дерево — только одно из множества возможных, поскольку аргументы по-

следовательных вызовов, а также количество вызовов вдоль любой ветви зависит от значений, возвращаемых функцией *partition*. □

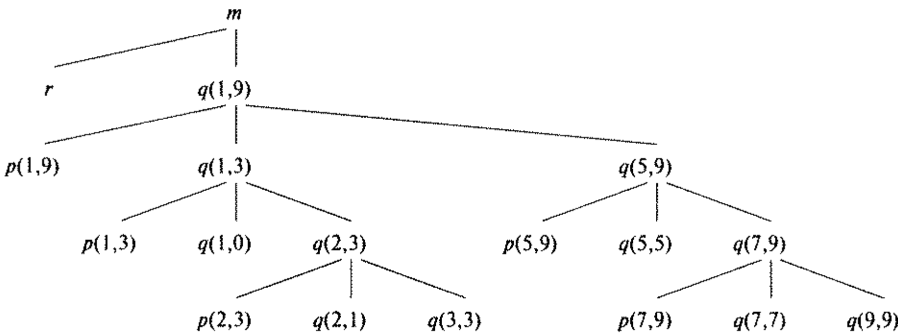


Рис. 7.4. Дерево активации, представляющее вызовы в процессе выполнения быстрой сортировки

Возможность использования стека времени выполнения обеспечивается некоторыми взаимосвязями между деревом активации и поведением программы.

1. Последовательность вызовов процедур соответствует обходу дерева активации в прямом порядке.

2. Последовательность возвратов из процедур соответствует обходу дерева активации в обратном порядке.

3. Предположим, что управление находится в некоторой активации какой-то процедуры, соответствующей узлу N дерева активации. Тогда открытыми в настоящий момент активациями (*активными* — live) являются активации, соответствующие узлу N и его предкам. Порядок вызова этих активаций определяется порядком их появления на пути от корня дерева активации до узла N, а возврат из них будет осуществляться в порядке, обратном порядку их активаций.

7.2.2 Записи активации

Вызовы процедур и возвраты из них обычно управляются стеком времени выполнения, именуемым *стеком управления* (control stack). Каждая активная активация имеет *запись активации* (activation record), иногда именуемую *кадром* (frame), в стеке управления. На дне стека находится запись активации корня дерева активации, а последовательность записей активации в стеке соответствует пути в дереве активации от корня до текущей активации, в которой в настоящий момент находится управление. Запись последней по времени активации находится на вершине стека.

Пример 7.3. Если управление в настоящий момент находится в активации $q(2,3)$ дерева на рис. 7.4, то запись активации для $q(2,3)$ находится на вершине стека управления. Непосредственно под ней находится запись активации для $q(1,3)$ — родителя $q(2,3)$ в дереве активации. Еще ниже в стеке находится запись активации для $q(1,9)$, а в самом низу стека размещена запись активации для m — активации функции `main`, представленной корнем дерева активации. □

Будет удобнее изображать стеки управления так, чтобы низ стека находился выше его вершины, так что элементы записи активации, находящиеся ниже на странице книги, на самом деле были ближе к вершине стека.

Содержимое записи активации варьируется в зависимости от реализуемого языка программирования. Вот список данных, которые могут находиться в записи активации (элементы данных и их возможный порядок в стеке показаны на рис. 7.5).

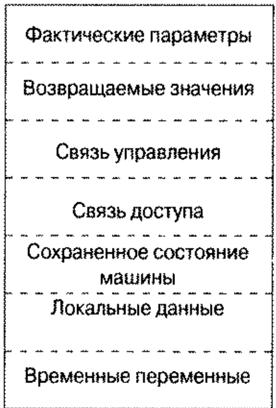


Рис. 7.5. Общий вид записи активации

1. Вре́менные значения, появляющиеся, например, в процессе вычисления выражений, когда эти значения не могут храниться в регистрах.

2. Локальные данные процедуры, к которой относится данная запись активации.

3. Информация о состоянии машины непосредственно перед вызовом процедуры. Эта информация обычно включает *адрес возврата* (значение счетчика программы, которое должно быть восстановлено по выходу из процедуры) и содержимое регистров, которые использовались вызывающей процедурой и должны быть восстановлены при возврате из вызываемой процедуры.

4. *Связь доступа* может потребоваться для обращения вызванной процедуры к данным, хранящимся в другом месте, например в другой записи активации. Связи доступа рассматриваются в разделе 7.3.5.

5. *Связь управления* указывает на запись активации вызывающей процедуры.

6. Память для возвращаемого значения вызываемой функции, если таковое имеется. Не все вызываемые процедуры возвращают значения, а кроме того, для большей эффективности возвращаемое значение может находиться не в стеке, а в регистре.

7. Фактические параметры, используемые вызываемой процедурой. Зачастую эти значения также располагаются не в стеке, а по возможности для большей эффективности передаются в регистрах. Однако в общем случае место для них отводится в стеке.

Пример 7.4. На рис. 7.6 показаны снимки стека времени выполнения при прохождении управления по дереву активации на рис. 7.4. Пунктирные линии в частях дерева указывают завершенные активации. Поскольку массив a — глобальный, память для него выделяется до начала активации функции *main*, как показано на рис. 7.6, *а*.

Когда управление достигает первого вызова в теле функции *main*, активируется процедура r и ее запись активации вносится в стек (см. рис. 7.6, *б*). Запись активации для r содержит пространство для локальной переменной i (вспомните, что вершина стека находится внизу). Когда управление возвращается из этой активации, ее запись снимается со стека, оставляя там только одну запись активации функции *main*. Затем управление достигает вызова q с фактическими параметрами 1 и 9 и на вершину стека помещается запись активации для этого вызова, как показано на рис. 7.6, *в*. Запись активации для q содержит пространство для параметров m и n и локальной переменной i, следуя общей схеме на рис. 7.5. Обратите внимание, что сейчас повторно используется пространство стека, которое ранее было отведено для записи активации r. Никакие локальные данные функции r не доступны для вызова $q(1,9)$. Когда происходит возврат из вызова функции $q(1,9)$, стек вновь содержит только одну запись активации функции *main*.

Между двумя последними снимками на рис. 7.6 произошло несколько активаций. Был выполнен рекурсивный вызов $q(1,3)$. За время жизни этой активации начались и закончились активации $p(1,3)$ и $q(1,0)$, оставляя запись активации для $q(1,3)$ на вершине стека (см. рис. 7.6, *г*). Заметим, что в случае рекурсивной процедуры одновременное наличие нескольких ее записей активации в стеке — обычное явление. □

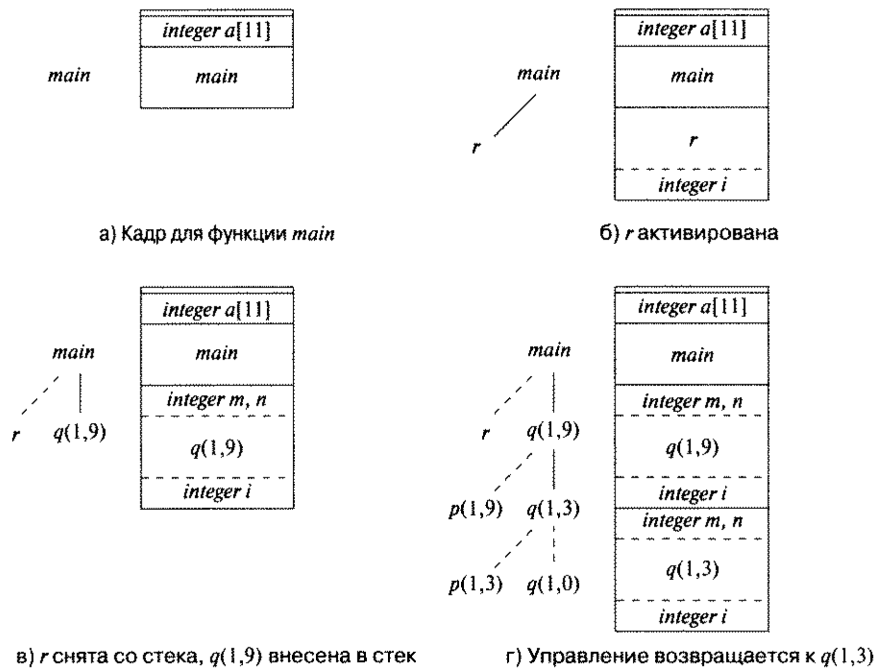


Рис. 7.6. Растущий вниз стек записей активации

7.2.3 Последовательности вызовов

Вызовы процедур реализованы с помощью *последовательностей вызовов* (calling sequences), состоящих из кода, который выделяет память в стеке для записи активации и вносит информацию в ее поля. *Последовательность возврата* представляет собой аналогичный код, который восстанавливает состояние машины, так что вызывающая процедура может продолжать работу.

Последовательности вызовов и схема записей активации могут существенно отличаться даже в разных реализациях одного и того же языка. Код в последовательности вызова зачастую разделяется между вызывающей и вызываемой процедурами. Не существует четкого разграничения задач времени выполнения между вызывающей и вызываемой процедурами; исходный язык, целевая машина и операционная система налагают свои требования, в силу которых может оказаться предпочтительным то или иное решение. В общем случае, если процедура вызывается n раз, то часть последовательности вызова в вызывающих процедурах генерируется n раз. Однако часть последовательности вызова в вызываемой процедуре генерируется лишь единожды. Следовательно, желательно разместить как можно бóльшую часть последовательности вызова в коде вызываемой проце-

дуры — с учетом того, что известно вызываемой процедуре при ее вызове. Как мы увидим, вызываемая процедура не имеет доступа ко всей информации.

При разработке последовательностей вызовов и схемы записей активации помогают следующие принципы.

1. Значения, передаваемые между вызывающей и вызываемой процедурами, в общем случае помещаются в начале записи активации вызываемой процедуры, так что они максимально близки к записи активации вызывающей процедуры. Смысл этого заключается в том, что вызывающая процедура может вычислить значения фактических параметров вызова и разместить их на вершине собственной записи активации, что позволяет ей избежать создания полной записи активации вызываемой процедуры или даже знания ее схемы. Более того, это позволяет реализовать процедуры, которые принимают при каждом вызове различное количество аргументов, как, например, функция `printf` в языке программирования C. Вызываемая функция знает, где в ее записи активации следует разместить возвращаемое значение; ее же аргументы располагаются в стеке последовательно ниже этого места.

2. Элементы фиксированного размера обычно располагаются посредине. Как видно из рис. 7.5, эти элементы обычно включают связь управления, связь доступа и состояние машины. При каждом вызове сохраняются одни и те же компоненты состояния машины, так что сохранение и восстановление состояния для каждого вызова осуществляется одним и тем же кодом. Более того, при стандартизации информации о состоянии машины программы наподобие отладчиков смогут легко расшифровывать состояние стека при возникновении ошибки.

3. Элементы, размер которых может быть не известен заранее, размещаются в конце записи активации. Большинство локальных переменных имеет фиксированную длину, которая может быть определена компилятором исходя из типа переменной. Однако некоторые локальные переменные имеют размер, который не может быть выяснен до выполнения программы; наиболее распространенным примером является массив с динамическим размером, который определяется одним из параметров вызываемой процедуры. Кроме того, количество памяти, необходимой для временных переменных, обычно зависит от того, насколько успешно они распределяются по регистрам на стадии генерации кода. Таким образом, хотя объем памяти, необходимой для временных переменных в конечном счете оказывается известен компилятору, он может не быть известен в момент генерации промежуточного кода.

4. Следует ответственно подходить к вопросу помещения указателя на вершину стека. Распространенный подход состоит в том, чтобы он указывал на конец полей фиксированного размера в записи активации. Тогда обращение к данным фиксированной длины может осуществляться с использованием фиксированных смещений относительно указателя на вершину стека, известных генератору промежуточного кода. Следствием такого метода является размещение полей переменной длины записи активации "выше" вершины стека. Их смещения вычисляются во время выполнения программы, но обращение к ним выполняется также посредством указателя на вершину стека — просто значения смещений оказываются положительными.

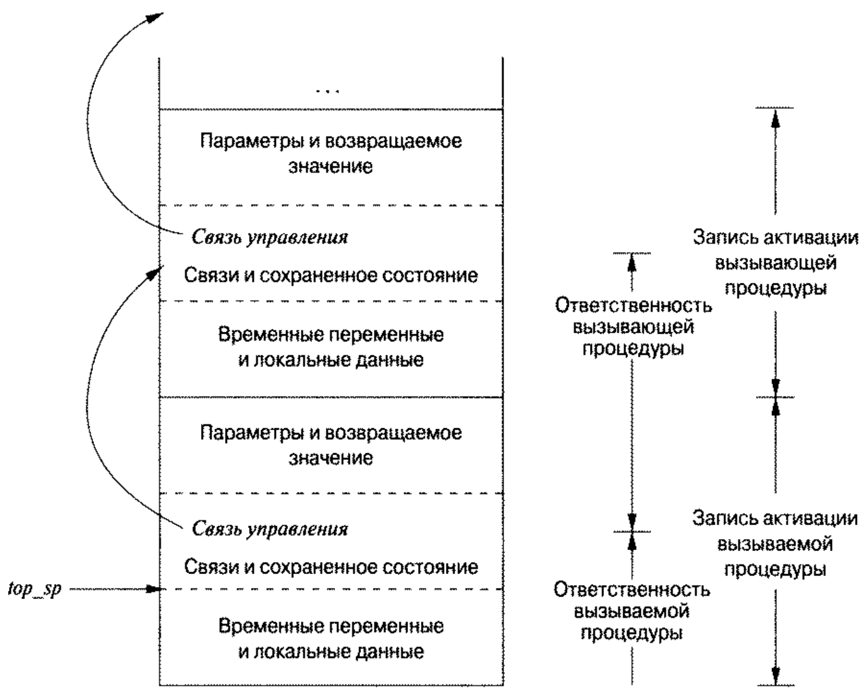


Рис. 7.7. Разделение задач между вызывающей и вызываемой процедурами

В качестве примера того, как вызывающая и вызываемая процедуры могут сотрудничать в плане управления стеком, может быть рис. 7.7. Регистр *top_sp* указывает на конец поля состояния машины в текущей записи активации. Эта позиция внутри записи активации вызываемой процедуры известна вызывающей процедуре, так что она может быть сделана ответственной за установку значения *top_sp* перед передачей управления вызываемой процедуре. Последовательность вызова и ее разделение между вызывающей и вызываемой процедурами имеет следующий вид.

1. Вызывающая процедура вычисляет фактические параметры.

2. Вызывающая процедура сохраняет адрес возврата и старое значение *top_sp* в записи активации вызываемой процедуры. Затем вызывающая процедура увеличивает *top_sp* до позиции, показанной на рис. 7.7, т.е. *top_sp* перемещается за локальные данные и временные переменные вызывающей процедуры и за поля параметров и состояния машины вызываемой процедуры.

3. Вызываемая процедура сохраняет значения регистров и другую информацию о состоянии машины.

4. Вызываемая процедура инициализирует свои локальные данные и начинает выполнение.

Соответствующая последовательность возврата выглядит следующим образом.

1. Вызываемая процедура помещает возвращаемое значение после параметров, как показано на рис. 7.5.

2. Используя информацию в поле состояния машины, вызывающая процедура восстанавливает *top_sp* и другие регистры и переходит к адресу возврата, который помещен вызывающей процедурой в поле состояния.

3. Хотя значение *top_sp* было уменьшено, вызывающая процедура знает, где находится возвращаемое значение по отношению к текущему значению *top_sp*; таким образом, вызывающая процедура может получить доступ к возвращаемому значению и использовать его.

Приведенные выше последовательности вызова и возврата позволяют количеству аргументов вызываемой процедуры изменяться от вызова к вызову (как в функции `printf` языка программирования C). Обратите внимание, что во время компиляции целевому коду вызывающей процедуры известны количество и типы аргументов, передаваемых вызываемой процедуре. Следовательно, вызывающая процедура знает размер области параметров. Целевой код вызываемой процедуры, однако, должен быть готов к обработке различного количества параметров, так что после вызова он исследует поле параметров. При использовании организации, представленной на рис. 7.7, информация, описывающая параметры, должна располагаться после поля состояния машины, чтобы вызываемая процедура легко могла ее найти. Например, в случае функции `printf` языка программирования C первый аргумент описывает остальные, так что после того, как функция находит первый аргумент, она знает, где располагаются все остальные аргументы.

7.2.4 Данные переменной длины в стеке

Система управления памятью времени выполнения зачастую должна выделять память для объектов, размеры которых во время компиляции неизвестны, но которые являются локальными объектами процедуры и, таким образом, могут размещаться в стеке. В современных языках память для объектов неизвестного во время компиляции размера выделяется из кучи, которая будет рассматриваться в разделе 7.4. Однако выделение памяти в стеке для объектов, массивов и других структур неизвестного размера возможно, и мы рассмотрим здесь, как это делается. Причина, по которой размещение объектов в стеке предпочтительнее размещения в куче, в том, что в этом случае мы избегаем затрат на сборку мусора. Заметим, что стек может использоваться только для объектов, локальных по отношению к данной процедуре и становящихся недоступными после возврата из нее.

Распространенная стратегия выделения памяти для массивов переменной длины (т.е. массивов, размеры которых зависят от значений одного или нескольких параметров вызываемой процедуры) показана на рис. 7.8. Та же схема работает для объектов любого типа, если они локальны для вызываемой процедуры и имеют размер, зависящий от параметров вызова.

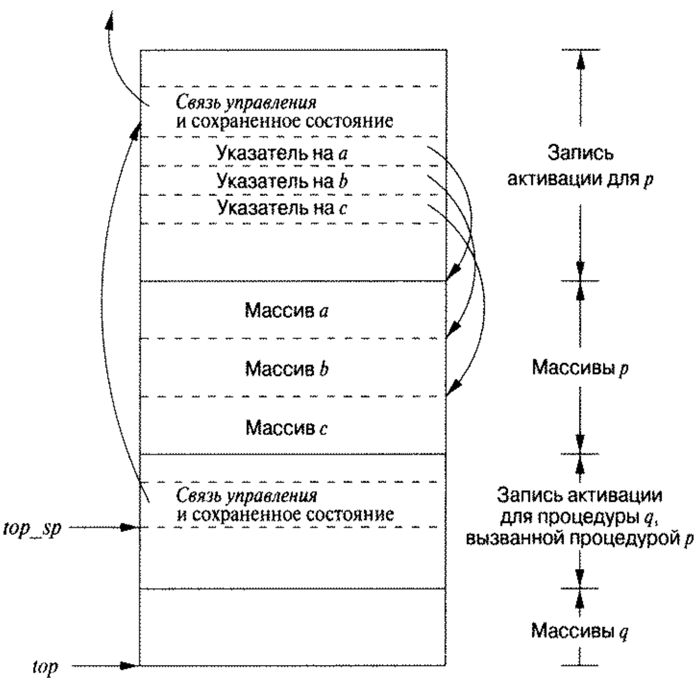


Рис. 7.8. Доступ к динамически распределенным массивам

На рис. 7.8 процедура p имеет три локальных массива, размеры которых, как мы полагаем, невозможно вычислить во время компиляции. Память для этих массивов не является частью записи активации для p, хотя она и находится в стеке. В самой записи активации находятся только указатели на начало каждого массива. Таким образом при выполнении p эти указатели находятся в позициях с известными смещениями относительно указателя на вершину стека, так что целевой код может получить доступ к элементам массива посредством этих указателей.

На рис. 7.8 также показана запись активации для процедуры q, вызванной процедурой p. Запись активации для q начинается после массивов p, и все массивы переменной длины процедуры q располагаются за ней.

Доступ к данным в стеке осуществляется при помощи двух указателей — *top* и *top_sp*. Здесь *top* отмечает фактическую вершину стека; он указывает на позицию, в которой начнется очередная запись активации. Второй указатель, *top_sp*, используется для поиска локальных полей фиксированной длины в записи активации на вершине стека. Для согласованности с рис. 7.7 мы считаем, что указатель *top_sp* указывает на конец поля состояния машины. На рис. 7.8 *top_sp* указывает на конец этого поля в записи активации для q. Там можно найти поле связи управления для q, которое приведет к тому месту в записи активации для p, на которое указывал *top_sp*, когда запись активации для p находилась на вершине стека.

Код для перестановки *top* и *top_sp* может быть сгенерирован во время компиляции с использованием размеров, которые станут известны только во время выполнения. Когда осуществляется возврат из q, *top_sp* может быть восстановлен по сохраненной связи управления в записи активации для q. Новое значение *top* представляет собой (старое невосстановленное) значение *top_sp* минус длина полей состояния машины, связей управления и доступа, возвращаемого значения и параметров (см. рис. 7.5) в записи активации q. Эта длина известна вызывающей процедуре во время компиляции и может зависеть от вызывающей процедуры в случае, когда количества параметров могут изменяться от вызова к вызову q.

7.2.5 Упражнения к разделу 7.2

Упражнение 7.2.1. Предположим, что программа на рис. 7.2 использует функцию *partition*, которая всегда выбирает в качестве разделителя v элемент массива $a[m]$. Считаем также, что при переупорядочении массива $a[m], \dots, a[n]$ его порядок по возможности сохраняется, т.е. сначала идут все элементы, меньшие v, в их исходном относительном порядке, затем — элемент v, а затем — все элементы, бóльшие v, также в исходном относительном порядке.

а) Изобразите дерево активаций для сортировки чисел 9, 8, 7, 6, 5, 4, 3, 2, 1.

б) Чему равно максимальное количество записей активации, одновременно находящихся в стеке?

Упражнение 7.2.2. Повторите упражнение 7.2.1 для набора чисел 1, 3, 5, 7, 9, 2, 4, 6, 8.

Упражнение 7.2.3. На рис. 7.9 приведен код на языке программирования C, рекурсивно вычисляющий числа Фибоначчи. Предположим, что запись активации для f включает следующие элементы в указанном порядке: (возвращаемое значение, аргумент n, локальная переменная s, локальная переменная t). В записи активации находятся и другие обычные элементы. Приведенные ниже вопросы предполагают, что начальным вызовом был вызов $f(5)$.

а) Изобразите полное дерево активаций.

б) Как выглядит стек и записи активации в нем в момент первого возврата из вызова $f(1)$?

! в) Как выглядит стек и записи активации в нем в момент пятого возврата из вызова $f(1)$?

```
int f(int n) {
    int t, s;
    if (n < 2) return 1;
    s = f(n-1);
    t = f(n-2);
    return s+t;
}
```

Рис. 7.9. Код вычисления чисел Фибоначчи к упражнению 7.2.3

Упражнение 7.2.4. Перед вами набросок двух функций на языке программирования C:

```
int f(int x) { int i; ··· return i+1; ··· }
int g(int y) { int j; ··· f(j+1) ··· }
```

Как видите, функция g вызывает функцию f. Изобразите вершину стека, начиная с записи активации g, после того, как g вызывает f, в момент непосредственно перед возвратом из функции f. Вы можете рассматривать только возвращаемые значения, параметры, связи управления и память для локальных переменных; рассматривать сохраненное состояние машины или временные переменные (а также локальные переменные, помимо приведенных в наброске) не требуется. Для каждого из элементов вы должны указать следующее.

а) Какая функция создает место в стеке для элемента?

б) Какая функция записывает значение элемента?

в) Какой записи активации принадлежит элемент?

Упражнение 7.2.5. В языке, в котором параметры передаются по ссылке, имеется функция $f(x, y)$, делающая следующее:

```
x = x + 1;  y = y + 2;  return x+y;
```

Что вернет вызов $f(a, a)$, если перед ним a присвоено значение 3?

Упражнение 7.2.6. Функция f на языке программирования C определена следующим образом:

```
int f(int x, *py, **ppz) {
    **ppz += 1; *py += 2; x += 3; return x+y+z;
}
```

Переменная a представляет собой указатель на b; переменная b — указатель на c, а c — целочисленная переменная, значение которой в настоящий момент равно 4. Что вернет вызов $f(c, b, a)$?

7.3 Доступ к нелокальным данным в стеке

В этом разделе мы рассмотрим, как процедура обращается к своим данным. Особенно важен механизм поиска данных, используемых в процедуре p, но не принадлежащих p. Доступ становится особенно сложным в языках программирования, которые допускают объявление одних процедур в пределах других процедур. Поэтому мы начнем с простого случая функций C, а затем перейдем к языку программирования ML, который допускает как вложенные объявления функций, так и использование функций в качестве объектов, т.е. функции могут как принимать функции в качестве параметров, так и возвращать функции. Эта возможность может быть обеспечена путем изменения реализации стека времени выполнения, и мы рассмотрим несколько вариантов такого изменения записей активации из раздела 7.2.

7.3.1 Доступ к данным при отсутствии вложенных процедур

В семействе языков программирования C все переменные определены либо в одной из функций, либо вне любой функции (“глобально”). Самое важное то,

что невозможно объявить одну процедуру, область видимости которой находится полностью внутри другой процедуры. Глобальная переменная v имеет область видимости, которая состоит из всех функций, следующих после объявления v, за исключением мест, где имеется локальное определение идентификатора v. Переменные, объявленные внутри функции, имеют область видимости, состоящую только из этой функции (или ее части, если в функции имеются вложенные блоки, рассматривавшиеся в разделе 1.6.3).

Для языков, которые не допускают вложенные объявления процедур, выделение памяти для переменных и доступ к ним просты.

1. Глобальным переменным выделяется статическая память. Размещение этих переменных остается фиксированным и известно во время компиляции. Так что для доступа к любой переменной, не являющейся локальной для текущей выполняемой процедуры, просто используются статически определенные адреса.

2. Любое другое имя должно быть локально для активации на вершине стека. Обратиться к этим переменным можно при помощи указателя стека *top_sp*.

Важное преимущество статического распределения памяти для глобальных переменных заключается в том, что объявленные процедуры могут передаваться в качестве параметров и возвращаться в качестве результата (в C передается указатель на функцию) без существенных изменений в стратегии обращения к данным. В случае правил статических областей видимости C и при отсутствии вложенных процедур любое имя, нелокальное для одной процедуры, будет нелокальным для всех процедур, независимо от того, как они были активированы. Аналогично, если процедура возвращается как результат, нелокальные имена в ней ссылаются на статически выделенную для них память.

7.3.2 Вложенные процедуры

Доступ существенно усложняется, если язык программирования разрешает объявлениям процедур быть вложенными и при этом использует обычные правила областей видимости (т.е. процедура имеет доступ к переменным процедуры, объявление которой охватывает объявление текущей процедуры, следуя правилам для вложенных областей видимости, которые были изложены для блоков в разделе 1.6.3). Причина этого в том, что знание во время компиляции того, что объявление p непосредственно вложено в объявление q ничего не говорит нам об относительном размещении их записей активации во время выполнения. В действительности, поскольку процедуры p или q (или обе) могут быть рекурсивными, в стеке могут быть несколько записей активации для p и/или q.

Поиск объявления, применимого к нелокальному имени x во вложенной процедуре p выполняется статически; он может быть выполнен при помощи рас-

ширения статических правил областей видимости для блоков. Предположим, что имя x объявлено в охватывающей процедуре q. Поиск соответствующей активации q из активации p выполняется динамически, так как требует дополнительной информации времени выполнения об активациях. Одно из возможных решений заключается в использовании "связей доступа", о которых мы расскажем в разделе 7.3.5.

7.3.3 Язык с вложенными объявлениями процедур

Семейство языков программирования C, как и многие другие распространенные языки программирования, не поддерживает вложенные процедуры, так что мы вкратце расскажем о них здесь. Вложенные процедуры в языках программирования имеют долгую историю. Предшественник C Algol 60 обладал этой возможностью, как и его другой потомок — Pascal, популярный учебный язык программирования. Среди более поздних языков программирования с вложенными процедурами одним из наиболее важных является ML, у которого мы позаимствуем синтаксис и семантику (см. врезку "Дополнительная информация об ML").

- ML представляет собой *функциональный язык*; это означает, что однажды объявленные и инициализированные переменные не изменяются. Существует лишь несколько исключений, таких как массивы, элементы которых могут быть изменены специальными вызовами функций.

- Переменные определяются и получают свои начальные неизменяемые значения при помощи инструкций вида

  `val` ⟨имя⟩ `=` ⟨выражение⟩

- Функции определяются при помощи следующего синтаксиса:

  `fun` ⟨имя⟩ `(`⟨аргументы⟩`)` `=` ⟨тело⟩

- Для тел функций используются инструкции вида

  `let` ⟨список определений⟩ `in` ⟨инструкции⟩ `end`

  Определения обычно являются инструкциями `val` или `fun`. Область видимости каждого такого определения состоит из всех определений до `in` и всех инструкций до `end`. Особенно важно то, что определения функций могут быть вложенными. Например, тело функции p может содержать `let`-инструкцию, которая включает определение другой (вложенной) функции q. Аналогично q может иметь в своем теле определения функций, что приводит к вложенности функций произвольной глубины.

Дополнительная информация об ML

В дополнении к чистой функциональности ML готовит массу сюрпризов программисту, работающему с языком программирования C или иным языком из его семейства.

- ML поддерживает *функции высшего порядка* (higher-order functions), т.е. функция может принимать функции в качестве аргументов, а также создавать и возвращать другие функции. Эти функции, в свою очередь, могут принимать функции как аргументы — до любого уровня.
- ML, по сути, не имеет итераций, таких, например, как for- и while-инструкции C. Эффект итераций достигается путем применения рекурсии. Этот подход является неотъемлемым свойством функционального языка, поскольку изменение переменной итерации наподобие i в `for(i=0;i<10;i++)` в C невозможно. Вместо этого ML делает i аргументом функции, и эта функция вызывает саму себя с возрастающим значением i, пока не будет достигнут требуемый предел.
- ML поддерживает в качестве примитивных типов данных списки и помеченные древовидные структуры.
- ML не требует объявления типов переменных. Вместо этого он выводит типы во время компиляции, и, если это невозможно, программа считается содержащей ошибку. Например, из `val x = 1` совершенно очевидно, что x имеет целочисленный тип, а из `val y = 2*x` становится ясно, что тот же тип имеет и y.

7.3.4 Глубина вложенности

Присвоим *глубину вложенности* (nesting depth) 1 процедурам, которые не вложены ни в какую иную процедуру. Например, все функции C имеют глубину вложенности 1. Однако если процедура p определена непосредственно в процедуре с глубиной вложенности i, то глубина вложенности такой процедуры p составляет $i + 1$.

Пример 7.5. На рис. 7.10 приведен набросок нашего примера быстрой сортировки на ML. Единственная функция с глубиной вложенности 1 — внешняя функция *sort*, которая считывает массив a из 9 целых чисел и сортирует его с применением алгоритма быстрой сортировки. Во второй строке функции *sort* определен массив a. Обратите внимание на вид этого объявления ML. Первый аргумент `array`

указывает, что нам нужен массив из 11 элементов; все массивы ML индексируются целыми числами, начинающимися с 0, так что этот массив очень похож на массив a языка программирования C на рис. 7.2. Второй аргумент `array` говорит о том, что изначально все элементы массива a хранят значение 0. Выбор целочисленного начального значения 0 позволяет компилятору ML вывести, что a — массив целых чисел, так что мы не должны объявлять тип a.

```
 1) fun sort(inputFile, outputFile) =
        let
 2)         val a = array(11,0);
 3)         fun readArray(inputFile) = ··· ;
 4)             ··· a ··· ;
 5)         fun exchange(i,j) =
 6)             ··· a ··· ;
 7)         fun quicksort(m,n) =
                let
 8)                 val v = ··· ;
 9)                 fun partition(y,z) =
10)                     ··· a ··· v ··· exchange ···
                in
11)                 ··· a ··· v ··· partition ··· quicksort
                end
        in
12)         ··· a ··· readArray ··· quicksort ···
        end;
```

Рис. 7.10. Версия быстрой сортировки в стиле ML с использованием вложенных функций

Внутри *sort* объявлены несколько функций: *readArray*, *exchange* и *quicksort*. Предполагается, что в строках 4 и 6 функции *readArray* и *exchange* получают доступ к массиву a. Заметим, что в ML массивы позволяют нарушать функциональную природу языка и обе указанные функции изменяют значение элементов a, как и в C-версии быстрой сортировки. Поскольку все эти три функции определены непосредственно внутри функции с глубиной вложенности 1, их глубины вложенности равны 2.

Строки 7–11 раскрывают ряд деталей функции *quicksort*. В строке 8 объявляется локальное значение v — опорный элемент разбиения. В строке 10 предполагается, что функция *partition* обращается как к массиву a, так и к опорному элементу v, а также вызывает функцию *exchange*. Поскольку функция *partition* определена внутри функции с глубиной вложенности 2, ее глубина вложенности

равна 3. В строке 11 предполагается, что функция *quicksort* обращается к переменным a и v, функции *partition* и рекурсивно к самой себе.

Строка 12 предполагает, что охватывающая функция *sort* обращается к a и вызывает две процедуры — *readArray* и *quicksort*. □

7.3.5 Связи доступа

Непосредственная реализация обычного статического правила области видимости для вложенных функций получается путем добавления к каждой записи активации указателя, называющегося *связью доступа* (access link). Если в исходном тексте процедура p непосредственно вложена в процедуру q, то связь доступа в каждой активации p указывает на последнюю активацию q. Заметим, что глубина вложенности q должна быть ровно на единицу меньше глубины вложенности процедуры p. Связи доступа образуют цепочку от записи активации на вершине стека к последовательности активаций с монотонно уменьшающимися глубинами вложенности. Вдоль этой цепочки располагаются все активации, данные и процедуры которых доступны для текущей выполняющейся процедуры.

Предположим, что на вершине стека находится процедура p, глубина вложенности которой — n_p, и p требуется доступ к имени x, представляющему собой элемент, определенный в некоторой процедуре q, которая окружает p и имеет глубину вложенности n_q. Понятно, что $n_q \leqslant n_p$, причем равенство выполняется, только если p и q — одна и та же процедура. Поиск x начнем с записи активации для p на вершине стека и проследуем по связям доступа от записи активации к записи активации $n_p - n_q$ раз. Таким образом мы доберемся до записи активации q, и это всегда будет самая последняя (наивысшая) запись активации для q из имеющихся в стеке. Эта запись активации содержит искомый элемент x. Поскольку компилятору известна схема записи активации, x может быть найдено с некоторым фиксированным смещением от позиции в записи активации q, на которую указывает последняя связь доступа.

Пример 7.6. На рис. 7.11 показана последовательность состояний стека, которая может получиться при выполнении функции *sort* на рис. 7.10. Как и ранее, имена функций представлены только их первыми буквами; на рисунке представлены некоторые данные, которые могут находиться в различных записях активации, а также связи доступа для каждой записи активации. На рис. 7.11, *а* показана ситуация после того, как функция *sort* вызвала функцию *readArray* для загрузки входных данных в массив a и функцию *quicksort* $(1, 9)$ для сортировки массива. Связь доступа *quicksort* $(1, 9)$ указывает на запись активации для *sort*, но не потому, что *sort* вызвала *quicksort*, а потому, что *sort* — наиболее близкая вложенная функция, охватывающая *quicksort* в программе на рис. 7.10.

В последовательных шагах на рис. 7.11 показаны рекурсивный вызов *quicksort* $(1, 3)$, за которым следует вызов функции *partition*, которая, в свою

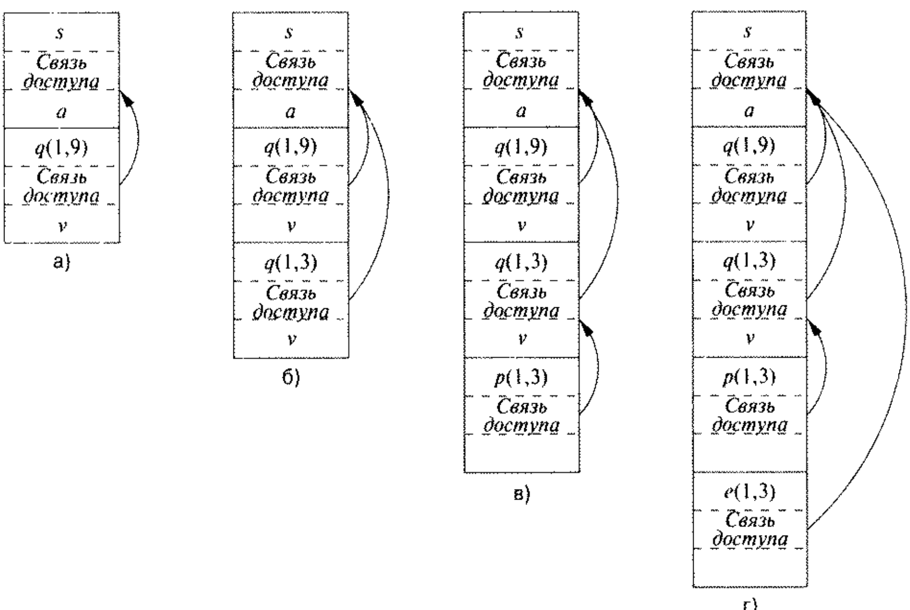


Рис. 7.11. Связи доступа для поиска нелокальных данных

очередь, вызывает функцию *exchange*. Обратите внимание, что связь доступа *quicksort* $(1, 3)$ указывает на *sort* по той же причине, что и в случае *quicksort* $(1, 9)$.

На рис. 7.11, *г* связь доступа "перепрыгивает" через записи активации *quicksort* и *partition*, поскольку функция *exchange* вложена непосредственно в *sort*. Так и должно быть, поскольку *exchange* требуется доступ только к массиву a, в котором данная функция должна обменять два элемента, определяемые параметрами i и j. □

7.3.6 Работа со связями доступа

Как определяются связи доступа? Простой случай — когда вызов представляет собой вызов некоторой процедуры по явно указанному имени. Более сложная ситуация — когда происходит вызов процедуры, переданной в качестве параметра. В этом случае конкретная вызываемая процедура становится известна только во время выполнения программы и глубина вложенности вызываемой процедуры может отличаться от вызова к вызову. Давайте сначала рассмотрим случай, когда процедура q явным образом вызывает процедуру p. Возможны три ситуации.

1. Процедура p имеет бóльшую глубину вложенности, чем q. В таком случае p должна быть определена непосредственно в q, иначе вызов процедурой q оказывается не в пределах области видимости имени процедуры p. Таким образом, глубина вложенности процедуры p ровно на единицу больше глу-

бины вложенности q и связь доступа должна вести от p к q. В этом простом случае достаточно включить в последовательность вызова шаг, который помещает в связь доступа p указатель на запись активации q. Примером может служить вызов процедуры *quicksort* процедурой *sort* на рис. 7.11, *а* или вызов процедуры *partition* процедурой *quicksort* на рис. 7.11, *в*.

2. Вызов рекурсивен, т.е. $q = p$.[3] В этом случае связь доступа для новой записи активации та же, что и запись активации ниже нее в стеке. Примером может служить вызов процедуры $quicksort\,(1,3)$ процедурой $quicksort\,(1,9)$ на рис. 7.11, *б*.

3. Глубина вложенности n_p процедуры p меньше глубины вложенности n_q процедуры q. Для того чтобы вызов в q находился в области видимости имени p, процедура q должна быть вложена в некоторую процедуру r, а p должна быть процедурой, определенной непосредственно в r. Тогда верхняя запись активации r будет найдена после прохождения $n_q - n_p + 1$ звеньев цепочки связей доступа, начинающейся в записи активации q. После этого связь активации p должна вести к упомянутой активации r.[4]

Пример 7.7. В качестве примера третьей ситуации рассмотрим переход от рис. 7.11, *в* к рис. 7.11, *г*. Глубина вложенности вызванной функции *exchange* равна 2, что на единицу меньше глубины вложенности вызывающей функции *partition*, равной 3. Таким образом, мы начинаем с записи активации для *partition* и проходим по $3 - 2 + 1 = 2$ связям доступа, что приводит нас от записи активации для *partition* через запись активации *quicksort* $(1, 3)$ к записи активации *sort*. Следовательно, связь доступа для *exchange* должна вести к записи активации для *sort*, что мы и видим на рис. 7.11, *г*.

Эквивалентный способ состоит в следовании по $n_q - n_p$ звеньям цепочки связей доступа с последующим копированием связи доступа из найденной записи активации. В нашем примере мы должны пройти одно звено цепочки, попадая в запись активации для *quicksort* $(1, 3)$, после чего скопировать из нее связь доступа, ведущую к *sort*. Обратите внимание, что эта связь оказывается корректной для процедуры *exchange*, несмотря на то, что последняя находится вне области видимости *quicksort*, являясь для нее "братской" функцией по вложенности в *sort*. □

[3]В ML возможны взаимно рекурсивные функции, обрабатываемые аналогично.

[4]Правило из третьей ситуации распространяется и на случай, когда $n_p = n_q$, т.е. когда вызовы не взаимно рекурсивны, но находятся на одном уровне вложенности. Таким образом, ситуация 2 по сути является частным случаем ситуации 3. — *Прим. ред.*

7.3.7 Связи доступа для процедур, являющихся параметрами

Когда процедура p передается процедуре q в качестве параметра, а затем процедура q вызывает этот параметр (таким образом вызывая p в данной активации q), вполне возможно, что q не известен контекст, в котором p появляется в программе. Если это так, то q не известно, как установить связь доступа для p. Решение этой проблемы в следующем: при использовании процедуры в качестве параметра вызывающая процедура должна передать наряду с именем процедуры корректную связь доступа.

Вызывающей процедуре всегда известна эта связь, поскольку, если p передается процедурой r как фактический параметр, p должна быть именем, доступным для r, а следовательно, r может определить связь доступа для p так же, как если бы эта процедура была вызвана ею непосредственно, т.е. с использованием правил для построения связи доступа из раздела 7.3.6.

Пример 7.8. На рис. 7.12 приведен набросок ML-функции a, которая имеет вложенные функции b и c. Функция b получает параметр-функцию f, которую затем вызывает. Функция c определяет в себе функцию d, а затем вызывает b, передавая ей фактический параметр d.

```
fun a(x) =
    let
        fun b(f) =
            ··· f ··· ;
        fun c(y) =
            let
                fun d(z) = ···
            in
                ··· b(d) ···
            end
    in
        ··· c(1) ···
    end;
```

Рис. 7.12. Набросок программы ML, использующей передачу функции в качестве параметра

Рассмотрим, что происходит при выполнении a. Сначала a вызывает c, так что мы размещаем запись активации для c в стеке над a. Связь доступа для c указывает на запись для a, поскольку c определена непосредственно в a. Затем c вызывает $b(d)$. Последовательность вызова настраивает запись активации b так, как показано на рис. 7.13, a.

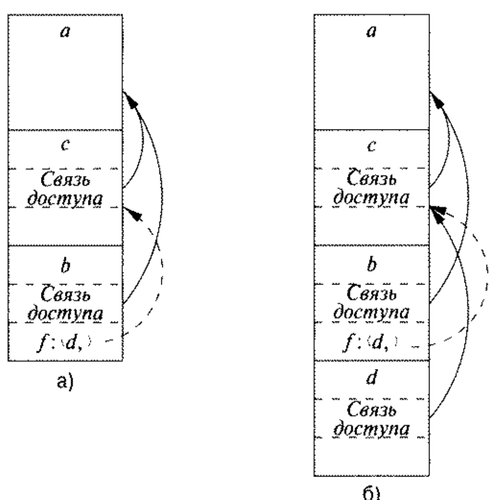


Рис. 7.13. Фактический параметр включает связь доступа

В этой записи активации находится фактический параметр d и его связь доступа, которые вместе образуют значение формального параметра f в записи активации для b. Заметим, что функция d известна функции c, поскольку d определена в c, а следовательно, c передает в качестве связи доступа указатель на собственную запись активации. Не имеет значения, где именно была определена d; если c находится в области видимости этого определения, то должно быть применимо одно из трех правил раздела 7.3.6 и c может предоставить необходимую связь доступа.

Теперь рассмотрим, что делает b. Мы знаем, что в некоторой точке она использует свой параметр f, применение которого состоит в вызове d. В стеке появляется запись активации для d, как показано на рис. 7.13, *б*. Корректная связь доступа для размещения в записи активации находится в значении параметра f; эта связь указывает на запись активации для c, поскольку c непосредственно окружает определение d. Заметим, что b оказывается в состоянии установить корректное значение связи доступа, несмотря на то, что b находится вне области видимости определения c. □

7.3.8 Дисплеи

Одна из проблем обращения к нелокальным данным при помощи связей доступа заключается в том, что, когда глубина вложенности становится большой, может потребоваться пройти по длинной цепочке связей доступа для достижения необходимых данных. Более эффективная реализация использует вспомога-

тельный массив d, именуемый *дисплеем*[5] (display), который содержит по одному указателю для каждой глубины вложенности. Мы договариваемся, что в любой момент времени $d[i]$ представляет собой указатель на наивысшую запись активации в стеке для процедуры с глубиной вложенности i. Пример работы с дисплеем приведен на рис. 7.14. Например, на рис. 7.14, *г* мы видим дисплей d, элемент $d[1]$ которого хранит указатель на запись активации для *sort*, наивысшей (и единственной) записи активации для функции с глубиной вложенности 1. Аналогично $d[2]$ хранит указатель на запись активации для *exchange* — наивысшую запись для глубины 2, а $d[3]$ — указатель на запись активации для *partition* — наивысшую запись для глубины 3.

Преимущество использования дисплея состоит в том, что при выполнении процедуры p, которой требуется доступ к элементу x некоторой процедуры q, нам достаточно провести поиск только в $d[i]$, где i — глубина вложенности q; мы следуем по указателю $d[i]$ в запись активации для q, в которой с известным смещением находится x. Компилятору известно значение i, так что он в состоянии сгенерировать код для обращения к x с использованием $d[i]$ и смещения x относительно вершины записи активации для q. Таким образом, коду никогда не приходится следовать по длинной цепочке связей доступа.

Для корректной поддержки дисплея мы должны сохранять предыдущие значения записей дисплея в записях активации. Если вызывается процедура p с глубиной вложенности n_p и ее запись активации не первая в стеке для процедур с глубиной вложенности n_p, то запись активации для p должна хранить предыдущее значение $d[n_p]$, которое теперь указывает на данную активацию p. При возврате из процедуры p и удалении ее записи активации из стека мы восстанавливаем значение $d[n_p]$ таким, каким оно было до вызова p.

Пример 7.9. На рис. 7.14 показано несколько шагов управления дисплеем. На рис. 7.14, *а* функция *sort* с глубиной вложенности 1 вызывает *quicksort*$(1,9)$ с глубиной вложенности 2. Запись активации для *quicksort* содержит место для хранения старого значения $d[2]$, которое в силу того, что к этому моменту в стеке нет записей активации для процедур с глубиной вложенности 2, представляет собой нулевой указатель.

На рис. 7.14, *б* *quicksort*$(1,9)$ вызывает *quicksort*$(1,3)$. Поскольку оба вызова имеют глубину 2, указатель на *quicksort*$(1,9)$, хранившийся в $d[2]$, должен быть сохранен в записи активации для *quicksort*$(1,3)$. Указатель в $d[2]$ после этого делается указывающим на *quicksort*$(1,3)$.

Далее вызывается функция *partition*. Эта функция имеет глубину вложенности 3, так что сейчас впервые используется элемент $d[3]$, который становится

[5] В русскоязычной литературе по компиляции этот термин появился в 1960-е годы в виде кальки английского *display* — "выставка", "показ". Ввиду локальности его использования мы оставляем его таким же. По сути же он означает стек ссылок на записи активации. — *Прим. ред.*

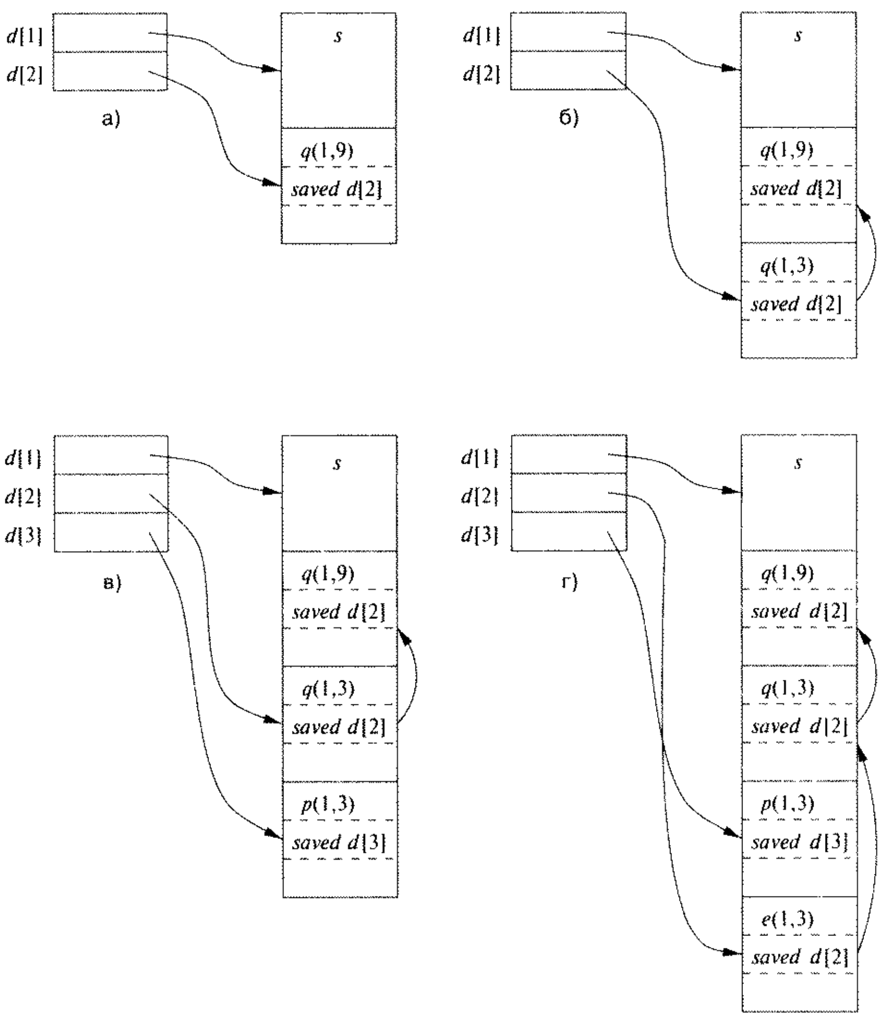


Рис. 7.14. Работа с дисплеем

указывающим на запись активации для *partition*. Запись активации для *partition* содержит место для хранения предыдущего значения $d\,[3]$, но в данном случае такового нет, так что указатель остается нулевым. Состояние стека и дисплея в этот момент показано на рис. 7.14, *в*.

После этого функция *partition* вызывает функцию *exchange*. Эта функция имеет глубину вложенности 2, так что ее запись активации хранит старый указатель $d\,[2]$, который вел к записи активации для *quicksort* $(1, 3)$. Заметим, что указатели дисплея "пересекаются", т.е. $d\,[3]$ указывает на место в стеке, находящееся ниже места, на которое указывает $d\,[2]$. Однако это вполне корректная ситуация —

функция *exchange* может обращаться только к своим данным и данным функции *sort* посредством указателя $d[1]$. □

7.3.9 Упражнения к разделу 7.3

Упражнение 7.3.1. На рис. 7.15 приведена функция `main` на языке программирования ML, которая вычисляет число Фибоначчи нестандартным образом. Функция `fib0` вычисляет n-е число Фибоначчи для любого $n \geqslant 0$. В нее вложена функция `fib1`, которая вычисляет n-е число Фибоначчи в предположении $n \geqslant 2$, а в `fib1` вложена функция `fib2`, вычисляющая n-е число Фибоначчи при $n \geqslant 4$. Обратите внимание, что ни функции `fib1`, ни функции `fib2` не требуется проверка базовых случаев. Покажите, как выглядит стек записей активации, получающийся при вызове `main`, в момент непосредственно перед первым возвратом (из `fib0(1)`). Укажите связи доступа для каждой из записей активации в стеке.

```
fun main () {
   let
       fun fib0(n) =
           let
               fun fib1(n) =
                   let
                       fun fib2(n) = fib1(n-1) + fib1(n-2)
                   in
                       if n >= 4 then fib2(n)
                       else fib0(n-1) + fib0(n-2)
                   end
           in
               if n >= 2 then fib1(n)
               else 1
           end
   in
       fib0(4)
   end;
```

Рис. 7.15. Вложенные функции, вычисляющие числа Фибоначчи

Упражнение 7.3.2. Предположим, что функции на рис. 7.15 реализованы с использованием дисплея. Покажите состояние дисплея в момент непосредственно перед первым выходом из `fib0(1)`. Укажите также сохраненные элементы дисплея в каждой из записей активации, находящихся в этот момент в стеке.

7.4 Управление кучей

Куча представляет собой часть памяти, использующуюся для размещения данных с неопределенным временем жизни, пока программа не удалит их. В то время как локальные переменные обычно становятся недоступными по окончании их процедур, многие языки программирования позволяют создавать объекты или иные данные, существование которых не привязано к активации создавшей их процедуры. Например, и C++, и Java имеют оператор `new` для создания объектов, которые (или указатели на которые) могут быть переданы от процедуры к процедуре, так что они продолжают существовать долгое время после того, как завершится создавшая их процедура. Такие объекты хранятся в куче.

В данном разделе мы рассмотрим *диспетчер памяти* (memory manager) — подсистему, которая выделяет и освобождает память в куче; она служит интерфейсом между прикладной программой и операционной системой. Для языков типа C или C++, которые освобождают блоки памяти *вручную* (т.е. путем явных инструкций программы, таких как `free` и `delete`), диспетчер памяти отвечает за реализацию освобождения памяти.

В разделе 7.5 мы рассмотрим *сборку мусора* (garbage collection) — процесс поиска в куче памяти, которая больше не нужна программе и может быть использована для хранения других данных. В языках наподобие Java освобождает память именно сборщик мусора. Там, где он используется, сборщик мусора представляет собой важную подсистему управления памятью.

7.4.1 Диспетчер памяти

Диспетчер памяти постоянно отслеживает свободную память в куче. Он выполняет две основные функции.

1. *Выделение памяти.* Когда программа запрашивает память для переменной или объекта[6], диспетчер создает в куче непрерывный блок памяти требуемого размера. По возможности запрос на выделение памяти удовлетворяется путем использования свободной памяти в куче; если же свободного блока требуемого размера нет, предпринимаются попытки увеличения кучи за счет получения блока виртуальной памяти от операционной системы. Если память оказывается исчерпанной, диспетчер передает соответствующую информацию прикладной программе.

2. *Освобождение памяти.* Диспетчер памяти возвращает освобожденную память в пул свободной памяти, так что она может затем использоваться для удовлетворения новых запросов на выделение памяти. Диспетчер памяти

[6]Мы говорим далее обо всем, что требует выделения памяти, как об "объектах", несмотря на то, что это не обязательно объекты в смысле объектно-ориентированного программирования.

обычно не возвращает память операционной системе даже при снижении или полном прекращении использования кучи.

Управление памятью оказывается более простым, если *а*) все запросы на выделение памяти оказываются запросами на выделение блоков одного и того же размера и *б*) память освобождается предсказуемо, скажем, раньше освобождается та память, которая была раньше выделена. Имеются языки, такие, как Lisp, в которых выполняется требование *а*. Чистый Lisp использует только один элемент данных — ячейку с двумя указателями, — из которого строятся все структуры данных. Условие *б* также выполняется в некоторых ситуациях; наиболее распространенный случай — данные, память для которых может быть выделена в стеке времени выполнения. Однако в большинстве языков программирования в общем случае не выполняются ни условие *а*, ни условие *б*. Как правило, происходит выделение памяти для объектов разных размеров, причем не существует хорошего способа предсказать времена жизни всех объектов.

Таким образом, диспетчер памяти должен быть готов обслуживать в произвольном порядке запросы на выделение и освобождение памяти любого размера, от одного байта до всего адресного пространства программы.

Вот какими свойствами должен обладать хороший диспетчер памяти.

- *Эффективное использование памяти.* Диспетчер памяти должен минимизировать общий размер кучи, требующейся программе. Это позволяет большим программам выполняться в фиксированном виртуальном адресном пространстве. Пространственная эффективность достигается путем минимизации "фрагментации", рассматривающейся в разделе 7.4.4.

- *Программная эффективность.* Диспетчер памяти должен так использовать подсистему памяти, чтобы программа выполнялась как можно быстрее. Как мы увидим в разделе 7.4.2, время выполнения команды может существенно зависеть от места размещения объекта в памяти. К счастью, обычно программы проявляют тенденцию к "локальности", явлению, которое будет рассмотрено в разделе 7.4.3 и которое означает неслучайную кластеризацию при обращениях типичной программы к памяти. Уделяя внимание размещению объектов в памяти, диспетчер сможет эффективнее использовать доступную память и, следует надеяться, повысить скорость работы программы.

- *Низкие накладные расходы.* Поскольку во многих программах выделение и освобождение памяти — весьма распространенные операции, очень важно, чтобы они были максимально эффективны и минимизировали накладные расходы — долю времени выполнения, затрачиваемого на операции выделения и освобождения памяти. Заметим, что стоимость выделения памяти доминирует при малых запросах; для больших запросов она не столь

существенна, поскольку обычно амортизируется большим количеством вычислений.

7.4.2 Иерархия памяти компьютера

Управление памятью и оптимизация должны выполняться с учетом информации о поведении памяти. Современные машины разрабатываются так, чтобы программисты могли писать программы, не концентрируясь на деталях работы подсистемы памяти. Однако эффективность программы определяется не только количеством выполняемых команд, но и временем выполнения каждой команды. Время выполнения команды может изменяться в широких пределах, поскольку время доступа к различным частям памяти может варьироваться в пределах от наносекунд до миллисекунд. Таким образом, программы, интенсивно работающие с данными, могут существенно выиграть от оптимизации, повышающей эффективность использования подсистемы управления памятью. Как будет видно в разделе 7.4.3, они могут воспользоваться преимуществами явления локальности — неслучайного поведения типичных программ.

Большая разница времен доступа к памяти связана с фундаментальными технологическими ограничениями в области электроники. Можно создать небольшую и быструю память; можно — большую и медленную, но получить большую быструю память нереально. Сегодняшняя технология не позволяет получить гигабайтную память с наносекундным временем доступа, соответствующим скорости работы современных высокопроизводительных процессоров. Поэтому память практически всех современных компьютеров образует *иерархию памяти*. Иерархия памяти, как показано на рис. 7.16, состоит из ряда элементов памяти, в котором более быстрая небольшая память оказывается "ближе" к процессору, а большая и более медленная — дальше от него.

Обычно у процессора имеется небольшое количество регистров, содержимое которых управляется программно. Также есть один или несколько уровней кэшей, обычно находящихся вне статической памяти и имеющих размер от килобайтов до нескольких мегабайтов. Следующим уровнем иерархии памяти является физическая (основная) память, состоящая из сотен мегабайтов или гигабайтов динамической оперативной памяти (RAM). Физическая память "подпирается" виртуальной памятью, реализованной в виде гигабайтов дисковой памяти. При обращении к памяти машина сначала ищет данные в ближайшей памяти (на наинизшем уровне) и, если их там нет, переходит к следующему уровню, и т.д.

Регистров очень мало, так что их применение ограничено специфическими приложениями и управляется кодом, генерируемым компилятором. Все остальные уровни иерархии управляются автоматически. Это не только упрощает программирование, но и позволяет одной и той же программе эффективно работать на различных машинах с разными конфигурациями памяти. При каждом обращении

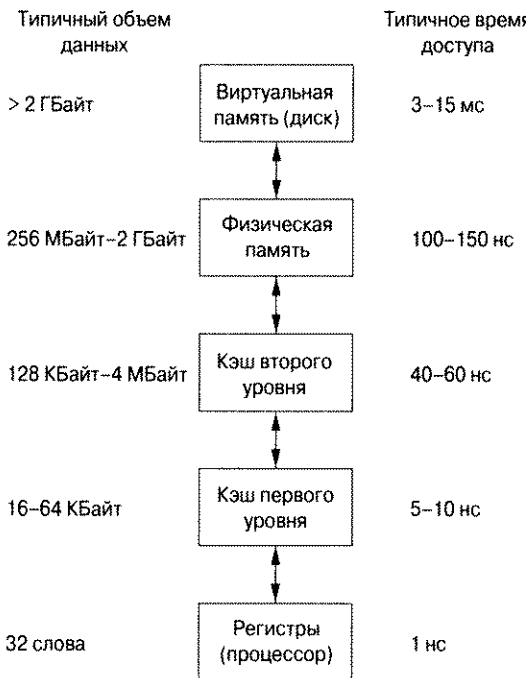


Рис. 7.16. Типичная конфигурация иерархии памяти

Статическая и динамическая оперативная память

В основном, память с произвольным доступом является *динамической*, что означает, что она построена из очень простых электронных схем, которые быстро теряют свой заряд (и "забывают" хранящийся в них бит). Такие схемы должны периодически обновляться, т.е. хранящиеся в них биты должны считываться и перезаписываться. *Статическая* же оперативная память разработана с использованием более сложных схем, и сохраненный бит хранится в них неограниченно долго, пока не будет изменен. Очевидно, что схема может хранить большее количество данных при использовании динамических элементов, чем при применении статических, так что обычно большая основная память компьютера — динамическая, в то время как меньшая по размеру память, например кэши, собирается из статических схем.

к памяти машина последовательно просматривает каждый уровень памяти, начиная с наинизшего уровня, до тех пор, пока не будут найдены искомые данные. Кэши управляются исключительно аппаратно, что позволяет обеспечить относи-

тельно быстрый доступ к оперативной памяти. Поскольку диски сравнительно медленны, виртуальная память управляется операционной системой при помощи аппаратной структуры, известной как буфер преобразования адреса (translation lookaside buffer).

Данные пересылаются блоками непрерывной памяти. Для амортизации стоимости доступа на более медленных уровнях доступа используются блоки большего размера. Между основной памятью и кэшем данные пересылаются блоками, известными под названием *строки кэша* (cache lines), обычно длиной 32–256 байт. Между виртуальной памятью (диском) и основной памятью данные передаются блоками, известными как *страницы* (pages), размер которых обычно составляет 4–64 Кбайт.

7.4.3 Локальность в программах

Большинство программ демонстрирует высокую степень *локальности* (locality), т.е. бóльшую часть времени они затрачивают на выполнение относительно небольших частей кода и обращаются только к небольшой части данных. Мы говорим, что программа обладает *временнóй локальностью* (temporal locality), если обращение к ячейкам памяти означает, что, скорее всего, через небольшой период времени к ним вновь будет выполнено обращение. Программа обладает *пространственной локальностью* (spatial locality), если, скорее всего, через небольшой промежуток времени будет выполнено обращение к ячейкам памяти, находящимся вблизи тех ячеек, к которым только что было выполнено обращение.

Традиционная мудрость гласит, что 90% времени программа затрачивает на выполнение 10% кода. Вот с чем это связано.

- Часто программы содержат команды, которые никогда не выполняются. Программы, собранные с компонентами и библиотеками, используют только малую часть предоставляемой ими функциональности. Кроме того, в процессе развития программ в них часто остается старый, больше не нужный и не выполняющийся код.

- При типичном запуске программы реально выполняется только малая часть кода, который может быть вызван. Например, команды обработки некорректного ввода и исключений хотя и критичны для корректности программы, на деле выполняются очень редко.

- Типичная программа больше всего времени затрачивает на выполнение внутренних циклов и рекурсию.

Локальность позволяет воспользоваться преимуществами иерархии памяти современного компьютера, показанной на рис. 7.16. Помещая чаще всего используемые команды и данные в быструю, хотя и меньшую, память и оставляя остальную

Архитектуры кэшей

Как узнать, находится ли строка кэша в кэше? Проверка каждой отдельной линии в кэше может оказаться слишком дорогим удовольствием, так что распространенная практика заключается в ограничении размещения строки кэша в кэше. Такое ограничение известно как *множественная ассоциативность*. Кэш обладает k-вариантной множественной ассоциативностью, если строка кэша может находиться только в k местах кэша. Простейший кэш — одновариантный, известный также как кэш с *прямым отображением*, в котором данные с адресом n в физической памяти могут размещаться в кэше только по адресу $n \bmod s$, где s — размер кэша. Аналогично k-вариантный кэш разбивается на k множеств, и данные с адресом n могут отображаться только в позиции $n \bmod (s/k)$ в каждом множестве. Большинство кэшей команд и данных имеют ассоциативность от 1 до 8. Когда строка кэша помещается в кэш, в котором все возможные позиции для этой строки заняты, обычно из кэша удаляется строка, которая не использовалась дольше всех.

часть программы в большей (и медленной) памяти, можно существенно снизить среднее время обращения программы к памяти.

Обнаружено, что многие программы демонстрируют как временну́ю, так и пространственную локальность при обращении как к командам, так и к данным. Обращение к данным, тем не менее, демонстрирует большее разнообразие, чем обращение к коду. Стратегии, такие как хранение использованных последними данных в наиболее быстрой памяти иерархии, обычно неплохо работают для большинства программ, но могут быть непригодными для некоторых программ, интенсивно работающих с данными, например для циклически проходящих по очень большим массивам.

Чаще всего по внешнему виду кода невозможно сказать, будет ли он активно использоваться при некоторых конкретных входных данных. Но даже если известен наиболее активно используемый код, размера наиболее быстрого кэша часто недостаточно для того, чтобы полностью разместить этот код. Поэтому приходится динамически обновлять содержимое кэша и использовать его только для хранения тех команд, активная работа которых наиболее вероятна в ближайшее время.

Оптимизация с использованием иерархии памяти

Стратегия хранения последних использованных команд в кэше обычно неплохо работает; другими словами, в общем случае прошлое — неплохой предсказатель будущего использования памяти. При выполнении новой команды весьма высока

вероятность, что следующая за ней команда также будет выполнена. Это явление – пример пространственной локальности. Один эффективный метод повышения пространственной локальности состоит в том, что компилятор размещает базовые блоки (последовательности команд, всегда выполняющихся одна за другой), которые, скорее всего, будут выполняться последовательно один за другим так, чтобы они оказывались в одной странице или по возможности даже в одной строке кэша. Команды из одного цикла или одной функции также имеют высокую вероятность совместного выполнения.[7]

Можно повысить временную и пространственную локальность обращения к данным в программе путем изменения схемы размещения данных или порядка вычислений. Например, программа, которая многократно обходит большие количества данных, выполняя каждый раз небольшие вычисления, неэффективна. Было бы лучше, если бы можно было перенести некоторые данные из медленного уровня иерархии памяти в быстрый (например, с диска в оперативную память) однократно и выполнить над ними все необходимые действия, пока они находятся на более высоком уровне. Эта концепция рекурсивно применима к использованию данных на всех уровнях – в физической памяти, кэшах и регистрах.

7.4.4 Снижение фрагментации

Когда программа начинает выполняться, куча представляет собой один непрерывный блок свободной памяти. Когда программа запрашивает и освобождает память, весь объем кучи разбивается на свободные и используемые блоки памяти, причем свободные блоки вовсе не образуют одной непрерывной области в куче. При каждом запросе на выделение памяти диспетчер должен поместить запрошенный блок памяти в свободный блок достаточно большого размера. Если свободный блок с размером, в точности равным запрашиваемому, не найден, необходимо разбить некоторый блок большего размера, создавая при этом как блок для запрошенной памяти, так и свободный блок меньшего размера.

При каждом запросе на освобождение памяти освобождаемый блок возвращается в пул свободной памяти. Смежные свободные блоки сливаются, поскольку иначе блоки будут становиться все меньше и меньше. Если не проявить осмотрительности, свободная память может стать сильно *фрагментированной*, состоящей из большого количества мелких несмежных свободных блоков. В таком случае легко может оказаться, что для очередного запроса на выделение памяти не найдется свободного блока достаточного размера, хотя суммарное количество свободной памяти может при этом во много раз превышать запрашиваемое.

---

[7]Когда машина осуществляет выборку слова из памяти, можно относительно недорого осуществить одновременно *предвыборку* нескольких последовательно идущих слов. Поэтому распространенной особенностью иерархии памяти является выборка блоков слов из каждого уровня памяти при обращении к нему.

Методы наилучшего подходящего и следующего подходящего

Мы снижаем фрагментацию, управляя методом размещения объектов в куче диспетчером памяти. Эмпирически было обнаружено, что хорошей стратегией минимизации фрагментации в реальных программах является выделение запрошенной памяти в наименьшем из блоков, размеры которых достаточно велики для такого выделения. Такой алгоритм *наилучшего подходящего* (best-fit) стремится к сохранению больших свободных блоков для удовлетворения могущих последовать запросов на выделение большого количества памяти. Альтернативный метод *первого подходящего* (first-fit) помещает объект в первый (с наименьшим адресом) свободный блок достаточного размера, тратя, таким образом, меньше времени на выделение памяти, но проигрывая в суммарной эффективности методу наилучшего подходящего.

Для более эффективной реализации размещения методом наилучшего подходящего можно разделить блоки свободной памяти на *карманы* (bin) в соответствии с их размерами. Одна из применяющихся на практике идей состоит в том, чтобы иметь побольше карманов малых размеров, поскольку обычно в программах создается гораздо больше мелких объектов, чем больших. Например, диспетчер памяти Lea, использующийся в компиляторе GNU C `gcc`, выравнивает все блоки на 8-байтовые границы. Имеется по одному карману для блоков каждого кратного 8 байт размера в пределах от 16 до 512 байт. Карманы бóльших размеров имеют логарифмические промежутки, т.е. минимальный размер каждого кармана вдвое больше предыдущего, а внутри каждого из этих карманов блоки упорядочены в соответствии с их размерами. Всегда существует блок свободной памяти, который может быть расширен путем запроса у операционной системы дополнительных страниц. Этот блок — так называемый "пустынный" (wilderness) — рассматривается Lea как карман наибольшего размера в силу его расширяемости.

Такая схема позволяет легко найти наиболее подходящий блок.

- Если у диспетчера запрошен блок памяти малого размера и существует карман с блоками только этого размера, можно выделить в ответ на запрос любой блок из этого кармана.

- Если для данного размера кармана не существует, мы ищем карман, который включает блоки требуемого размера. Внутри такого кармана можно использовать стратегию первого подходящего или наилучшего подходящего; т.е. мы либо ищем и выбираем первый достаточно большой блок, либо тратим немного больше времени и находим среди блоков достаточного размера наименьший. Заметим, что, если размер блока не совпадает с размером выделенной памяти, остаток блока, вообще говоря, следует поместить в карман с блоками меньшего размера.

- Однако может оказаться, что целевой карман пуст или все блоки в таком кармане слишком малы для удовлетворения запроса. В этом случае поиск повторяется в кармане для следующего большего размера (или размеров). В конце концов мы либо находим блок, который можем использовать, либо достигаем "пустынного" блока, из которого можем получить требуемую память, возможно, воспользовавшись услугами операционной системы для получения дополнительных страниц памяти для кучи.

Хотя стратегия наилучшего подходящего и повышает степень использования памяти, она может оказаться не лучшей с точки зрения пространственной локальности. Обычно выделенные примерно в одно и то же время блоки имеют примерно одно время жизни и одинаковую схему обращений к ним. Их размещение рядом друг с другом в памяти повышает пространственную локальность программы. Алгоритм наилучшего подходящего можно слегка модифицировать с учетом сказанного для ситуации, когда блок в точности совпадающего с запросом размера не найден. В этом случае используется стратегия *следующего подходящего* (next-fit), которая пытается выделить память сначала в том блоке, который только что был разделен, если, конечно, в оставшейся свободной части блока достаточно памяти для удовлетворения запроса. Такая стратегия одновременно имеет тенденцию к повышению скорости выделения памяти.

Управление свободной памятью и ее слияние

Когда память, выделенная объекту, освобождается вручную, диспетчер памяти должен сделать соответствующий блок памяти свободным, чтобы он мог быть выделен снова. В ряде случаев оказывается возможным объединить (слить) этот блок с соседним блоком в куче, чтобы получить блок большего размера. Преимущество такого подхода заключается в том, что всегда можно использовать большой блок для выделения малого количества памяти, но много малых блоков не могут хранить большой объект.

При наличии кармана для блоков одного фиксированного размера (как в диспетчере Lea для малых размеров) может не иметь смысла объединять соседние свободные блоки в блок удвоенного размера. В этом случае схема выделения и освобождения памяти предельно проста: достаточно иметь битовую карту, в которой каждому блоку кармана соответствует один бит. 1 указывает, что блок занят; 0 — что блок свободен. При освобождении блока достаточно просто установить соответствующий бит равным 0. При выделении памяти выполняется поиск блока с нулевым битом, бит изменяется на 1 и данный блок возвращается в качестве выделенного. Если свободных блоков нет, можно запросить у операционной системы новую страницу памяти, разделить ее на блоки соответствующего размера и расширить битовый вектор.

Ситуация усложняется при управлении кучей в целом, без разбивки на корзины, или при слиянии смежных блоков с переносом при необходимости получившегося блока в другую корзину. Имеются две структуры данных, поддерживающих слияние смежных свободных блоков.

- *Дескрипторы границ.* С обоих концов каждого блока, как свободного, так и выделенного, хранится важная информация. На каждом конце блока имеется бит, указывающий, свободен ли данный блок. Рядом с этим битом указывается размер блока в байтах.

- *Двухсвязный встроенный список свободных блоков.* Свободные (но не выделенные!) блоки связываются в дважды связанный список. Указатели этого списка хранятся в самих блоках, например, по соседству с дескрипторами границ с любой стороны. В этом случае нет дополнительных расходов памяти на организацию списка, хотя при этом его наличие определяет нижнюю границу размера свободного блока; такой блок должен содержать два дескриптора границ и два указателя, даже если объект состоит из одного байта. Порядок блоков в списке остается неопределенным. Например, список может быть отсортирован по размеру, что облегчает использование стратегии наилучшего подходящего.

Пример 7.10. На рис. 7.17 показана часть кучи с тремя смежными блоками A, B и C. Блок B размером 100 только что освобожден и возвращен в список свободных блоков. Поскольку известно начало (левый конец) блока B, также известен конец блока, непосредственно примыкающего к B слева, т.е. блока A в нашем примере. Бит использования блока на правом конце A равен 0, т.е. блок A также свободен. Следовательно, блоки A и B можно слить в один блок длиной 300 байт.

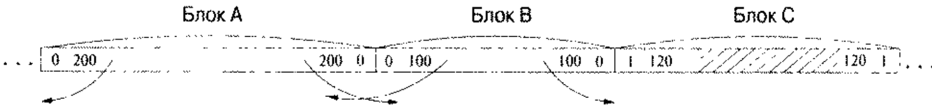


Рис. 7.17. Часть кучи и дважды связанный список свободных блоков

Может оказаться, что блок C, примыкающий к B справа, также свободен; в этом случае можно объединить все три блока — A, B и C. Заметим, что если при освобождении блока всегда выполняется его слияние, то в ходе этого процесса не надо просматривать иные блоки, кроме двух смежных с освобождаемым. В нашем случае найти блок C можно, зная левый конец блока B и общее количество байтов в блоке (известное из дескриптора границы). После этого можно проверить, свободен ли блок C (рассматривая соответствующий бит в его дескрипторе границы). В нашем случае этот бит равен 1, так что блок C занят и не может быть слит с блоком B.

Поскольку мы сливаем блоки A и B, нам надо удалить один из них из списка свободных блоков. Структура дважды связанного списка позволяет легко определить блоки перед и после A и B. Заметим, что "физические" соседи A и B не обязаны быть соседями в списке свободных блоков. Зная предшествующие и последующие блоки A и B, легко выполнить необходимые действия с указателями для замещения A и B одним объединенным свободным блоком. □

Автоматическая сборка мусора может устранить фрагментацию путем перемещения всех выделенных объектов в один непрерывный блок. Взаимодействие сборки мусора и управления памятью более подробно рассматривается в разделе 7.6.4.

7.4.5 Освобождение памяти вручную

Завершим этот раздел управлением памятью вручную, когда программист должен явно освободить используемую память, как это делается в языках программирования C и C++. В идеальном случае любая память, которая больше не нужна, должна быть освобождена. И наоборот, любая память, к которой может быть выполнено обращение, не должна освобождаться. К сожалению, и то, и другое трудно выполнить. Наряду с рассмотрением трудностей ручного освобождения памяти будут описаны и некоторые методы их преодоления.

Проблемы освобождения памяти вручную

Освобождение памяти вручную может приводить к ошибкам. Наиболее распространены ошибки двух видов: "неудаление" данных, обращение к которым более невозможно, называемое *утечкой памяти*, и обращение к удаленным данным, называемое ошибкой *разыменования висящего указателя*.

Программистам трудно сказать, действительно ли программа в будущем *никогда* не обратится к некоторым данным, так что первая распространенная ошибка состоит в том, что не освобождается память, к которой больше не будет обращений. Заметим, что хотя утечки памяти могут замедлить выполнение программы из-за возрастания количества требующейся ей памяти, они не влияют на корректность программы, пока память не окажется исчерпанной. Многие программы могут выдержать утечки памяти, особенно если они невелики. Однако для программ с большим временем работы, в особенности для программ, работающих безостановочно, наподобие операционных систем или серверного кода, отсутствие утечек становится критичным.

Автоматическая сборка мусора позволяет избавиться от утечек путем освобождения более не используемой памяти. Но даже в этом случае программа может использовать памяти больше, чем это необходимо. Программист может знать, что обращений к объекту больше не будет, несмотря на наличие в каком-то месте про-

граммы ссылки на него. В таком случае программист обязан удалить эту ссылку, чтобы объект мог быть удален автоматически.

Слишком усердно удаляя объекты, можно вызвать еще большие проблемы. Вторая распространенная ошибка состоит в удалении объекта, к которому позже пытается обратиться программа. Указатель на освобожденную память известен под названием *висящий указатель* (dangling pointer). После того, как освобожденная память вновь выделена новому объекту, операции чтения, записи и освобождения памяти с использованием висящего указателя могут привести к самым непредсказуемым результатам. Освобождение памяти с использованием висящего указателя означает, что память новой переменной может использоваться для размещения еще одной переменной, и обращения к старой и новой переменным будут конфликтовать друг с другом.

В отличие от утечки памяти разыменование висящего указателя после того, как освобожденная память выделена заново, почти всегда приводит к ошибке в программе, которую к тому же очень сложно отладить. В результате программисты не спешат удалять объекты, пока не становятся абсолютно уверены, что они больше не потребуются.

Еще одна ошибка на ту же тему — доступ по некорректному адресу. Распространенными примерами этой ошибки являются разыменование нулевого указателя и обращение к элементам за границей массива. Лучше, если такие ошибки проявляются сразу, чем если они тихонько портят результат работы программы. В действительности многие нарушения защиты используют такие программные ошибки, когда некоторая программа допускает непредусмотренное обращение к данным, что позволяет хакерам получить управление программой и машиной. Противоядием для этого являются проверки, вносимые компилятором для каждого обращения к памяти, чтобы убедиться в том, что все обращения происходят внутри разрешенных границ. Оптимизатор может обнаружить и удалить те проверки, для которых он в состоянии убедиться, что все обращения будут выполняться в пределах границ, так что эти проверки реально не нужны.

Программные соглашения и инструментарий

Теперь мы представим несколько популярных соглашений и инструментов, разработанных для того, чтобы помочь программистам в нелегком деле управления памятью.

- *Владение объектом* (object ownership) полезно, если вопрос о времени жизни объекта может быть решен статически. Идея заключается в том, чтобы на все время жизни связать с каждым объектом его *владельца*. Владелец представляет собой указатель на объект, предположительно принадлежащий некоторому вызову функции. Владелец (т.е. его функция) отвечает за удаление объекта либо передачу его другому владельцу. Можно иметь

Пример: Purify

Purify — один из многих популярных коммерческих инструментов, которые помогают программистам обнаруживать ошибки обращения к памяти и утечки памяти в программах. Purify добавляет к бинарному коду дополнительные команды для выявления ошибок во время работы программы. Этот инструмент отслеживает карту памяти с указанием свободных и занятых блоков. Каждый объект в памяти окружен дополнительной памятью. Обращение к не выделенной объекту памяти или памяти между объектами рассматривается как ошибка. Такой подход позволяет выявить ряд обращений по висящим указателям, но не в ситуации, когда объект был удален и его место занял другой объект. Данный инструментарий находит также обращения за пределы массива — если оно выполняется к дополнительной памяти, добавленной им в конце объекта.

Purify обнаруживает и утечки памяти по окончании работы программы. Он просматривает содержимое всех распределенных объектов в поисках возможных значений указателей. Любой объект, на который не указывает ни один указатель, является потерянным блоком памяти. Purify сообщает о количестве потерянной памяти и расположении потерянных объектов. В этом плане можно сравнить Purify с “консервативным сборщиком мусора”, который будет рассмотрен в разделе 7.8.3.

и другие, не владеющие объектом, указатели на него; такие указатели могут в любой момент быть перезаписаны и их значения не должны участвовать в удалениях объектов. Такое соглашение устраняет как утечки памяти, так и попытки многократного удаления одного и того же объекта. Однако оно не спасает от висящих указателей, поскольку возможно обращение к удаленному объекту посредством указателя, не являющегося владельцем.

- *Подсчет ссылок* (reference counting) полезен в ситуации, когда время жизни объекта должно быть определено динамически. Идея заключается в связывании с каждым динамически выделенным объектом счетчика. При создании ссылки на объект значение счетчика ссылок увеличивается; при удалении ссылки значение счетчика уменьшается. Когда значение счетчика обнуляется, обращение к объекту становится невозможным и, следовательно, объект может быть удален. Однако такой метод не в состоянии выявить зацикленную структуру без внешних ссылок на нее (обращение к такой структуре невозможно из-за отсутствия в программе ссылок на нее, но и удаление ее также невозможно, поскольку части структуры вза-

имно ссылаются друг на друга, так что счетчики не обнулены). Пример такой структуры приведен в примере 7.11. Подсчет ссылок решает проблему с висящими указателями, поскольку указателей на удаленный объект не остается. Однако это достаточно дорогой метод, поскольку приводит к накладным расходам для каждой операции по сохранению указателя.

- *Выделение памяти областями* (region-based allocation) используется для наборов объектов, времена жизни которых связаны с некоторой фазой вычислений. Когда объекты создаются только для использования на некотором шаге вычислений, можно выделить память для них в одной области, которая затем, по завершении этого шага, будет освобождена. Такой метод распределения памяти имеет ограниченное применение, но там, где он может использоваться, он весьма эффективен. Вместо удаления объектов по одному он удаляет всю область целиком.

7.4.6 Упражнения к разделу 7.4

Упражнение 7.4.1. Предположим, что куча состоит из семи блоков, начиная с адреса 0. Размеры блоков, начиная с младшего адреса, – 80, 30, 60, 50, 70, 20 и 40 байт. При размещении объекта в блоке (если позволяет размер последнего) он помещается в старших адресах блока и образует меньший блок. Однако размеры блоков не могут быть меньше 8 байт, так что если объект почти такого же размера, как и блок, в котором он располагается, то объекту выделяется весь блок полностью, а сам объект располагается в младших адресах блока. Как будет выглядеть список свободной памяти после удовлетворения запросов на 32, 64, 48 и 16 байт в указанном порядке, если используется метод

а) первого подходящего;

б) наилучшего подходящего.

7.5 Введение в сборку мусора

Данные, обращение к которым невозможно, в общем случае известны как *мусор* (garbage). Многие высокоуровневые языки программирования снимают бремя управления памятью вручную с плеч программиста путем использования автоматической сборки мусора, которая освобождает память, занятую недостижимыми данными. Возникновение сборки мусора датируется 1958 годом, когда она была применена в начальной реализации Lisp. Другими важными примерами языков программирования, использующих сборку мусора, являются Java, Perl, ML, Modula-3, Prolog и Smalltalk.

В этом разделе мы рассмотрим ряд концепций сборки мусора. Понятие "достижимости" объекта, пожалуй, интуитивно понятное, но нам следует быть точными. Точные правила рассматриваются в разделе 7.5.2. В разделе 7.5.3 мы рассмотрим простой, но несовершенный метод сборки мусора — подсчет ссылок, основанный на той идее, что как только программа теряет все ссылки на объект, она уже не может обратиться к занимаемой им памяти.

В разделе 7.6. рассматриваются сборщики, основанные на отслеживании, алгоритмы которых определяют объекты, к которым возможны обращения, и превращают все остальные блоки кучи в свободные.

7.5.1 Цели проектирования сборщиков мусора

Сборка мусора представляет собой утилизацию блоков памяти, в которых хранятся объекты, более не требующиеся программе. Мы должны считать, что объекты имеют тип, который сборщик мусора в состоянии определить во время выполнения программы. Исходя из типа объекта можно определить, насколько велик сам объект и какие его компоненты содержат ссылки (указатели) на другие объекты. Мы также считаем, что ссылки на объекты всегда указывают на адрес начала объекта и никогда — на место внутри объекта. Таким образом, все ссылки на объект имеют одно и то же значение и могут быть легко идентифицированы.

Пользовательская программа, о которой в дальнейшем мы будем говорить как о *мутаторе* (mutator), модифицирует набор объектов в куче. Мутатор создает объекты, запрашивая пространство для них у диспетчера памяти, и, кроме того, может создавать и удалять ссылки на существующие объекты. Объекты становятся мусором, когда мутатор не в состоянии "достичь" его в том смысле, который будет точно определен в разделе 7.5.2. Сборщик мусора находит недостижимые объекты и утилизирует отведенное им пространство, передавая его диспетчеру памяти, который отслеживает свободную память в куче.

Основное требование: безопасность типов

Не все языки программирования являются хорошими кандидатами для автоматической сборки мусора. Чтобы сборщик мусора корректно работал, он должен быть в состоянии сообщить, является ли любой данный элемент данных или его компонент указателем (или может использоваться в качестве такового) на блок выделенной памяти. Язык, в котором можно определить тип любого компонента данных, называется *безопасным с точки зрения типов* (type safe). Существуют безопасные с точки зрения типов языки наподобие ML, в котором типы могут быть определены во время компиляции. Имеются и другие безопасные языки наподобие Java, типы которого не могут быть определены во время компиляции, но могут быть определены во время выполнения. Такие языки называются *динамически типизированными* (dynamically typed). Если язык не является ни статически,

ни динамически безопасным с точки зрения типов, говорят, что он *небезопасен* (unsafe).

Небезопасные языки программирования, которые, к сожалению, включают такие важные языки, как C и C++, являются плохими кандидатами для автоматической сборки мусора. В небезопасных языках допускается произвольная работа с адресами — к указателям могут применяться арифметические операции для создания новых указателей; произвольные целые числа могут рассматриваться как указатели. Таким образом, программа теоретически может в любой момент времени обратиться к любой ячейке памяти. Следовательно, ни одна ячейка памяти не может рассматриваться как недостижимая и никакая память не может быть безопасно утилизирована.

На практике большинство программ на языках программирования C и C++ не генерируют указатели произвольным образом, и можно разработать теоретически ненадежный сборщик мусора, который, тем не менее, эмпирически будет хорошо работать. Мы рассмотрим консервативный сборщик мусора для C и C++ в разделе 7.8.3.

Критерии производительности

Сборка мусора часто настолько дорогостояща, что, хотя она и изобретена десятилетия назад и абсолютно предотвращает утечки памяти, она все еще должна быть принятой многими широко распространенными языками программирования. За время существования сборки мусора предложена масса различных подходов, но ни один из них не может претендовать на звание наилучшего алгоритма сборки мусора. Перед рассмотрением различных вариантов давайте сначала перечислим критерии производительности, которые должны учитываться при проектировании сборщика мусора.

- *Общее время работы*. Сборка мусора может быть очень медленной. Очень важно, чтобы она не увеличивала существенно общее время работы приложения. Поскольку сборщик мусора должен обойти огромное количество данных, его производительность сильно зависит от того, как он использует подсистему памяти.

- *Использование памяти*. Очень важно, чтобы сборщик мусора устранял фрагментацию и позволял эффективнее использовать доступную память.

- *Работа в паузах*. Простые сборщики мусора знамениты тем, что заставляют программы — мутаторы — внезапно, безо всякого предупреждения приостанавливаться на длительное время, пока не будет выполнена сборка мусора. Таким образом, помимо минимизации времени работы, желательно, чтобы максимальная пауза в работе основной программы была минимизирована. В качестве важного частного случая программы, работающие

в реальном времени, требуют, чтобы определенные вычисления укладывались в отведенные для них временны́е рамки. Следует либо подавлять работу сборщика мусора в программах реального времени, либо ограничивать время максимальной паузы. Таким образом, сборщики мусора редко используются в приложениях реального времени.

- *Локальность программ.* Мы не можем оценить скорость работы сборщика мусора только по его времени работы. Сборщик мусора управляет размещением данных и, таким образом, влияет на локальность данных в мутаторе. Он может повысить временну́ю локальность мутатора, освобождая память и повторно используя ее; он может повысить пространственную локальность мутатора, перемещая совместно используемые данные в один и тот же кэш или страницы.

Некоторые из перечисленных целей проектирования конфликтуют друг с другом, и компромисс может быть достигнут путем тщательного рассмотрения типичного поведения программы. Кроме того, объекты с разными характеристиками могут потребовать различной обработки, с применением разных методов для объектов разного вида.

Например, среди выделенных объектов обычно доминируют объекты малого размера, так что выделение памяти для малых объектов не должно приводить к большим накладных расходам. С другой стороны, рассмотрим сборщик мусора, который переносит достижимые объекты. Перенос больших объектов — более дорогостоящая операция, чем перенос малых.

Еще один пример: вообще говоря, чем дольше мы ждем вызова сборщика, основанного на отслеживании, тем больше становится объектов, которые могут быть собраны. Причина в том, что объекты часто "умирают молодыми", так что если немного подождать, то многие из вновь распределенных объектов станут недостижимыми. Стоимость работы такого сборщика в среднем на один собранный объект оказывается меньшей. С другой стороны, редкая сборка мусора повышает расход памяти, уменьшает локальность данных и увеличивает продолжительность пауз.

В противоположность этому сборщик с подсчетом ссылок, внося постоянные накладные расходы для многих операций мутатора, может существенно замедлить работу программы в целом. Но, с другой стороны, подсчет ссылок не приводит к длинным паузам и эффективно использует память, поскольку мусор обнаруживается в момент его образования (за исключением определенных циклических структур, рассматривающихся в разделе 7.5.3).

Дизайн языка программирования также влияет на характеристики использования памяти. Некоторые языки способствуют стилю программирования, генерирующему много мусора. Например, программы на функциональных, или почти функциональных, языках программирования создают большое количество объек-

тов, чтобы избежать изменения уже существующих объектов. В Java все объекты не фундаментальных типов (наподобие целых чисел или ссылок) создаются в куче, а не в стеке, даже если их время жизни ограничено единственным вызовом функции. Такой дизайн освобождает программиста от забот о времени жизни переменных ценой повышенной генерации мусора. Оптимизация в процессе компиляции может проанализировать время жизни переменной и, если это возможно, выделить для нее память в стеке.

7.5.2 Достижимость

Мы говорим обо всех данных, которые могут быть доступны непосредственно из программы, без разыменования какого-либо указателя, как о *корневом множестве* (root set). Например, в Java корневое множество программы состоит из всех статических полей-членов и всех переменных в стеке. Очевидно, что программа в любой момент времени может достичь любого элемента корневого множества. Рекурсивно достижим любой объект, ссылка на который хранится в поле-члене или элементе массива достижимого объекта.

Достижимость существенно усложняется при оптимизации программы компилятором. Во-первых, компилятор может хранить ссылочную переменную в регистре. Такие ссылки также могут рассматриваться как часть корневого множества. Во-вторых, хотя безопасные с точки зрения типов языки и не работают с адресами памяти непосредственно, компилятор часто делает это для ускорения кода. Таким образом, регистры в скомпилированном коде могут указывать на средину объекта или содержать значение, к которому следует добавить некоторое смещение для получения корректного адреса. Вот что компилятор может делать для того, чтобы обеспечить возможность сборщику мусора корректно определить корневое множество.

- Компилятор может ограничить вызов сборщика мусора только определенными точками в программе, в которых нет "скрытых" ссылок.

- Компилятор может записывать информацию, которая используется сборщиком мусора для восстановления всех ссылок, такую как указание, какие регистры содержат ссылки или как вычислить базовый адрес объекта по его внутреннему адресу.

- Компилятор может обеспечить существование ссылок на базовые адреса всех достижимых объектов в момент вызова сборщика мусора.

Множество достижимых объектов изменяется в процессе выполнения программы. Оно растет при создании новых объектов и уменьшается, когда объекты становятся недостижимы. Важно помнить, что когда объект становится недостижимым, он не может стать достижимым впоследствии. Есть четыре фунда-

ментальные операции, выполняемые мутатором, которые приводят к изменению множества достижимых объектов.

- *Выделение памяти для объекта.* Эта операция выполняется диспетчером памяти, который возвращает ссылку на каждый выделенный блок памяти. Эта операция добавляет члены в множество достижимых объектов.

- *Передача параметров и возврат значений.* Ссылки на объекты передаются из фактических входных параметров в соответствующие формальные параметры, а также из возвращаемого результата обратно в вызывающую процедуру. Объекты, на которые указывают эти ссылки, остаются достижимыми.

- *Присваивание ссылок.* Присваивание вида `u = v`, где u и v являются ссылками, имеет два результата. Во-первых, u становится ссылкой на объект, на который указывает v. Если достижима переменная u, то, конечно же, достижим и объект, на который он указывает. Во-вторых, теряется исходная ссылка u. Если это была последняя ссылка на некоторый достижимый объект, то этот объект становится недостижимым. Всякий раз, когда объект становится недостижимым, становятся недостижимыми и все объекты, достижимые только посредством ссылок в нем.

- *Возврат из процедуры.* При выходе из процедуры кадр, в котором хранятся ее локальные переменные, снимается со стека. Если кадр хранит единственные достижимые ссылки на некоторый объект, то этот объект становится недостижимым. И вновь, если недостижимый теперь объект хранил единственные ссылки на некоторые другие объекты, те тоже становятся недостижимыми, и этот процесс продолжается далее по цепочке.

Итак, новые объекты вводятся в программу путем выделения для них памяти. Передача параметров и присваивание могут распространять достижимость; присваивание и завершение процедуры могут прекращать достижимость. Когда некоторый объект становится недостижимым, он может сделать таковыми некоторые другие объекты.

Имеется два фундаментальных способа поиска недостижимых объектов: либо перехватываются превращения достижимых объектов в недостижимые, либо периодически выявляются все достижимые объекты, после чего все прочие объекты рассматриваются как недостижимые. *Подсчет ссылок*, упоминавшийся в разделе 7.4.5, представляет собой хорошее приближение первого подхода. Поддерживается счетчик ссылок на объект, отслеживающий действия мутатора, которые могут изменить множество достижимых объектов. Когда счетчик обнуляется, объект становится недостижимым. Этот подход более детально рассматривается в разделе 7.5.3.

Жизнь и смерть стековых объектов

При вызове процедуры указатели на локальную переменную v, объект которой находится в стеке, могут храниться в нелокальных переменных. Эти указатели продолжают существовать после возврата из процедуры, в то время, когда память для v уже освобождена, что приводит к появлению висящей ссылки. Должны ли мы вообще располагать локальные переменные наподобие v в стеке, как это делается, например, в C? Ответ заключается в том, что семантика многих языков программирования *требует*, чтобы локальные переменные прекращали существование при завершении их процедур. Ссылки на такие переменные представляют собой ошибки программирования, и компилятор не обязан исправлять их в программе.

Второй подход определяет достижимость путем транзитивного отслеживания всех ссылок. Сборщик мусора *на основе отслеживания* (trace-based) начинает с того, что помечает все объекты корневого множества как "достижимые", итеративно просматривает все ссылки в достижимых объектах в поисках других достижимых объектов и помечает их соответствующим образом. При таком подходе перед тем, как считать некоторый объект недостижимым, должны быть отслежены все ссылки. Однако после того, как множество достижимых объектов вычислено, за один раз могут быть как обнаружены все недостижимые объекты, так и освобождена занимаемая ими память. Поскольку все ссылки анализируются в одно и то же время, имеется возможность перенесения объектов и снижения тем самым фрагментации памяти. Существует множество различных алгоритмов отслеживания, и некоторые из них будут рассматриваться в разделах 7.6 и 7.7.1.

7.5.3 Сборщики мусора с подсчетом ссылок

Теперь рассмотрим простой, хотя и несовершенный сборщик мусора, основанный на подсчете ссылок, который идентифицирует мусор, как объект, состояние которого меняется с достижимого на недостижимое. Объект может быть удален, когда количество ссылок на него падает до нуля. В случае сборщика мусора с подсчетом ссылок каждый объект должен иметь поле для количества ссылок, работа с которыми может осуществляться следующим образом.

1. *Создание объекта.* Количество ссылок вновь созданного объекта устанавливается равным 1.

2. *Передача параметров.* Значение счетчика ссылок каждого передаваемого в процедуру объекта увеличивается на 1.

3. *Присваивание ссылок.* В случае инструкции u = v, где u и v — ссылки, значение счетчика ссылок объекта, на который указывает v, увеличивается на 1, а объекта, на который указывала переменная u до присваивания, уменьшается на 1.

4. *Возврат из процедуры.* При возврате из процедуры должны быть уменьшены на 1 значения счетчиков объектов, на которые ссылались локальные переменные в записи активации этой процедуры. Если ссылка на один и тот же объект хранилась в нескольких локальных переменных, его значение счетчика должно быть уменьшено на 1 для каждой из переменных.

5. *Транзитивная потеря достижимости.* Когда значение счетчика ссылок некоторого объекта становится равным нулю, надо уменьшит значения счетчиков всех объектов, на которые указывают ссылки из исходного объекта.

Счетчики ссылок имеют два основных недостатка: они не позволяют собирать недостижимые циклические структуры данных и их поддержка дорогостояща в плане эффективности работы программы. Циклические структуры данных достаточно распространены; например, структуры данных часто указывают на родительские узлы или друг на друга, что приводит к перекрестным ссылкам.

Пример 7.11. На рис. 7.18 показаны три объекта со ссылками между ними, но при отсутствии ссылок откуда-либо еще. Если ни один из этих объектов не является частью корневого множества, то они представляют собой мусор, однако при этом значения счетчиков ссылок каждого из объектов больше 0. Такая ситуация при использовании подсчета ссылок для сборки мусора эквивалентна утечке памяти,

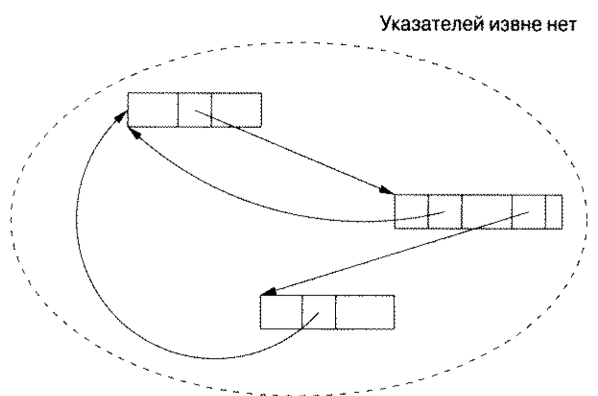


Рис. 7.18. Недостижимая циклическая структура данных

поскольку этот мусор, как и другие подобные структуры, никогда не будет удален из памяти. □

Накладные расходы на подсчет ссылок достаточно высоки из-за наличия дополнительных операций при каждом присваивании ссылок, входе в процедуру и выходе из нее. Эти накладные расходы пропорциональны количеству вычислений в программе, а не только количеству объектов в системе. В особенности это касается обновлений ссылок в корневом множестве программы. В качестве средства снижения накладных расходов, связанных с обновлением счетчиков при обращениях к локальному стеку, была предложена концепция *отложенного подсчета ссылок*. Иными словами, счетчики ссылок не включают ссылки от корневого множества программы. Объект не рассматривается как мусор до тех пор, пока не будет просканировано все корневое множество и при этом не будут обнаружены ссылки на этот объект.

Преимущества подсчета ссылок заключаются в том, что сборка мусора выполняется инкрементно. Несмотря на высокие общие накладные расходы, операции распределены по всей программе. Хотя удаление одного объекта может привести к возникновению большого количества недостижимых объектов, операции рекурсивного обновления счетчиков ссылок могут легко быть отложены и выполнены по частям в разные моменты времени. Таким образом, подсчет ссылок особенно привлекателен в тех случаях, когда возникают вопросы работы в реальном времени, а также в интерактивных программах, в которых нежелательны длинные неожиданные паузы. Еще одним преимуществом подсчета ссылок является немедленная сборка мусора, обеспечивающая экономное использование памяти.

7.5.4 Упражнения к разделу 7.5

Упражнение 7.5.1. Что произойдет со счетчиками ссылок объектов на рис. 7.19 при следующих действиях:

а) удаляется указатель от A на B;

б) удаляется указатель от X на A;

в) удаляется узел C.

Упражнение 7.5.2. Что произойдет со счетчиками ссылок при удалении указателя на D из A на рис. 7.20?

7.6 Введение в сборку на основе отслеживания

Вместо сборки мусора в момент его образования сборщик мусора на основе отслеживания периодически запускается для поиска недостижимых объектов

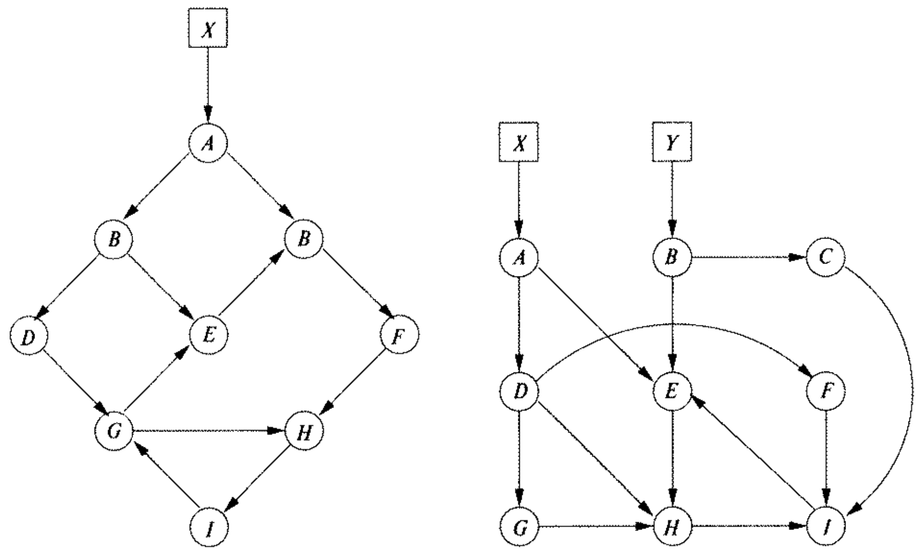


Рис. 7.19. Сеть объектов

Рис. 7.20. Еще одна сеть объектов

и утилизации используемой ими памяти. Обычно такой сборщик мусора запускается при исчерпании свободной памяти или когда ее количество становится меньше некоторого порогового значения.

Мы начнем с рассмотрения простейшего алгоритма сборки мусора "пометить и подмести". Затем будут рассмотрены различные алгоритмы на основе отслеживания, использующие четыре состояния, в которых могут находиться блоки памяти. В этом разделе вы также найдете массу усовершенствований базового алгоритма, включая те, в которых частью функции сборки мусора является перемещение объектов.

7.6.1 Базовый сборщик мусора

Алгоритмы сборки мусора *пометить и подмести* (mark-and-sweep) представляют собой простые алгоритмы, которые находят все недостижимые объекты и помещают их в список свободной памяти. Алгоритм 7.12 на первом этапе посещает и помечает все достижимые объекты, после чего "подметает" всю кучу, освобождая недостижимые объекты. Алгоритм 7.14, который мы рассмотрим после общей схемы алгоритмов на основе отслеживания, представляет собой оптимизацию алгоритма 7.12. Используя дополнительный список для хранения всех объектов, для которых выделена память, он посещает достижимые объекты только один раз.

Алгоритм 7.12. Сборка мусора "пометить и подмести"

ВХОД: корневое множество объектов, куча и *список свободной памяти* (free list) *Free*, в котором перечислены все свободные блоки кучи. Как и в разделе 7.4.4, все блоки памяти помечены при помощи дескрипторов границ, которые указывают их статус (занято/свободно) и размер.

ВЫХОД: модифицированный список *Free* после удаления всего мусора.

МЕТОД: алгоритм, приведенный на рис. 7.21, использует несколько простых структур данных. Список *Free* содержит объекты, память которых должна быть освобождена. Список *Unscanned* хранит объекты, которые определены как достижимые, но преемники которых пока что рассмотрены не были, т.е. мы пока что не сканировали эти объекты в поисках достижимых через них других объектов. Изначально список *Unscanned* пуст. Кроме того, каждый объект включает бит (*бит достижимости* – reached-bit), который указывает, достижим ли данный объект. Перед началом алгоритма у всех объектов, для которых выделена память, данный бит устанавливается равным 0.

```
        /* Фаза маркировки */
 1)     Добавить в список Unscanned все объекты, на которые имеется ссылка
                в корневом множестве, и установить бит достижимости каждого
                такого объекта равным 1;
 2)     while (Unscanned ≠ ∅) {
 3)             Удалить некоторый объект o из Unscanned;
 4)             for (Каждый объект o′, на который есть ссылка из o) {
 5)                     if (o′ недостижим, т.е. его бит достижимости равен 0) {
 6)                             Установить бит достижимости o′ равным 1;
 7)                             Внести o′ в список Unscanned;
                        }
                }
        }
        /* Фаза подметания */
 8)     Free = ∅;
 9)     for (Каждый блок памяти o в куче) {
10)             if (o недостижим (бит достижимости равен 0)) Добавить o
                        в список Free;
11)             else Установить бит достижимости o равным 0;
        }
```

Рис. 7.21. Сборщик мусора методом "пометить и подмести"

В строке 1 на рис. 7.21 выполняется инициализация списка *Unscanned* путем размещения в нем всех объектов, на которые имеются ссылки из корневого мно-

жества. Бит достижимости этих объектов устанавливается равным 1. Строки 2–7 представляют собой цикл, в котором поочередно проверяется каждый объект o, помещенный в список *Unscanned*.

Цикл for в строках 4–7 реализует сканирование объекта o. Мы просматриваем каждый объект o', на который в объекте o обнаруживается ссылка. Если o' уже был достигнут (его бит достижимости равен 1), то никаких действий с o' выполнять не требуется – либо этот объект уже был сканирован, либо он находится в списке *Unscanned* и будет сканирован позже. Однако если o' еще не был достигнут, то в строке 6 его бит достижимости устанавливается равным 1, а сам o' в строке 7 добавляется в список *Unscanned*. Этот процесс проиллюстрирован на рис. 7.22. Здесь показан список *Unscanned* из четырех объектов. Выполняется сканирование первого объекта в этом списке, соответствующего рассматривавшемуся выше объекту o. Пунктирные линии соответствуют трем видам объектов, которые могут быть достижимы из o.

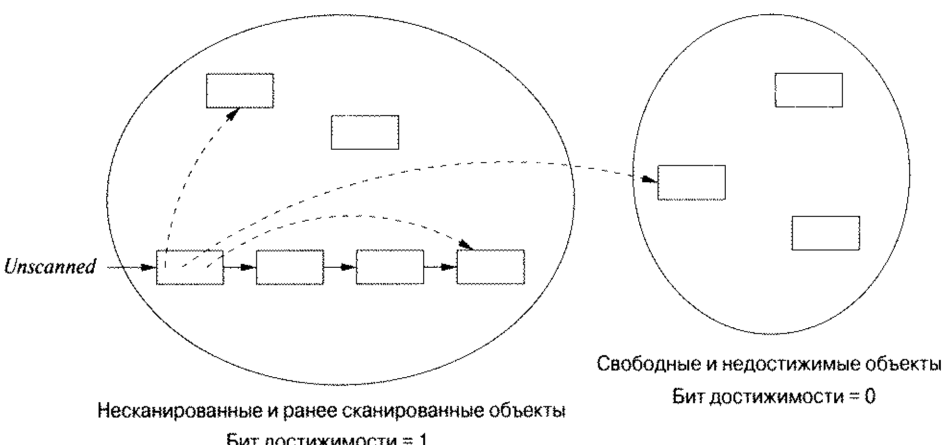


Рис. 7.22. Соотношения между объектами на фазе маркировки сборщика мусора "пометить и подмести"

1. Ранее сканированный объект, который не требуется сканировать повторно.

2. Объект, находящийся в настоящий момент в списке *Unscanned*.

3. Достижимый элемент, который ранее считался недостижимым.

Строки 8–11 представляют собой фазу подметания – утилизацию памяти всех объектов, которые остались недостижимыми по окончании фазы маркировки. Заметим, что сюда входят все объекты, которые были в списке *Free* изначально. Поскольку непосредственное перечисление недостижимых объектов невозможно, алгоритм "подметает" всю кучу. Строка 10 по одному помещает свободные

блоки памяти и недостижимые объекты в список *Free*, а строка 11 обрабатывает достижимые объекты. Их бит достижимости устанавливается равным 0, чтобы при следующем выполнении алгоритма сборки мусора выполнялись необходимые предусловия. □

7.6.2 Базовая абстракция

Все алгоритмы на основе отслеживания вычисляют множество достижимых объектов, а затем получают дополнение этого множества. Память используется следующим образом.

а) Программа (мутатор) во время работы выполняют запросы на выделение памяти.

б) Сборщик мусора выясняет достижимость объектов путем отслеживания.

в) Сборщик мусора утилизирует память недостижимых объектов.

Этот цикл проиллюстрирован на рис. 7.23 с использованием четырех состояний блоков памяти: *Свободен* (*Free*), *Недостижим* (*Unreached*), *Несканирован* (*Unscanned*) и *Сканирован* (*Scanned*). Состояние блока может храниться в самом блоке или в структуре данных, использующейся алгоритмом сборки мусора.

Хотя реализации алгоритмов на основе отслеживания могут отличаться, все они могут быть описаны в терминах указанных состояний.

1. *Свободен*. Блок памяти, находящийся в состоянии *Свободная*, готов для выделения. Таким образом, свободный блок не должен хранить достижимый объект.

2. *Недостижим*. Блоки считаются недостижимыми, если только их достижимость не доказана путем отслеживания. Пока его достижимость не установлена, блок памяти в процессе сборки мусора находится в состоянии *Недостижим*.

3. *Несканирован*. Блоки памяти, достижимость которых установлена, могут находиться в одном их двух состояний — *Несканирован* или *Сканирован*. Блок находится в состоянии *Несканирован*, если известно, что он достижим, но указатели в нем еще не просканированы. Переход в состояние *Несканирован* из состояния *Недостижим* осуществляется при определении достижимости блока (см. рис. 7.23, *б*).

4. *Сканирован*. Каждый несканированный объект в конечном счете должен быть просканирован и перейти в состояние *Сканирован*. Для *сканирования* объекта в нем отслеживаются все указатели и объекты, на которые они указывают. Если обнаружена ссылка на объект в состоянии *Недостижим*, он

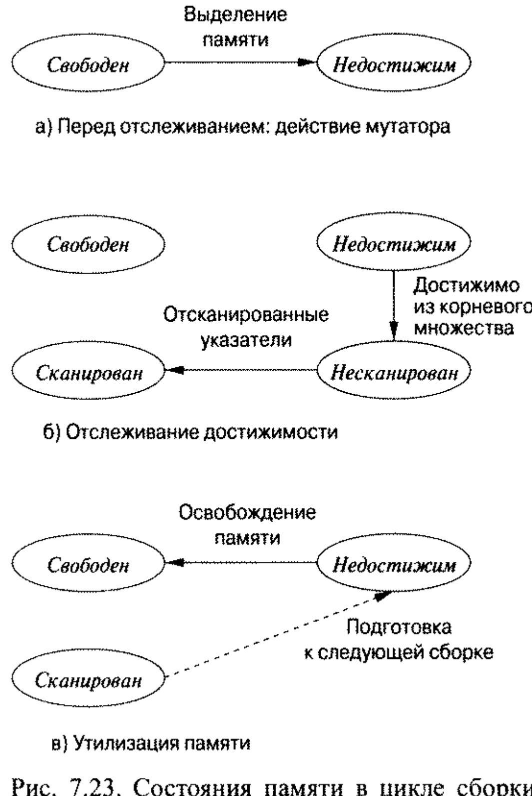


Рис. 7.23. Состояния памяти в цикле сборки мусора

переходит в состояние *Несканирован*. По завершении сканирования объект переходит в состояние *Сканирован* (см. нижнюю часть рис. 7.23, *б*). Сканированный объект может содержать ссылки только на другие объекты в состоянии *Сканирован* или *Несканирован* и никогда — на объекты в состоянии *Недостижим*.

Когда больше не остается объектов в состоянии *Несканирован*, вычисление достижимости завершается. Объекты, оставшиеся после этого в состоянии *Недостижим*, являются действительно недостижимыми объектами. Сборщик мусора утилизирует занимаемую ими память и помещает соответствующие блоки памяти в состояние *Свободен*, что показано сплошной линией на рис. 7.23, *в*. Для подготовки к следующей сборке мусора объекты из состояния *Сканирован* переходят в состояние *Недостижим* (пунктирная линия на рис. 7.23, *в*). Помните, что на самом деле сейчас все эти объекты достижимы; состояние *Недостижим* потребуется при следующем цикле сборки мусора, поскольку к тому моменту достижимые в настоящее время объекты могут превратиться в недостижимые.

Пример 7.13. Рассмотрим, как структуры данных алгоритма 7.12 связаны с четырьмя описанными состояниями. Членство в списках *Free* и *Unscanned* и бит достижимости позволяют однозначно идентифицировать все четыре состояния. На рис. 7.24 приведены всех четыре состояния в терминах структур данных алгоритма 7.12. □

Состояние	В списке *Free*	В списке *Unscanned*	Бит достижимости
Свободен	Да	Нет	0
Недостижим	Нет	Нет	0
Несканирован	Нет	Да	1
Сканирован	Нет	Нет	1

Рис. 7.24. Представление состояний в алгоритме 7.12

7.6.3 Оптимизация алгоритма "пометить и подмести"

Последний шаг базового алгоритма дорогостоящ в силу отсутствия простого способа поиска только недостижимых объектов без просмотра всей кучи. Усовершенствованный алгоритм Бейкера (Baker) поддерживает список всех объектов, для которых выделена память. Для того чтобы найти множество недостижимых объектов, память которых следует освободить, необходимо вычислить разность множества всех объектов и множества достижимых объектов.

Алгоритм 7.14. Сборщик мусора "пометить и подмести" Бейкера

Вход: корневое множество объектов, куча, список свободных блоков *Free* и список *Unreached* объектов, для которых выделена память.

Выход: модифицированные списки свободных блоков *Free* и выделенной памяти *Unreached*.

Метод: В этом алгоритме, показанном на рис. 7.25, структуры данных сборщика мусора представляют собой четыре списка, *Free*, *Unreached*, *Unscanned* и *Scanned*, каждый из которых хранит все объекты с состоянием, соответствующим имени списка. Эти списки могут быть реализованы как встроенные дважды связанные списки из раздела 7.4.4. Бит достижимости объекта не используется, но мы считаем, что у каждого объекта имеются биты, указывающие, в каком из четырех состояний находится этот объект. Изначально *Free* представляет собой список, поддерживаемый диспетчером памяти, а все объекты, для которых выделена память, находятся в списке *Unreached* (также поддерживаемом диспетчером памяти при выделении памяти для объектов).

В строках 1 и 2 инициализируются список *Scanned*, представляющий собой пустое множество, и список *Unscanned*, который содержит объекты, достижи-

```
1) Scanned = ∅;
2) Unscanned = множество объектов, на которые есть ссылки из корневого
   множества. Объекты, помещенные в множество Unscanned, удаляются
   из множества Unreached;
3) while (Unscanned ≠ ∅) {
4)       переместить объект o из Unscanned в Scanned;
5)       for (Каждый объект o', на который есть ссылки из o) {
6)             if (o' ∈ Unreached)
7)                   Переместить o' из Unreached в Unscanned;
         }
   }
8) Free = Free ∪ Unreached;
9) Unreached = Scanned;
```

Рис. 7.25. Алгоритм "пометить и подмести" Бейкера

мые из корневого множества. Заметим, что эти объекты ранее предположительно находились в списке *Unreached*, из которого они должны быть удалены. Строки 3–7 представляют собой непосредственную реализацию базового алгоритма маркировки с использованием упомянутых списков. Цикл for в строках 5–7 просматривает ссылки в несканированном объекте o, и, если какие-то из этих ссылок o' не были достижимы, в строке 7 состояние o' изменяется на несканированное.

В конце строка 8 берет множество объектов, все еще остающихся в списке *Unreached*, и освобождает их память, перенося их в список *Free*. Строка 9 берет все отсканированные объекты и инициализирует список *Unreached* этими объектами. При создании новых объектов диспетчер памяти будет изымать соответствующие блоки из списка *Free* и добавлять их в список *Unreached*. □

В обоих алгоритмах этого раздела считается, что блоки, которые возвращаются в список свободных, остаются такими же, какими были до освобождения. Однако, как говорилось в разделе 7.4.4, объединение смежных свободных блоков в бóльшие блоки имеет свои преимущества. Если мы хотим делать это, то всякий раз при возврате блока в список свободных блоков в строке 10 на рис. 7.21 или в строке 8 на рис. 7.25 следует проверить, не свободны ли блоки слева и справа от освобождаемого, и выполнить слияние, если имеется смежный свободный блок.

7.6.4 Сборщики мусора "пометить и сжать"

Перемещающие (relocating) сборщики переносят достижимые объекты в куче таким образом, чтобы устранить фрагментацию памяти. Обычно занятое достижимыми объектами пространство существенно меньше освобожденной памяти. Таким образом, после идентификации всех "дыр" между достижимыми объектами

можно не освобождать их индивидуально, а просто переместить все достижимые объекты в один конец кучи, оставив всю остальную память кучи в виде одного большого блока свободной памяти. В конце концов, сборщик все равно анализирует все ссылки в достижимых объектах, так что обновить их так, чтобы они указывали на новые местоположения объектов, — не такая уж большая работа. К ним следует только не забыть добавить ссылки из корневого множества, которые также должны быть обновлены.

Сборка всех достижимых объектов в смежных позициях снижает фрагментацию памяти, упрощая размещение крупных объектов. Кроме того, данные при этом занимают меньше строк кэша и страниц, так что перемещение повышает временну́ю и пространственную локальность приложения. Это связано с тем, что новые объекты создаются примерно в одно и то же время и в близко расположенных блоках памяти. Объекты в близких блоках памяти используют преимущества предвыборки при совместном использовании. При перемещении упрощается также структура для управления свободной памятью: все, что нам надо вместо списка свободных блоков, — указатель *free* на начало свободного блока.

Перемещающие сборщики различаются: одни выполняют перемещение без привлечения дополнительной памяти, "на месте", другие резервируют пространство для перемещения.

- *Сборщик "пометить и сжать"*, описанный в этом разделе, выполняет уплотнение "на месте". Такое уплотнение снижает количество необходимой для сжатия памяти.

- Более эффективный и популярный *копирующий сборщик* из раздела 7.6.5 перемещает объекты из одной области памяти в другую. Резервирование дополнительной памяти позволяет перемещать объекты по мере их обнаружения.

Сборщик "пометить и сжать" из алгоритма 7.15 состоит из трех фаз.

1. Первая фаза — маркировка, аналогичная такой же фазе из ранее рассматривавшихся алгоритмов "пометить и подмести".

2. Во второй фазе алгоритм сканирует выделенный раздел кучи и вычисляет новые адреса для каждого достижимого объекта. Новые адреса назначаются с нижнего конца кучи так, чтобы между достижимыми объектами не было неиспользованных промежутков. Новые адреса каждого объекта записываются в структуру *NewLocation*.

3. Наконец, алгоритм копирует объекты в их новые позиции, параллельно обновляя все ссылки в этих объектах так, чтобы они указывали на новые положения в памяти. Необходимые адреса можно найти в структуре *NewLocation*.

Алгоритм 7.15. Сборщик мусора "пометить и сжать"

Вход: корневое множество объектов, куча и указатель *free* на начало свободного пространства.

Выход: новое значение указателя *free*.

Метод: алгоритм на рис. 7.26, использующий следующие структуры данных.

1. Список *Unscanned*, как и в алгоритме 7.12.

2. Биты достижимости во всех объектах, как в алгоритме 7.12. Для того чтобы упростить описание алгоритма, будем называть объекты достижимыми или недостижимыми, имея в виду равенство их битов достижимости соответственно 1 или 0. Изначально все объекты недостижимы.

3. Указатель *free*, который отмечает начало нераспределенной памяти в куче.

4. Таблица *NewLocation*. Эта структура может представлять собой хеш-таблицу, дерево поиска или иную структуру, реализующую две операции:

   а) установку значения $NewLocation\,(o)$ равным новому адресу объекта o;

   б) получение для данного объекта o значения $NewLocation\,(o)$.

   Мы не будем останавливаться на конкретной структуре, хотя вы можете считать, что *NewLocation* — хеш-таблица, операции установки и получения значения в которой требуют в среднем постоянного времени, независимого от количества объектов в куче.

Первая фаза — маркировки — состоит из строк 1–7 и, по сути, представляет собой то же, что и первая фаза алгоритма 7.12. На второй фазе (строки 8–12) происходит обход всех блоков в выделенной части кучи начиная с нижнего конца, с меньших адресов. В результате блоки получают новые адреса, увеличивающиеся в том же порядке, что и старые адреса. Этот порядок важен, поскольку объекты можно перемещать только "влево", в место, ранее занятое объектами, которые к этому моменту уже перемещены.

Строка 8 присваивает указателю *free* адрес начала кучи. На этой фазе указатель *free* используется для указания первого доступного нового адреса. Новые адреса создаются только для тех объектов o, которые помечены как достижимые. В строке 10 такой объект получает новый адрес, а в строке 11 указатель *free* увеличивается на количество памяти, требующееся объекту o, так что после этого *free* вновь указывает на начало свободной памяти.

Последняя фаза (строки 13–17) вновь обходит достижимые объекты в том же порядке слева направо, что и вторая фаза. В строках 15 и 16 все внутренние

```
     /* Маркировка */
 1)  Unscanned = множество объектов, достижимых из корневого множества;
 2)  while (Unscanned ≠ ∅) {
 3)      Удалить объект o из Unscanned;
 4)      for (Каждый объект o', на который имеется ссылка в o) {
 5)          if (o' недостижим) {
 6)              Пометить o' как достижимый;
 7)              Поместить o' в список Unscanned;
             }
         }
     }
     /* Вычисление новых позиций */
 8)  free = Начальный адрес памяти кучи;
 9)  for (Каждый блок памяти o в куче, начиная с нижнего конца) {
10)      if (o достижим) {
11)          NewLocation (o) = free;
12)          free = free + sizeof(o);
         }
     }
     /* Изменение ссылок и перенос достижимых объектов */
13)  for (Каждый блок памяти o в куче, начиная с нижнего конца) {
14)      if (o достижим) {
15)          for (Каждая ссылка o.r в o)
16)              o.r = NewLocation (o.r);
17)          Копирование o в NewLocation (o);
         }
     }
18)  for (Каждая ссылка r в корневом множестве)
19)      r = NewLocation (r);
```

Рис. 7.26. Сборщик мусора "пометить и сжать"

указатели достижимого объекта заменяются корректными новыми значениями с использованием таблицы *NewLocation*. После этого в строке 17 выполняется перемещение объекта o с исправленными внутренними ссылками в новое положение. Наконец, в строках 18 и 19 выполняется исправление указателей в тех элементах корневого множества, которые не являются объектами кучи, т.е. память для них выделена либо статически, либо в стеке. На рис. 7.27 показаны как перемещение достижимых объектов (они не заштрихованы) вниз по куче, так и обновление их внутренних указателей, которые после этого указывают на новые положения перемещенных объектов. □

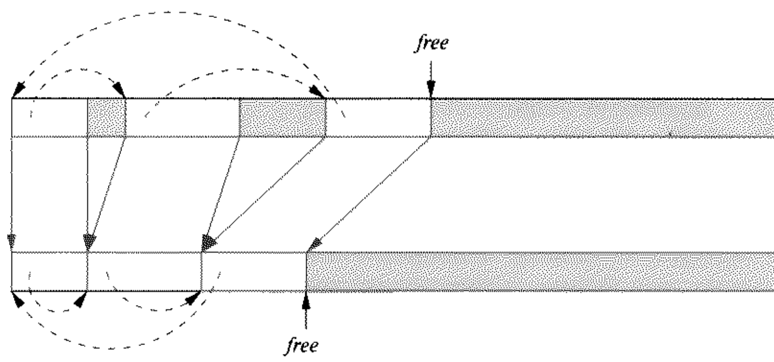


Рис. 7.27. Перемещение достижимых объектов к началу кучи с соответствующим обновлением внутренних указателей

7.6.5 Копирующие сборщики

Копирующие сборщики заранее резервируют память, в которую могут быть перемещены объекты, тем самым разрывая зависимость между отслеживанием и поиском свободной памяти.

Вся память разделена на два *полупространства* — A и B. Мутатор выделяет память в одном полупространстве, скажем, A, пока не заполнит его. В этот момент работа мутатора приостанавливается и сборщик мусора копирует все достижимые объекты в другое полупространство, в данном случае — B. По завершении работы сборщика полупространства меняются ролями. Мутатор продолжает выделять память для объектов в полупространстве B, а при очередном вызове сборщика он переместит достижимые объекты в полупространство A. Приведенный далее алгоритм разработан К.Д. Чени (C.J. Cheney).

Алгоритм 7.16. Копирующий сборщик Чени

ВХОД: корневое множество объектов и куча, состоящая из подпространства *From*, в котором находятся объекты, и свободного подпространства *To*.

ВЫХОД: по окончании алгоритма все объекты хранятся в полупространстве *To*. Указатель *free* указывает на начало оставшейся в полупространстве *To* свободной памяти. Полупространство *From* полностью свободно.

МЕТОД: алгоритм приведен на рис. 7.28. Алгоритм Чени находит достижимые объекты в полупространстве *From* и по мере выявления копирует их в полупространство *To*. Такое размещение группирует вместе связанные объекты и может повысить пространственную локальность.

Перед тем как приступить к детальному рассмотрению собственно алгоритма, который представляет собой функцию *CopyingCollector* на рис. 7.28, рассмотрим вспомогательную функцию *LookupNewLocation* в строках 11–16. Эта функция по-

```
1)  CopyingCollector () {
2)      for (Все объекты o из полупространства From)
            NewLocation (o) = NULL;
3)      unscanned = free = Начальный адрес полупространства To;
4)      for (Каждая ссылка r в корневом множестве)
5)          Заменяем r значением LookupNewLocation (r);
6)      while (unscanned ≠ free) {
7)          o = объект в позиции unscanned;
8)          for (Каждая ссылка o.r в o)
9)              o.r = LookupNewLocation (o.r);
10)         unscanned = unscanned + sizeof(o);
        }
    }
    /* Поиск новой позиции перемещенного объекта. */
    /* В противном случае помещение объекта в состояние Несканирован. */
11) LookupNewLocation (o) {
12)     if (NewLocation (o) = NULL) {
13)         NewLocation (o) = free;
14)         free = free + sizeof(o);
15)         Копирование o в позицию NewLocation (o);
        }
16)     return NewLocation (o);
    }
```

Рис. 7.28. Копирующий сборщик мусора

лучает объект *o* и находит его новое положение в полупространстве *To*, если объект еще не был перемещен. Все новые положения объектов записываются в структуре *NewLocation*, а значение NULL указывает, что *o* пока не имеет назначенной позиции в целевом полупространстве[8]. Как и в алгоритме 7.15, вид структуры *NewLocation* может варьироваться, но вполне можно считать, что это хеш-таблица.

Если в строке 12 выясняется, что у *o* пока нет новой позиции, то она назначается в начале свободной памяти в полупространстве *To* (строка 13). В строке 14 указатель *free* увеличивается на количество памяти, выделенной для объекта *o*, а в строке 15 объект *o* копируется из полупространства *From* в полупространство *To*. Таким образом, перемещение объекта из одного полупространства в другое оказывается побочным действием при первом поиске нового положения объекта.

[8]Если *o* не назначена никакая позиция, то в типичной структуре данных, такой как хеш-таблица, просто не будет упоминания об *o*.

Независимо от того, был ли уже перенесен объект, в строке 16 всегда возвращается его новое положение в полупространстве *To*.

Теперь можно приступить к рассмотрению самого алгоритма. Строка 2 устанавливает, что ни один из объектов полупространства *From* не имеет нового адреса. В строке 3 два указателя, *unscanned* и *free*, инициализируются начальным адресом полупространства *To*. Указатель *free* всегда указывает на начало свободной памяти в полупространстве *To*. При добавлении объектов в полупространство *To* те из них, адреса которых располагаются ниже *unscanned*, находятся в состоянии *Сканирован*, а объекты с адресами между указателями *unscanned* и *free* — в состоянии *Несканирован*. Таким образом, указатель *free* всегда опережает указатель *unscanned*, и, когда последний догоняет первый, объектов в состоянии *Несканирован* не остается и сборка мусора оказывается завершенной. Заметим, что мы работаем с полупространством *To*, хотя все ссылки в объектах, рассматриваемых в строке 8, ведут в полупространство *From*.

Строки 4 и 5 обрабатывают объекты, достижимые из корневого множества. Заметим, что в качестве побочного действия некоторые вызовы *LookupNewLocation* в строке 5 увеличивают указатель *free* — при выделении блоков памяти для объектов в полупространстве *To*. Таким образом, вход в цикл в строках 6–10 может не состояться только в одном случае — если корневое множество не содержит ссылки ни на один объект (в этом случае вся куча представляет собой мусор). Этот цикл сканирует все объекты в полупространстве *To*, находящиеся в состоянии *Несканирован*. Строка 7 получает очередной несканированный объект *o*. Затем в строках 8 и 9 каждая ссылка в *o* заменяется новым значением, указывающим положение объекта в полупространстве *To*. В качестве побочного действия, если ссылка в *o* указывает ранее недостижимый объект, вызов *LookupNewLocation* в строке 9 выделяет память для этого объекта в полупространстве *To* и переносит его туда. Наконец, строка 10 увеличивает указатель *unscanned* так, чтобы он указывал на следующий за *o* объект в полупространстве *To*. □

7.6.6 Сравнение стоимости

Алгоритм Чени обладает тем преимуществом, что он никак не затрагивает недостижимые объекты. С другой стороны, копирующий сборщик мусора вынужден перемещать содержимое всех достижимых объектов. Этот процесс оказывается особенно дорогостоящим в случае больших и долгоживущих объектов, время жизни которых превышает много циклов сборки мусора. Можно резюмировать информацию о времени работы каждого из четырех рассмотренных алгоритмов следующим образом, игнорируя стоимость обработки корневого множества.

- *Сборщик мусора "пометить и подмести"* (алгоритм 7.12). Время пропорционально количеству блоков в куче.

- *Сборщик мусора "пометить и подмести" Бейкера* (алгоритм 7.14). Время пропорционально количеству достижимых объектов.
- *Сборщик мусора "пометить и сжать"* (алгоритм 7.15). Время пропорционально количеству блоков в куче плюс общий размер достижимых объектов.
- *Копирующий сборщик Чени* (алгоритм 7.16). Время пропорционально общему размеру достижимых объектов.

7.6.7 Упражнения к разделу 7.6

Упражнение 7.6.1. Укажите шаги сборщика мусора "пометить и подмести" для

а) рис. 7.19 с удаленным указателем $A \to B$;

б) рис. 7.19 с удаленным указателем $A \to C$;

в) рис. 7.20 с удаленным указателем $A \to D$;

г) рис. 7.20 с удаленным объектом B.

Упражнение 7.6.2. Алгоритм "пометить и подмести" Бейкера перемещает объекты между четырьмя списками: *Free*, *Unreached*, *Unscanned* и *Scanned*. Для каждой сети объектов из упражнения 7.6.1 укажите для каждого объекта последовательность списков, в которых он находится с момента до начала сборки мусора и до момента после ее завершения.

Упражнение 7.6.3. Предположим, что мы выполнили сборку мусора "пометить и сжать" для каждой из сетей упражнения 7.6.1. Кроме того, будем считать, что

1. каждый объект имеет размер 100 байт;
2. изначально девять объектов в куче размещены в алфавитном порядке, начиная с нулевого байта кучи.

Каким будет адрес каждого объекта по завершении сборки мусора?

Упражнение 7.6.4. Предположим, что для каждой из сетей упражнения 7.6.1 выполняется алгоритм копирующей сборки мусора Чени. Предположим также, что

1. каждый объект имеет размер 100 байт;
2. список несканированных объектов представляет собой очередь, а когда объект имеет более одного указателя, достижимые объекты помещаются в очередь в алфавитном порядке;

3. полупространство *From* начинается с адреса 0, полупространство *To* — с адреса 10 000.

Каким будет значение *NewLocation* (o) для каждого объекта o, оставшегося после сборки мусора?

7.7 Распределенная сборка мусора

Простые сборщики мусора на основе отслеживания работают по принципу "остановись, мгновение" и могут вносить большие паузы в работу пользовательской программы. Сократить паузы можно, выполняя сборку мусора по частям. Можно разнести работу во времени, чередуя сборку мусора и работу программы, а можно разбить ее по пространственному принципу, выполняя при каждом запуске сборку мусора только в определенном подмножестве. Первый подход известен как *инкрементная сборка* (incremental collection), а второй — как *частичная сборка* (partial collection).

Инкрементный сборщик мусора разбивает анализ достижимости на небольшие модули, позволяя мутатору работать в промежутках между работой этих модулей. В процессе работы мутатора множество объектов изменяется, так что инкрементная сборка мусора усложняется. Из раздела 7.7.1 вы узнаете, что поиск несколько более консервативного ответа может повысить эффективность отслеживания.

Наилучший из известных алгоритмов частичной сборки — *сборка мусора по поколениям* (generational garbage collection). Он разбивает объекты в соответствии с тем, как давно они были созданы, и выполняет сборку мусора среди недавно созданных объектов более часто, поскольку, как правило, в среднем их время жизни меньше. Альтернативой является *алгоритм поезда* (train algorithm), также собирающий мусор в подмножествах объектов, но этот алгоритм дает лучшие результаты при работе с более старыми объектами. Два указанных алгоритма могут использоваться совместно для разработки частичного сборщика мусора, который по-разному обрабатывает более молодые и более старые объекты. Базовый алгоритм, лежащий в основе частичной сборки, будет рассмотрен в разделе 7.7.3, а затем более детально будет рассмотрена работа алгоритма сборки по поколениям и алгоритма поезда.

Идеи инкрементной и частичной сборки могут использоваться для создания алгоритма, который в многопроцессорной среде выполняет параллельную сборку мусора (см. раздел 7.8.1).

7.7.1 Инкрементная сборка мусора

Инкрементные сборщики мусора консервативны. Сборщик мусора не только не должен собирать объекты, не являющиеся мусором, но и не обязан собирать

весь мусор при каждом запуске. Будем говорить о мусоре, остающемся после сборки, как о *плавающем мусоре* (floating garbage). Конечно, желательно минимизировать плавающий мусор. В частности, инкрементный сборщик не должен оставлять никакой мусор, который был недостижим в начале цикла сборки. Если сборщик гарантирует такое поведение, то любой мусор, не собранный в некотором цикле, будет собран в следующем, так что при таком подходе нет никаких утечек памяти.

Другими словами, инкрементные сборщики обеспечивают безопасность определенной переоценкой множества достижимых объектов. Сначала они автоматически обрабатывают корневое множество программы, не взаимодействуя с мутатором. После определения начального множества несканированных объектов действия мутатора чередуются с отслеживанием. В это время любые действия мутатора, которые могут изменить достижимость, кратко записываются в дополнительной таблице, так что сборщик может внести необходимые изменения при возобновлении работы. Если память оказывается исчерпанной до завершения отслеживания, мутатор останавливается и сборщик завершает свою работу без перерывов. В любом варианте по завершении отслеживания утилизация памяти выполняется автоматически.

Точность инкрементной сборки

После того как объект становится недостижимым, его невозможно сделать достижимым вновь. Таким образом, в процессе работы сборщика и мутатора множество достижимых объектов может только

1. расти благодаря выделению памяти для новых объектов после начала сборки мусора;
2. уменьшаться из-за потери ссылок на объекты.

Пусть R — множество достижимых объектов в начале сборки мусора, New — множество объектов, для которых выделена память в процессе сборки мусора, и $Lost$ — множество объектов, которые стали недостижимыми из-за потери ссылок после начала отслеживания. Множество достижимых объектов по завершении отслеживания —

$$(R \cup New) - Lost$$

Определять достижимость объекта всякий раз, когда мутатор теряет ссылку на объект, — задача дорогостоящая, так что инкрементные сборщики не пытаются собрать весь мусор по окончании отслеживания. Любой остающийся мусор — плавающий мусор — должен быть подмножеством $Lost$. Выражаясь формально, множество найденных при отслеживании объектов S должно удовлетворять соотношению

$$(R \cup New) - Lost \subseteq S \subseteq (R \cup New)$$

Простое инкрементное отслеживание

Рассмотрим сначала простой алгоритм отслеживания, который находит верхнюю границу $R \cup New$. Поведение мутатора изменяется в процессе отслеживания следующим образом.

- Все ссылки, существовавшие до сборки мусора, сохраняются; т.е. прежде чем мутатор перепишет ссылку, ее старое значение запоминается и рассматривается как дополнительный несканированный объект, содержащий только одну эту ссылку.

- Все созданные объекты немедленно рассматриваются как достижимые, и им назначается состояние *Несканирован*.

Такая схема консервативна, но корректна, поскольку она находит множество R всех достижимых до начала сборки мусора объектов, плюс множество *New* всех вновь созданных объектов. Однако стоимость такого метода высока, поскольку алгоритм перехватывает все операции записи и запоминает все перезаписанные ссылки. Часть этой работы излишня, поскольку может включать объекты, недостижимые в конце сборки мусора. Можно избежать выполнения некоторой доли этой работы и повысить точность и эффективность алгоритма, если удастся определить, когда переписанные ссылки указывают на объекты, недостижимые в конце текущего цикла сборки мусора. В описанном ниже алгоритме используются указанные идеи.

7.7.2 Инкрементный анализ достижимости

Если работа мутатора чередуется с работой базового алгоритма отслеживания, такого как алгоритм 7.12, то некоторые достижимые объекты могут быть ошибочно классифицированы как недостижимые. Проблема в том, что действия мутатора способны нарушить ключевой инвариант алгоритма, а именно — то, что *Сканированный* объект может содержать ссылки только на другие *Сканированные* или *Несканированные* объекты, но никогда — на объекты *Недостижимые*. Рассмотрим следующий сценарий.

1. Сборщик мусора выясняет, что объект o_1 — достижимый, и сканирует указатели внутри o_1, помещая при этом объект o_1 в состояние *Сканирован*.

2. Мутатор сохраняет ссылку на объект o, находящийся в состоянии *Недостижим* (но при этом достижимый), в объекте o_1 в состоянии *Сканирован*. Это делается путем копирования ссылки на o из объекта o_2, который в настоящий момент находится в состоянии *Недостижим* или *Несканирован*.

3. Мутатор теряет ссылку на o в объекте o_2. Эта ссылка может быть переписана до того, как она будет сканирована, или o_2 может стать недостижимым

и не оказаться в состоянии *Несканирован*, так что указанная ссылка так и не будет просканирована.

Итак, объект o достижим через объект o_1, но сборщик мусора не может найти ссылку на o ни в объекте o_1, ни в объекте o_2.

Ключом к более точному инкрементному отслеживанию является запись всех копирований ссылок на недостижимые в настоящий момент объекты из объектов, которые еще не были сканированы, в уже просканированные. Для перехвата проблемных пересылок ссылок алгоритм может изменять действия мутатора в процессе отслеживания любым из приведенных ниже способов.

- *Барьеры записи.* Перехватываются записи ссылок на объект o в состоянии *Недостижим* в объект o_1 в состоянии *Сканирован*. В этом случае o классифицируется как достижимый объект и помещается в множество *Unscanned*. Другим способом является помещение объекта o_1 назад в список *Unscanned* для повторного сканирования.

- *Барьеры чтения.* Перехватываются чтения ссылок из объектов в состоянии *Недостижим* или *Несканирован*. Когда мутатор читает ссылку на объект o из объекта в состоянии *Недостижим* или *Несканирован*, объект o классифицируется как достижимый и помещается в множество *Unscanned*.

- *Барьеры пересылки.* Перехватываются потери исходных ссылок в объектах в состоянии *Недостижим* или *Несканирован*. Когда мутатор переписывает ссылку в объекте в состоянии *Недостижим* или *Несканирован*, переписываемая ссылка сохраняется, классифицируется как достижимая и помещается в множество *Unscanned*.

Ни один из указанных вариантов не позволяет найти наименьшее множество достижимых объектов. Если в процесс отслеживания выясняется, что объект является достижимым, то он остается достижимым, даже если все ссылки на него будут переписаны до завершения отслеживания. Таким образом, найденное множество достижимых объектов находится между $(R \cup New) - Lost$ и $(R \cup New)$.

Наиболее эффективным из предложенных выше вариантов является барьер записи. Барьер чтения более дорогостоящий, поскольку обычно в программе выполняется больше чтений, чем записей. Барьер пересылки неконкурентоспособен; поскольку многие объекты "умирают молодыми", этот подход оставляет много недостижимых объектов.

Реализация барьера записи

Реализовать барьер записи можно двумя способами. Первый способ состоит в запоминании в фазе мутатора всех новых ссылок, записанных в объекты

в состоянии *Сканирован*. Можно поместить все эти ссылки в список; размер такого списка пропорционален количеству операций записи в объекты в состоянии *Сканирован*, если только не удалять из списка дубликаты. Заметим, что ссылки в списке могут быть переписаны и потенциально проигнорированы.

Второй, более эффективный, подход состоит в запоминании позиций, в которых выполнялась запись. Эти места могут быть запомнены в виде списка позиций с возможным устранением дубликатов. Заметим, что точность в указании позиций записи не важна, поскольку все они будут сканированы заново. Таким образом, имеется несколько методов, которые позволяют обойтись запоминанием меньшего количества деталей.

- Вместо запоминания точного адреса записываемых объекта и поля можно запомнить только объект, в котором содержится это поле.

- Можно разделить память на блоки фиксированного размера, известные как *карты* (cards), и использовать битовый массив для запоминания карт, в которые выполнялась запись.

- Можно запоминать страницы, которые содержат записываемые ячейки памяти. Можно защитить только страницы, содержащие объекты в состоянии *Сканирован*. Тогда все записи в сканированные объекты будут обнаружены без явного выполнения каких-либо команд, поскольку они будут приводить к нарушению защиты, и операционная система будет генерировать соответствующие программные исключения.

В общем случае огрубление уровня детализации, на котором выполняется запоминание записанных позиций, приводит к меньшему количеству требуемой памяти, но повышенным затратам на повторное сканирование. В первой схеме должны быть сканированы все ссылки в измененных объектах, независимо от того, какая именно ссылка была модифицирована. В двух последних схемах в конце процесса отслеживания требуется повторно сканировать все достижимые объекты в измененных картах или страницах.

Объединение инкрементных и копирующих методов

Описанных выше методов достаточно для сборки мусора "пометить и подмести". Копирующая сборка несколько более сложная из-за ее взаимодействия с мутатором. Объекты в состояниях *Сканирован* и *Несканирован* имеют по два адреса: один — в полупространстве *From*, другой — в полупространстве *To*. Как в алгоритме 7.16, мы должны поддерживать отображение старого адреса объекта на его новый адрес после перемещения.

Имеется два варианта обновления ссылок. Во-первых, можно заставить мутатор выполнять все изменения в полупространстве *From* и только в конце сборки

мусора обновить все указатели и скопировать все содержимое в полупространство *To*. Во-вторых, можно вносить все изменения в представление в полупространстве *To*. Когда мутатор разыменовывает указатель в полупространство *From*, этот указатель транслируется в новое положение в полупространстве *To*, если таковое существует. В конце все указатели должны быть транслированы таким образом, чтобы указывать в полупространство *To*.

7.7.3 Основы частичной сборки

Фундаментальным является тот факт, что, как правило, объекты "умирают молодыми". Было обнаружено, что обычно от 80 до 98% вновь созданных объектов уничтожаются в пределах нескольких миллионов команд или до того, как будет выделен следующий мегабайт памяти. Таким образом, очень часто объекты становятся недостижимыми еще до начала сборки мусора. Таким образом, достаточно выгодно часто выполнять сборку мусора новых объектов.

Объекты, которые переживут одну сборку мусора, скорее всего, переживут и многие другие. При работе описанных до сих пор сборщиков мусора одни и те же "старые" объекты будут определяться как достижимые вновь и вновь и в случае копирующих сборщиков будут снова и снова, при каждом запуске сборщика, перемещаться в памяти с места на место. Сборка мусора по поколениям чаще всего работает в области кучи, содержащей самые молодые объекты, так что при этом, как правило, собирается много мусора ценой относительно небольшой работы. С другой стороны, алгоритм поезда не тратит большую часть времени на молодые объекты, но уменьшает паузы в работе программы, возникающие из-за сборки мусора. Таким образом, одна из неплохих стратегий состоит в объединении указанных алгоритмов, когда сборка мусора по поколениям применяется для молодых объектов; когда объект достаточно "стареет", он переносится в отдельную кучу под управлением алгоритма поезда.

Будем говорить о множестве объектов, которые обрабатываются в одном цикле частичной сборки мусора, как о *целевом* (target) множестве, а об остальных объектах — как о *стабильном* (stable) множестве. В идеальном случае частичный сборщик мусора должен утилизировать все объекты целевого множества, которые недостижимы из корневого множества. Однако это требует прохода по всем объектам, т.е. именно того, чего мы стремимся избежать. Поэтому частичные сборщики консервативно утилизируют только те объекты, которые не могут быть достигнуты ни из корневого, ни из стабильного множества. Поскольку некоторые объекты в стабильном множестве сами могут оказаться недостижимыми, может оказаться так, что достижимыми будут считаться некоторые объекты целевого множества, к которым на самом деле не существует путей из корневого множества.

Можно адаптировать сборщики мусора, описанные в разделах 7.6.1 и 7.6.4, для работы в качестве частичных путем изменения определения корневого мно-

жества. Вместо того, чтобы рассматривать только объекты в регистрах, стеке и глобальных переменных, корневое множество теперь должно включать все объекты стабильного множества, которые указывают на объекты целевого множества. Как и раньше, для поиска всех достижимых объектов отслеживаются ссылки из целевых объектов на другие целевые объекты. Можно игнорировать все указатели на стабильные объекты, поскольку в данном запуске частичной сборки все эти объекты рассматриваются как достижимые.

Чтобы определить стабильные объекты, имеющие ссылки на целевые объекты, можно применить методы, подобные использующимся в инкрементальной сборке мусора. В инкрементальной сборке требуется помнить все записи ссылок из сканированных объектов на недостижимые в процессе отслеживания. Здесь же необходимо запоминать все записи ссылок из стабильных объектов на целевые во время работы мутатора. Когда мутатор сохраняет в стабильном объекте ссылку на объект из целевого множества, надо запомнить либо ссылку, либо место записи. Будем говорить о множестве объектов, хранящих ссылки из стабильных объектов на целевые, как о *запомненном множестве* (remembered set) для данного набора целевых объектов. Как говорилось в разделе 7.7.2, можно уменьшить представление запомненного множества, если записывать только карту или страницу, содержащую объект, в который выполняется запись.

Частичные сборщики мусора часто реализуются как копирующие. Некопирующие сборщики могут быть также реализованы с использованием связанных списков для отслеживания достижимых объектов. Описанная ниже схема "по поколениям" представляет собой пример комбинации копирования с частичной сборкой.

7.7.4 Сборка мусора по поколениям

Сборка мусора по поколениям представляет собой эффективный способ использования того свойства, что большинство объектов "умирают молодыми". Куча в этом случае разбивается на ряд разделов. Будем использовать соглашение об их нумерации как $0, 1, 2, \ldots, n$, где разделы с малыми номерами хранят более молодые объекты. Первоначально объекты создаются в разделе 0. По заполнении раздела 0 в нем выполняется сборка мусора, а все достижимые объекты переносятся в раздел 1. Теперь, когда раздел 0 вновь пуст, продолжим создание новых объектов в этом разделе. Когда же раздел 0 заполнится опять[9], в нем выполняется сборка мусора, и достижимые объекты из него копируются в раздел 1, где

[9]Технически раздел не "заполняется", поскольку при необходимости он может быть расширен диспетчером памяти за счет дополнительных блоков на диске. Однако обычно существует предельный размер раздела, отличный от указанного. Мы будем говорить о достижении этого предела как о заполнении раздела.

присоединяются к ранее скопированным объектам. Эти действия повторяются до заполнения раздела 1; в этот момент выполняется сборка мусора в разделах 0 и 1.

В общем случае в каждом цикле сборка мусора применяется ко всем разделам с номерами i и менее для некоторого i. Значение i – это максимальный номер заполнившегося к этому моменту раздела. Каждый раз, когда объект переживает сборку мусора (т.е. выясняется его достижимость), он переносится из своего раздела в раздел со следующим по величине номером, пока не достигнет раздела с наибольшим номером n.

Пользуясь терминологией из раздела 7.7.3, можно сказать, что, когда выполняется сборка мусора в разделах с номерами i и меньших, разделы от 0 до i составляют целевое множество, а все разделы с номерами больше i – стабильное множество. Для обеспечения поиска корневых множеств для всех возможных частичных сборок мусора для каждого раздела i поддерживается *запомненное множество* (remembered set), состоящее из всех объектов в разделах с номерами больше i, которые указывают на объекты в множестве i. Корневое множество частичной сборки мусора, выполняемой над разделом i, включает запомненные множества для раздела i и разделов с номерами, меньшими i.

В такой схеме при выполнении сборки мусора в разделе i выполняется также сборка во всех разделах с номерами, меньшими i. Такая стратегия основана на двух соображениях.

1. Поскольку более молодые поколения содержат большее количество мусора и сборка у них в любом случае выполняется более часто, ее имеет смысл выполнить и при сборке мусора в более старом поколении.

2. Следуя описанной стратегии, нам надо запоминать только ссылки, указывающие из объектов более старых поколений на объекты более молодые. Иными словами, ни запись объекта более молодого поколения, ни перенос объекта в следующее поколение не приводят к обновлению никакого из запомненных множеств. Если бы выполнялась сборка мусора в более старшем поколении без сборки в младших поколениях, то более молодое поколение стало бы частью стабильного множества, и мы должны были бы запоминать ссылки, которые из более молодого поколения указывают на более старое.

Итак, данная схема собирает мусор среди более молодых объектов более часто, причем такая сборка оказывается более эффективной по сравнению с полной, поскольку объекты "умирают молодыми". Сборка мусора в старших поколениях отнимает больше времени, поскольку она включает сборку во всех младших поколениях, а кроме того, старшие поколения содержат меньшее количество мусора. Тем не менее иногда требуется сборка мусора и в старших поколениях для удаления недостижимых объектов. Самое старшее поколение содержит самые старые

объекты, а сборка мусора в нем эквивалентна полной сборке мусора в куче. Иначе говоря, такая схема иногда приводит к необходимости полного отслеживания всех объектов в куче и, соответственно, к большим паузам в выполнении программы. Альтернативный способ обработки только старых объектов описывается в следующем разделе.

7.7.5 Алгоритм поезда

Сборка мусора по поколениям очень эффективна для молодых объектов, но существенно менее эффективна в применении к старым объектам, поскольку старые объекты должны перемещаться всякий раз при охватывающей их сборке; кроме того, они редко бывают мусором. Для повышения эффективности обработки старых объектов был разработан другой алгоритм инкрементной сборки мусора, получивший название *алгоритм поезда* (train algorithm). Он может применяться для сборки всего мусора, однако, пожалуй, лучше всего для работы с молодыми объектами использовать сборку мусора по поколениям, а объекты, пережившие несколько циклов таких сборок, переносить в другую кучу, которая управляется алгоритмом поезда. Еще одним преимуществом алгоритма поезда является то, что нам никогда не потребуется выполнять полную сборку мусора, что все же приходится время от времени делать при сборке мусора по поколениям.

Чтобы объяснить, откуда взялся алгоритм поезда, давайте рассмотрим простой пример, почему время от времени необходима полная сборка мусора при применении сборки по поколениям. На рис. 7.29 показаны два объекта с взаимными ссылками друг на друга, находящиеся в двух разделах, i и j, где $j > i$. Поскольку на оба объекта имеются ссылки извне раздела, сборка мусора только лишь в разделе i или только в разделе j никогда не подберет ни один из указанных объектов, которые на самом деле могут оказаться частью циклической структуры, не имеющей ссылок на нее извне и являющейся, таким образом, мусором. В общем случае показанные связи между объектами могут быть сложнее и включать большее количество объектов и длинные цепочки ссылок.

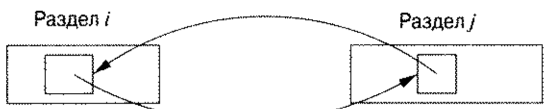


Рис. 7.29. Циклическая структура, охватывающая разные поколения, может оказаться циклическим мусором

В сборке мусора по поколениям рано или поздно происходит сборка в разделе j, а поскольку $i < j$, в этот же момент выполняется сборка и в разделе i. Таким образом, рассматриваемая структура оказывается полностью содержащейся в ча-

сти кучи, в которой выполняется сборка мусора, и при этом точно выясняется, является ли она мусором. Однако, если сборка мусора одновременно в блоках i и j не выполняется, мы получаем те же проблемы с циклическими ссылками, что и при использовании счетчиков ссылок.

Алгоритм поезда использует разделы фиксированного размера, называющиеся *вагонами* (car); вагон может представлять собой отдельный дисковый блок при условии, что не существует объектов, размер которых больше размера дискового блока; вагон может быть и большего размера, но этот размер фиксирован раз и навсегда. Вагоны собраны в *поезда* (train). Ограничений на количество вагонов в поезде нет, как не ограничено и количество поездов. Вагоны могут быть лексикографически упорядочены: сначала — по номеру поезда, а в пределах поезда — по номеру вагона, как показано на рис. 7.30.

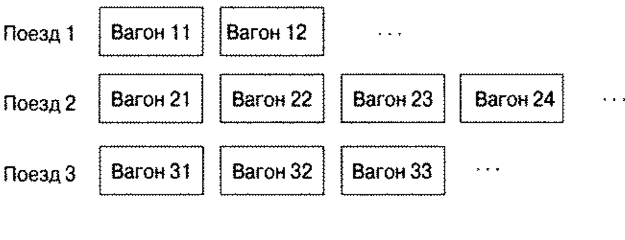


Рис. 7.30. Организация кучи для алгоритма поезда

Существует два способа сборки мусора алгоритмом поезда.

- Выполняется сборка мусора в первом вагоне в лексикографическом порядке (первый оставшийся вагон первого оставшегося поезда) за один шаг инкрементной сборки. Этот шаг аналогичен сборке мусора в первом разделе алгоритма сборки мусора по поколениям, поскольку мы поддерживаем "запомненный" список всех указателей извне вагона. На этом шаге мы идентифицируем как объекты без внешних ссылок, так и циклические структуры, являющиеся мусором и полностью размещающиеся в пределах одного вагона. Достижимые объекты вагона всегда перемещаются в некоторый другой вагон, так что каждый вагон, в котором проведена сборка мусора, становится пустым и может быть "отцеплен" от поезда.

- Иногда первый поезд не содержит внешних ссылок на него, т.е. не существует указателей из корневого множества ни на один вагон поезда, а запомненные множества вагонов содержат только указатели от других вагонов поезда, но не от других поездов. В таком случае поезд представляет собой большую коллекцию циклического мусора и можно удалить весь поезд целиком.

Запомненные множества

Теперь рассмотрим детали алгоритма поезда. Каждый вагон имеет запомненное множество, состоящее из всех ссылок на объекты вагона из

а) объектов вагонов того же поезда с бóльшими номерами;

б) объектов поездов с бóльшими номерами.

Кроме того, каждый поезд имеет запомненное множество, состоящее из всех ссылок из поездов с бóльшими номерами, т.е. запомненное множество поезда представляет собой объединение запомненных множеств его вагонов, за исключением ссылок, являющихся внутренними для данного поезда. Оба типа запомненных множеств могут быть разделены на внутреннюю (ссылки из того же поезда) и внешнюю (ссылки из иных поездов) части.

Заметим, что ссылки на объекты могут быть откуда угодно, а не только из объектов, следующих за данным в лексикографическом порядке. Однако указанные два процесса сборки мусора работают соответственно с первым вагоном первого поезда и с первым поездом в целом. Таким образом, когда в процессе сборки мусора требуется использовать запомненные множества, все ссылки могут быть только из более поздних в лексикографическом порядке вагонов и поездов — просто потому, что работа идет с первым в этом порядке вагоном (поездом). А значит, нет необходимости в запоминании ссылок из более ранних в лексикографическом порядке вагонов (поездов) в более поздние. Конечно, следует аккуратно работать с запомненными множествами, внося в них изменения всякий раз, когда мутатор переписывает ссылки в любом объекте.

Управление поездами

Наша цель — выявить в первом поезде все объекты, не являющиеся циклическим мусором. Затем либо в первом поезде не остается ничего, кроме циклического мусора, который будет утилизирован в следующем цикле сборки мусора, либо, если мусор не является циклическим, вагоны первого поезда могут быть обработаны по одному.

Следовательно, время от времени нам надо начинать новые поезда, несмотря на то, что ограничений на количество вагонов в поезде нет (так что, когда требуется дополнительная память, в принципе, можно просто добавить новый вагон к единственному поезду). Например, мы можем начинать новый поезд после каждых k созданий объектов, где k — некоторое заранее выбранное число. Иначе говоря, в общем случае новый объект размещается в последнем вагоне последнего поезда, если в нем имеется достаточно памяти, или, если памяти не хватает, в новом вагоне, который добавляется в конец последнего поезда. Однако периодически следует вместо этого начинать новый поезд с одним вагоном, в котором и размещать создаваемый объект.

Сборка мусора в вагоне

Ключевым моментом в алгоритме поезда является обработка первого вагона первого поезда в процессе выполнения цикла сборки мусора. Изначально в множество достижимых объектов входят те объекты вагона, на которые имеются ссылки из корневого множества и из запомненного множества этого вагона. Затем мы сканируем объекты вагона, как и в случае сборки мусора "пометить и подмести", но не сканируем ни один достижимый объект вне обрабатываемого вагона. После такого отслеживания некоторые объекты в вагоне могут быть идентифицированы как мусор. Утилизировать выделенную им память незачем, поскольку все равно весь вагон будет удален.

Однако вполне вероятно наличие в вагоне и достижимых объектов, которые должны быть перенесены в другое место. Вот как выглядят правила перемещения объектов.

- Если в запомненном множестве имеется ссылка из некоторого другого вагона (номер которого больше, чем номер вагона, в котором выполняется сборка), то объект перемещается в один из вагонов, из которых на него есть ссылка. Если имеется достаточное количество свободной памяти, объект может быть размещен в существующем вагоне поезда — источника ссылки; в противном случае объект может быть помещен в новый вагон.

- Если ссылок из других вагонов нет, но есть ссылка из корневого множества или из первого вагона, то объект перемещается в любой другой вагон данного поезда (или в новый вагон, если памяти в имеющихся вагонах недостаточно). По возможности следует выбирать тот вагон, из которого имеется ссылка на перемещаемый объект, — это поможет локализации циклической структуры в пределах одного вагона.

После перемещения всех достижимых объектов из первого вагона этот вагон удаляется.

Режим паники

С приведенными выше правилами связана одна проблема. Чтобы быть уверенным, что в конечном итоге будет собран весь мусор, необходима гарантия, что каждый поезд рано или поздно станет первым, и если этот поезд не представляет собой просто циклический мусор, то в конце концов все вагоны поезда будут удалены и поезд исчезнет по одному вагону за раз. Однако в соответствии со вторым правилом сборка мусора в первом вагоне первого поезда может привести к появлению нового последнего вагона. Очевидно, что при сборке не могут быть созданы два и более новых вагонов, поскольку все, что размещалось в первом вагоне, вполне поместится в новом, последнем вагоне. Но не может ли возникнуть ситуация, когда на каждом шаге сборки мусора будет создаваться новый вагон,

и в результате первый вагон никогда не будет полностью обработан, и сборка мусора так и не сможет перейти ко второму вагону?

К сожалению, такая ситуация возможна. Проблема возникает при наличии большой циклической структуры, не являющейся мусором, причем мутатор изменяет ссылки таким образом, что во время сборки мусора в вагоне в запомненном множестве никогда не встречаются ссылки из поездов с бóльшими номерами. Если в процессе каждого цикла сборки мусора в вагоне из поезда будет удаляться хотя бы один объект, то все в порядке, поскольку в конечном итоге первый поезд останется совсем без объектов. Но если окажется, что мусора, который можно собрать, нет, мы рискуем заниматься бесконечной чисткой первого вагона.

Чтобы избежать такой ситуации, следует изменить поведение в случае *безрезультатной* (futile) сборки мусора, т.е. вагона, из которого ни один объект не удаляется как мусор и не переносится в другой поезд. В этом случае режима паники мы вносим два изменения.

1. Когда переписывается ссылка на объект в первом вагоне, она становится новым членом корневого множества.

2. Если при сборке мусора объект в первом вагоне имеет ссылку из корневого множества, включая фиктивные ссылки, созданные первым правилом, то объект перемещается в другой поезд, даже если ссылок на него из других поездов нет. Не важно, в какой именно поезд будет перемещен объект, лишь бы это не был первый поезд.

Таким образом, при наличии ссылок на объекты в первом поезде извне первого поезда как минимум некоторый объект будет из этого поезда удален. После этого осуществляется выход из режима паники и возврат к обычному алгоритму.

7.7.6 Упражнения к разделу 7.7

Упражнение 7.7.1. Предположим, что сеть объектов на рис. 7.20 управляется инкрементным алгоритмом, который использует списки *Unreached*, *Unscanned*, *Scanned* и *Free*, как в алгоритме Бейкера. Пусть для конкретности список *Unscanned* представляет собой очередь, а при помещении в него в результате сканирования некоторого объекта сразу нескольких объектов, последние вносятся в список в алфавитном порядке. Предположим также, что, чтобы гарантировать, что ни один достижимый объект не станет мусором, используется барьер записи. Изначально список *Unscanned* состоит из A и B. Пусть далее происходят следующие события.

1. Сканируется A.

2. Указатель $A \to D$ переписывается и становится указателем $A \to H$.

3. Сканируется B.

4. Сканируется D.

5. Указатель $B \to C$ переписывается и становится указателем $B \to I$.

Смоделируйте инкрементную сборку мусора, полагая, что больше никакие указатели не переписываются. Какие объекты являются мусором? Какие объекты помещаются в список *Free*?

Упражнение 7.7.2. Повторите упражнение 7.7.1 в предположении, что

а) события 2 и 5 поменялись местами;

б) события 2 и 5 происходят до событий 1, 3 и 4.

Упражнение 7.7.3. Предположим, что куча состоит из девяти вагонов в трех поездах, как показано на рис. 7.30 (троеточия игнорируются). На объект o в вагоне 11 имеются ссылки из вагонов 12, 23 и 32. Где может оказаться объект o после сборки мусора в вагоне 11?

Упражнение 7.7.4. Повторите упражнение 7.7.3 для случаев, когда ссылки на объект o имеются

а) только из вагонов 22 и 31;

б) только из вагона 11.

Упражнение 7.7.5. Предположим, что куча состоит из девяти вагонов в трех поездах, как показано на рис. 7.30 (троеточия игнорируются). В настоящий момент мы находимся в режиме паники. На объект o_1 в вагоне 11 имеется единственная ссылка из объекта o_2 в вагоне 12. Эта ссылка переписывается. Что произойдет с объектом o_1 при сборке мусора в вагоне 11?

7.8 Дополнительные вопросы сборки мусора

Завершим наше исследование сборки мусора кратким рассмотрением четырех дополнительных тем.

1. Сборка мусора в параллельных средах.

2. Частичные перемещения объектов.

3. Сборка мусора в языках программирования, небезопасных с точки зрения типов.

4. Взаимодействие между автоматической и программно-управляемой сборкой мусора.

7.8.1 Параллельная сборка мусора

Сборка мусора становится особенно сложной при использовании в многопоточных приложениях на мультипроцессорных машинах. Вполне реальна ситуация, когда у серверного приложения параллельно работают тысячи потоков, каждый из которых является мутатором. Куча может состоять из гигабайтов памяти.

Масштабируемые алгоритмы сборки мусора должны использовать преимущества многопроцессорности. Будем называть сборщик мусора *параллельным*, если он использует несколько потоков, а работающий одновременно с мутатором сборщик будем называть *конкурентным*.

Мы опишем параллельный и частично конкурентный сборщик мусора, который использует конкурентную и параллельную фазы для выполнения основной части работы по отслеживанию, а затем переходит на стадию исключительного выполнения, чтобы гарантировать, что найдены все достижимые объекты и утилизирована вся память. В этом алгоритме нет никаких новых концепций; он просто демонстрирует, как можно скомбинировать рассмотренные ранее идеи для получения полного решения задачи параллельной и конкурентной сборки мусора. Однако перед нами возникают несколько новых вопросов реализации, связанных с параллельностью. Мы рассмотрим, как алгоритм координирует множественные параллельные потоки с применением весьма распространенной модели очереди.

Для понимания алгоритма следует помнить о масштабе задачи. Корневое множество многопоточного приложения существенно больше, чем однопоточного, так как, кроме регистров и глобальных переменных, в него входят стеки всех потоков. Размер кучи также может быть очень большим, как и количество достижимых данных. Существенно возрастает и скорость изменения данных.

Чтобы сократить продолжительность паузы, можно адаптировать уже рассматривавшиеся базовые идеи инкрементного анализа для многопоточного приложения, в котором сборщик мусора и мутатор работают одновременно. Вспомним, что инкрементный анализ состоит из следующих трех шагов (см. раздел 7.7).

1. Поиск корневого множества. Этот шаг обычно выполняется автоматически с остановленным мутатором (или мутаторами).

2. Чередование отслеживания достижимых объектов с выполнением мутатора (или мутаторов). На этом шаге всякий раз, когда мутатор переписывает ссылку из объекта в состоянии *Сканирован* на объект в состоянии *Недостижим*, эта ссылка запоминается. Как указано в разделе 7.7.2, есть разные варианты детализации запоминания таких ссылок. В этом разделе предполагается использование запоминания на основе карт, когда куча разделяется на части ("карты"), для которых в сопутствующем битовом массиве отмечается наличие в карте одной или нескольких переписанных ссылок.

3. Мутатор (или мутаторы) вновь блокируется для повторного сканирования всех карт, которые могут хранить ссылки на недостижимые объекты.

В случае больших многопоточных приложений множество объектов, достижимых из корневого множества, может быть очень большим. Затрачивать на посещение всех этих объектов время, когда все мутаторы приостановлены, – плохое решение. Кроме того, большой размер кучи и большое количество потоков мутатора приводят к тому, что после того, как все объекты просканированы, требуется повторное сканирование большого количества карт. Поэтому было бы разумно выполнять сканирование некоторых из этих карт параллельно, позволяя при этом мутаторам продолжать свою работу.

Для реализации параллельного выполнения шага 2 из описанных выше шагов будем одновременно с потоками мутатора использовать несколько потоков сборки мусора для отслеживания *большинства* достижимых объектов. Затем, при реализации шага 3, все потоки мутаторов блокируются, а потоки сборщиков работают параллельно, обеспечивая поиск всех достижимых объектов.

Отслеживание на шаге 2 выполняется за счет того, что каждый поток мутатора выполняет наряду со своей основной работой часть работы по сборке мусора. Кроме того, используются потоки, вся деятельность которых сосредоточена исключительно на сборке мусора. После начала цикла сборки мусора любое выделение памяти потоком мутатора сопровождается определенными действиями по отслеживанию. Потоки, занимающиеся исключительно сборкой мусора, активизируются только тогда, когда в работе основной программы возникает пауза. Как и в инкрементном анализе, когда мутатор записывает ссылку, которая указывает из объекта в состоянии *Сканирован* на объект в состоянии *Недостижим*, карта, хранящая эту ссылку, помечается как требующая повторного сканирования.

Вот как выглядит набросок параллельного конкурентного алгоритма сборки мусора.

1. Сканирование корневого множества для каждого потока мутатора и помещение всех объектов, достижимых непосредственно из корневого множества, в состояние *Несканирован*. Простейший инкрементный подход состоит в том, чтобы подождать, пока поток мутатора не вызовет диспетчер памяти, и сканировать его корневое множество, если это еще не было сделано. Если некоторый поток мутатора не вызывает функцию выделения памяти, а вся остальная работа по отслеживанию уже выполнена, для сканирования корневого множества этого потока его выполнение следует прервать.

2. Сканирование объектов, находящихся в состоянии *Несканирован*. Для поддержки параллельных вычислений используется очередь *рабочих пакетов* (work packets) фиксированной длины, каждый из которых хранит некоторое количество таких объектов. Несканированные объекты помещаются в ра-

бочие пакеты в момент обнаружения первых. Потоки извлекают пакеты из очереди и сканируют объекты, находящиеся в них. Такая стратегия позволяет равномерно распределить работу среди участников процесса отслеживания. Если в системе не хватает памяти для создания рабочих пакетов, можно просто маркировать карты с объектами, чтобы обеспечить их сканирование. Это всегда возможно, поскольку память под битовый массив для маркировки карт уже выделена.

3. Сканирование объектов в отмеченных картах. Когда в рабочей очереди не остается несканированных объектов и сканированы все корневые множества потоков, выполняется повторное сканирование карт в поиске достижимых объектов. Заметим, что, пока мутаторы работают, будут образовываться новые карты, которые надо сканировать заново. Таким образом, необходимо остановить процесс отслеживания при помощи некоторого критерия, такого как разрешение повторного сканирования карты только один раз (или некоторое фиксированное количество раз), или когда количество ожидающих сканирования карт снизится до некоторого порогового значения. В результате данный параллельный и конкурентный шаг нормально завершится до того, как на следующем шаге будет завершен процесс отслеживания.

4. Последний шаг гарантирует, что все достижимые объекты будут найдены и соответствующим образом помечены. Все мутаторы в этот момент остановлены, так что корневые множества всех потоков могут быть быстро найдены с использованием всех процессоров системы. Поскольку достижимость большинства объектов уже отслежена, в состояние *Несканирован*, как ожидается, будет помещено только небольшое количество объектов. После этого все потоки принимают участие в определении оставшихся достижимых объектов и повторном сканировании всех карт.

Важно управлять скоростью отслеживания. Этот процесс напоминает гонку — мутаторы создают новые объекты и новые ссылки, которые должны быть сканированы, а отслеживающие потоки в это же время пытаются сканировать все достижимые объекты и повторно сканировать помечаемые мутаторами карты. Плохо, если отслеживание начинается задолго до того, когда требуется выполнение сборки мусора, поскольку это приводит к увеличению количества плавающего мусора. С другой стороны, нельзя ожидать исчерпания памяти, поскольку в этом случае мутаторы не смогут продолжать свою работу и конкурентная сборка мусора выродится в обычную сборку при блокированном выполнении основной программы. Следовательно, алгоритм должен уделить особое внимание выбору времени начала сборки мусора и скорости отслеживания. При принятии данно-

го решения можно воспользоваться данными о работе мутаторов в предыдущих циклах сборки мусора.

7.8.2 Частичное перемещение объектов

Как говорилось выше, начиная с раздела 7.6.4, копирующие или уплотняющие сборщики мусора обладают тем преимуществом, что устраняют фрагментацию памяти. Однако для таких сборщиков характерны существенные накладные расходы. Уплотняющий сборщик мусора должен переместить все объекты и обновить все ссылки по окончании сборки. Копирующий сборщик выясняет новое местоположение объекта в процессе отслеживания; при инкрементном отслеживании требуется либо транслировать все обращения мутатора к объекту, либо перемещать все объекты и обновлять все ссылки на них по окончании сборки. И тот, и другой подходы очень дорогостоящи, в особенности для кучи больших размеров.

Однако можно воспользоваться копирующим сборщиком мусора по поколениям. Он весьма эффективен для сборки молодых объектов и снижения фрагментации, но при сборке старых объектов может оказаться дорогостоящим. Для ограничения количества старых объектов, анализируемых таким сборщиком, можно использовать алгоритм поезда. Однако накладные расходы алгоритма поезда весьма чувствительны к размеру запомненного множества каждого раздела.

Существует гибридная схема сборки мусора, которая использует конкурентное отслеживание для утилизации всех недостижимых объектов и в то же время перемещает только часть объектов. Такой метод снижает фрагментацию с меньшими расходами, чем полное перемещение всех объектов в каждом цикле сборки мусора.

1. Перед началом сборки выберите часть кучи, которая будет освобождена.

2. При маркировке достижимого объекта следует также запомнить все ссылки, указывающие на объекты в выбранной части.

3. По завершении отслеживания выполняется параллельное освобождение памяти, занятой недостижимыми объектами.

4. Наконец переносятся все достижимые объекты из выбранной части и исправляются все запомненные ссылки на них.

7.8.3 Консервативная сборка мусора для небезопасных языков программирования

Как говорилось в разделе 7.5.1, невозможно создать сборщик мусора, который гарантировал бы работу во всех программах на C и C++. Поскольку адрес

может быть вычислен при помощи арифметических операций, никакая ячейка памяти в программе на C или C++ не может считаться недостижимой. Однако многие программы на этих языках программирования никогда не создают адреса подобным образом. Можно показать, что для данного класса программ возможно построение вполне работоспособного на практике консервативного сборщика мусора (который необязательно удаляет весь мусор).

Консервативный сборщик мусора предполагает, что невозможно создать или вывести адрес некоторого выделенного блока памяти, не имея какого-либо иного адреса, указывающего на какую-либо ячейку памяти в том же блоке. Весь мусор в программе при таком допущении можно найти, рассматривая любую битовую последовательность в достижимой памяти как корректный адрес, если только эта битовая последовательность вообще может рассматриваться как адрес. Такая схема может ошибочно классифицировать некоторые данные как адрес. Однако в этом нет ничего страшного, поскольку это приводит только к излишней консервативности сборщика и сохранению большего количества объектов, чем это необходимо на самом деле.

Перемещение объектов, требующее обновления всех ссылок на старое местоположение объекта, не совместимо с принципом действия консервативного сборщика мусора. Поскольку такой сборщик мусора не знает, является ли конкретная битовая последовательность ссылкой на реальный адрес или это просто совпадение, он не имеет права изменять такие последовательности.

Вот как работает консервативный сборщик мусора. Сначала диспетчер памяти модифицируется таким образом, чтобы поддерживать *карту данных* (data map) для всех выделенных блоков памяти. Эта карта позволяет легко определять границы блока памяти по адресу любой ячейки памяти этого блока. Отслеживание начинается со сканирования корневого множества программы, чтобы определить все битовые последовательности, имеющие вид адресов (не беспокоясь при этом об их реальном типе). Проверяя эти потенциальные адреса с использованием карты данных, можно определить начальные адреса блоков памяти, которые достижимы с использованием указанных адресов, и перевести эти блоки в состояние *Несканирован*. Затем выполняется сканирование всех несканированных блоков памяти для поиска новых (вероятно) достижимых блоков памяти и помещения их в список блоков для сканирования, и так до тех пор, пока список не опустеет. После этого выполняется освобождение всех недостижимых блоков памяти с применением карты данных.

7.8.4 Слабые ссылки

Иногда программисты используют языки со сборкой мусора, но при этом хотят управлять памятью (или ее частями) самостоятельно. Иначе говоря, программист

может знать, что к некоторым объектам обращений больше не будет, несмотря на наличие ссылок на них. Пример из области компиляции поясняет сказанное.

Пример 7.17. Мы уже сталкивались с тем, что лексический анализатор работает с таблицей символов, создавая объект для каждого обнаруженного идентификатора. Эти объекты могут быть, например, лексическими значениями, связанными с представляющими соответствующие идентификаторы листьями дерева разбора. Однако для поиска этих объектов стоит создать также хеш-таблицу, ключом в которой является строка идентификатора. Такая таблица ускоряет поиск объекта лексическим анализатором, когда последний встречает лексему, представляющую собой идентификатор.

После того как компилятор выходит за пределы области видимости идентификатора I, в дереве разбора (или в любой иной промежуточной структуре, используемой компилятором) не остается ссылок на соответствующий объект в таблице символов. Однако в хеш-таблице ссылка на данный объект все еще остается. Поскольку хеш-таблица является частью корневого множества компилятора, объект не может быть утилизирован сборщиком мусора. Если в программе встретится другой идентификатор с той же лексемой I, то выяснится, что I находится вне области видимости, и ссылка на его объект будет удалена. Однако, если такой лексемы в программе больше нет, то при том, что объект идентификатора I будет невостребованным в процессе компиляции, сборщик мусора не сможет освободить выделенную ему память. □

Если проблема, описанная в примере 7.17, оказывается существенной, то разработчик компилятора может пойти по пути удаления из хеш-таблицы всех ссылок на объекты при завершении их областей видимости. Однако метод, известный как метод *слабых ссылок* (weak references), позволяет программисту положиться на автоматическую сборку мусора, чтобы освобождать кучу от достижимых, но на самом деле не используемых объектов. Такая система позволяет объявлять некоторые ссылки как "слабые". Примером могут служить только что рассмотренные ссылки в хеш-таблице. Когда сборщик мусора сканирует некоторый объект, он пропускает слабые ссылки в этом объекте и не делает объекты, на которые они указывают, достижимыми. Само собой, такие объекты могут быть достижимыми, если на них имеются другие ссылки, не являющиеся слабыми.

7.8.5 Упражнения к разделу 7.8

! **Упражнение 7.8.1.** В разделе 7.8.3 было сказано, что в программе на C сборка мусора возможна, если выражения, которые указывают на место в блоке памяти, могут быть в программе только при условии наличия в той же программе адреса некоторого места в том же блоке памяти. Таким образом, мы исключаем код наподобие

```
p = 12345;
x = *p;
```

поскольку p может случайно указывать на некоторый блок памяти при отсутствии других указателей на память в этом блоке. С другой стороны, высока вероятность, что p указывает "в никуда", так что выполнение указанного кода приведет к ошибке обращения к несуществующей памяти. Тем не менее в C можно написать такой код, в котором переменная типа p будет гарантированно указывать на некоторый блок памяти, при том, что указателя на этот блок памяти в программе не будет. Напишите такую программу.

7.9 Резюме к главе 7

✦ *Организация времени выполнения.* Для реализации абстракций, воплощенных в определении исходного языка программирования, компилятор создает среду времени выполнения во взаимодействии с операционной системой и целевой машиной и управляет ею. Среда времени выполнения имеет статические области данных для объектного кода и статические объекты данных, созданные в процессе компиляции. В ней также имеются динамические области стека и кучи для управления объектами, создаваемыми и уничтожаемыми в процессе выполнения целевой программы.

✦ *Стек управления.* Вызовы процедур и возвраты из них обычно осуществляются с использованием стека времени выполнения, именуемого *стеком управления*. Применение стека обусловливается тем, что вызовы процедур, или *активации*, вложены во времени, т.е. если процедура p вызывает процедуру q, то данная активация q вложена в активацию p.

✦ *Выделение памяти в стеке.* В языках программирования, которые позволяют локальным переменным быть недоступными после завершения процедуры (или требуют этого), память для локальных переменных может быть выделена в стеке времени выполнения. В таких языках каждая активная в настоящий момент активация имеет в стеке управления *запись активации* (или *кадр*), причем корень дерева активаций находится на дне стека, а последовательность записей активации в стеке соответствует пути в дереве активаций от корня к активации, выполняющейся в настоящее время процедуры. Запись последней активации находится на вершине стека управления.

✦ *Доступ к нелокальным данным в стеке.* Для языков программирования наподобие C, в которых запрещены вложенные объявления процедур, все переменные либо являются глобальными, либо находятся в записи активации

на вершине стека времени выполнения. В случае языков программирования с вложенными процедурами возможно обращение к нелокальным данным в стеке посредством *связей доступа*, которые представляют собой указатели, добавляемые к каждой записи активации. Требуемые нелокальные данные можно найти, проследовав по цепочке связей доступа к необходимой записи активации. Вместе со связями доступа может использоваться *дисплей*, который представляет собой вспомогательный массив, обеспечивающий эффективную альтернативу цепочке указателей.

✦ *Управление кучей. Куча* представляет собой часть памяти, использующуюся для данных с неограниченным временем жизни (или временем, ограниченным самой программой, которая явно уничтожает эти данные). *Диспетчер памяти* выделяет память в куче и освобождает ее. *Сборка мусора* находит в куче пространство, которое больше не используется и может быть выделено для размещения в нем других данных. В языках с использованием сборки мусора последняя является важной подсистемой управления памятью.

✦ *Использование локальности.* Эффективно используя иерархию памяти, диспетчеры памяти могут существенно влиять на время работы программы. Время, затрачиваемое на доступ к различным частям памяти, может варьироваться от наносекунд до миллисекунд. К счастью, большинство программ, в основном, затрачивают время на выполнение относительно малых частей кода и работают только с малыми частями данных. Программа обладает *временной локальностью*, если высока вероятность, что в ближайшее время она обратится к тем же данным, что и в настоящий момент; *пространственная локальность* означает, что, скорее всего, программа в ближайшее время обратится к данным, находящимся рядом с теми данными, с которыми она работает в настоящий момент.

✦ *Снижение фрагментации.* В процессе выделения и освобождения памяти куча может стать *фрагментированной*, т.е. разбитой на большое количество несмежных блоков свободной памяти. Эмпирически доказано, что неплохо работает стратегия *наилучшего подходящего*, состоящая в выделении наименьшего из доступных блоков свободной памяти, удовлетворяющих запросу. Однако при том, что эта стратегия обычно эффективнее других использует свободную память, с точки зрения пространственной локальности она оказывается не самой лучшей. Фрагментация может быть уменьшена путем слияния соседних свободных блоков памяти.

✦ *Освобождение памяти вручную.* Управление памятью вручную обычно приводит к двум распространенным ошибкам: "неудаление данных", к ко-

торым нельзя обратиться, приводит к *утечке памяти*, а обращение к удаленным данным — к ошибке *разыменования висящего указателя*.

- *Достижимость. Мусором* называются *недостижимые* данные, к которым невозможно обратиться. Существует два основных способа поиска недостижимых объектов: перехват перехода достижимых объектов в недостижимые и периодическое выявление всех достижимых объектов (все остальные объекты — недостижимые).

- *Сборщики с подсчетом ссылок* поддерживают счетчик количества ссылок на объект; когда количество ссылок становится равным нулю, объект становится недостижимым. Такие сборщики характеризуются дополнительными накладными расходами на поддержание ссылок и могут некорректно обрабатывать ситуацию "циклического мусора", который состоит из недостижимых объектов, указывающих друг на друга, возможно, посредством цепочки ссылок.

- *Сборщики мусора на основе отслеживания* итеративно исследуют, или отслеживают, все ссылки для выявления достижимых объектов, начиная с *корневого множества*, состоящего из объектов, обращение к которым может быть выполнено непосредственно, без разыменования каких-либо указателей.

- *Сборщики "пометить и подмести"* на первом этапе посещают и помечают все достижимые объекты, а затем "подметают" кучу, освобождая память, занятую недостижимыми объектами.

- *Сборщики "пометить и сжать"* являются усовершенствованием сборщиков "пометить и подмести"; они *перемещают* достижимые объекты в куче для устранения фрагментации памяти.

- *Копирующие сборщики* отделяют отслеживание от поиска свободной памяти. Они разделяют память на *полупространства* A и B. Запросы на выделение памяти удовлетворяются из одного полупространства, скажем, A, пока оно не будет исчерпано. В этот момент в действие вступает сборщик мусора, который копирует достижимые объекты в полупространство B, после чего меняет роли полупространств.

- *Инкрементные сборщики.* Простые сборщики, основанные на отслеживании, приостанавливают выполнение программы на время сборки мусора. *Инкрементные сборщики* чередуют свои действия по сборке мусора с работой *мутатора*, или пользовательской программы. Мутатор может влиять

на инкрементный анализ достижимости, изменяя ссылки в уже отсканированных объектах. Таким образом, инкрементные сборщики мусора обеспечивают безопасность определенной переоценкой множества достижимых объектов; появляющийся "плавающий мусор" может быть собран в очередном цикле сборки мусора.

✦ *Частичные сборщики* также уменьшают паузы в основной программе; за один цикл они собирают только часть мусора. Среди алгоритмов частичной сборки наиболее известна *сборка мусора по поколениям*, подразделяющая объекты в соответствии с их "возрастом" и выполняющая сборку среди новых объектов чаще, чем среди старых, поскольку обычно время жизни объектов невелико. Другой алгоритм — *алгоритм поезда* — использует разделы фиксированного размера, именуемые *вагонами*, собранные в *поезда*. Каждый цикл сборки работает с первым оставшимся вагоном первого оставшегося поезда. После сборки мусора в вагоне достижимые объекты перемещаются из него в другие вагоны, так что вагон остается только с мусором и удаляется из поезда. Эти два алгоритма используются совместно для создания частичного сборщика, в котором к молодым объектам применяется сборка по поколениям, а к старым — алгоритм поезда.

7.10 Список литературы к главе 7

В математической логике правила областей видимости и передача параметров подстановкой датируются выходом работы Фрега (Frege) [8]. Лямбда-исчисление Черча (Church) [3] использует лексическую область видимости; оно используется в качестве модели для изучения языков программирования. Лексические области видимости используют Algol 60 и его наследники, включая C и Java. Будучи использованной в начальной реализации Lisp, динамическая область видимости стала неотъемлемым свойством этого языка; описание этой истории можно прочесть у Мак-Карти (McCarthy) [14].

Многие концепции, связанные с выделением памяти в стеке, были разработаны в связи с блоками и рекурсией в языке программирования Algol 60. Идея дисплея для доступа к нелокальным переменным в языке с лексическими областями видимости принадлежит Дейкстре (Dijkstra) [5]. Детальное описание выделения памяти в стеке, использование дисплеев и динамическое выделение памяти для массивов можно найти в книге Ренделла (Randell) и Рассела (Russell) [16]. Джонсон (Johnson) и Риччи (Ritchie) [10] рассматривают дизайн вызывающей последовательности, которая позволяет количеству аргументов процедуры изменяться от вызова к вызову.

Сборка мусора была областью активных исследований; см., например, работу Вильсона (Wilson) [17]. Счетчики ссылок впервые описаны в работе Коллинза

(Collins) [4], а сборка мусора на основе отслеживания — в работе Мак-Карти (McCarthy) [13], который описал алгоритм "пометить и подмести" для ячеек фиксированного размера. Использование дескрипторов границ для управления свободной памятью было предложено Кнутом (Knuth) в 1962 году и опубликовано в [11].

Алгоритм 7.14 основан на работе Бейкера (Baker) [1], а алгоритм 7.16 — на нерекурсивной версии копирующего сборщика Феничела (Fenichel) и Йочельсона (Yochelson) [7], разработанной Чени (Cheney) [2].

Инкрементный анализ достижимости открыт Дейкстрой (Dijkstra) с коллегами [6]. Либерман (Lieberman) и Хьюитт (Hewitt) [12] представили сборщик мусора по поколениям, как расширение копирующей сборки. Алгоритм поезда разработан Хадсоном (Hudson) и Моссом (Moss) [9].

1. Baker, H. G. Jr., "The treadmill: real-time garbage collection without motion sickness", *ACM SIGPLAN Notices* **27**:3 (Mar., 1992), pp. 66–70.

2. Cheney, C. J., "A nonrecursive list compacting algorithm", *Comm. ACM* **13**:11 (Nov., 1970), pp. 677–678.

3. Church, A., *The Calculi of Lambda Conversion*, Annals of Math. Studies, No. 6, Princeton University Press, Princeton, N. J., 1941.

4. Collins, G. E., "A method for overlapping and erasure of lists", *Comm. ACM* **2**:12 (Dec., 1960), pp. 655–657.

5. Dijkstra, E. W., "Recursive programming", Numerische Math. 2 (1960), pp. 312–318.

6. Dijkstra, E. W., L. Lamport, A. J. Martin, C. S. Scholten, and E. F. M. Steffens, "On-the-fly garbage collection: an exercise in cooperation", *Comm. ACM* **21**:11 (1978), pp. 966–975.

7. Fenichel, R. R. and J. C. Yochelson, "A Lisp garbage-collector for virtual-memory computer systems", *Comm. ACM* **12**:11 (1969), pp. 611–612.

8. Frege, G., "Begriffschrift, a formula language, modeled upon that of arithmetic, for pure thought", (1879). In J. van Heijenoort, *From Frege to Godel*, Harvard Univ. Press, Cambridge MA, 1967.

9. Hudson, R. L. and J. E. B. Moss, "Incremental Collection of Mature Objects", *Proc. Intl. Workshop on Memory Management*, Lecture Notes In Computer Science **637** (1992), pp. 388–403.

10. Johnson, S. C. and D. M. Ritchie, "The C language calling sequence", Computing Science Technical Report 102, Bell Laboratories, Murray Hill NJ, 1981.

11. Knuth, D. E., *Art of Computer Programming, Volume 1: Fundamental Algorithms*, Addison-Wesley, Boston MA, 1968.

12. Lieberman, H. and C. Hewitt, "A real-time garbage collector based on the lifetimes of objects", *Comm. ACM* **26**:6 (June, 1983), pp. 419–429.

13. McCarthy, J., "Recursive functions of symbolic expressions and their computation by machine", *Comm. ACM* **3**:4 (Apr., 1960), pp. 184–195.

14. McCarthy, J., "History of Lisp." See pp. 173–185 in R. L.Wexelblat (ed.), *History of Programming Languages*, Academic Press, New York, 1981.

15. Minsky, M., "A LISP garbage collector algorithm using secondary storage", A. I. Memo 58, MIT Project MAC, Cambridge MA, 1963.

16. Randell, B. and L. J. Russell, *Algol 60 Implementation*, Academic Press, New York, 1964.

17. Wilson, P. R., "Uniprocessor garbage collection techniques", `ftp://ftp.cs.utexas.edu/pub/garbage/bigsurv.ps`.

Глава 8

Генерация кода

Последней стадией нашей модели компиляции является генератор кода. Он получает на вход промежуточное представление исходной программы от начальной стадии компилятора и выводит эквивалентную целевую программу, как показано на рис. 8.1.

Рис. 8.1. Положение генератора кода в модели компилятора

К генератору кода предъявляются жесткие требования. Получаемый код должен сохранять семантическое значение исходной программы и быть высококачественным, что означает эффективное использование доступных ресурсов целевой машины. Кроме того, эффективно должен работать и сам генератор кода.

Математически проблема генерации оптимальной целевой программы для данной исходной является неразрешимой; многие из подзадач, встречающихся при генерации кода, таких как распределение регистров, вычислительно трудноразрешимы. На практике мы вынуждены довольствоваться эвристическими методами, генерирующими хороший, но не обязательно оптимальный код. К счастью, эти эвристики достаточно стары и проверены, так что тщательно разработанный генератор может давать код в несколько раз более быстрый, чем получаемый от простейшего генератора без их применения.

Компиляторы, которые должны давать эффективные целевые программы, перед началом генерации кода включают фазу оптимизации. Оптимизатор превращает одно промежуточное представление в другое, из которого может быть сгенерирован более эффективный код. В общем случае фазы оптимизации и генерации кода, известные как заключительная стадия компилятора, могут выполнять несколько проходов по промежуточному представлению перед генерацией целевой программы. Подробнее оптимизация кода рассматривается в главе 9. Представленные в данной главе методы могут как использовать оптимизацию перед началом генерации целевого кода, так и обходиться без нее.

Перед генератором кода стоят три основные задачи: выбор команд, распределение и назначение регистров и упорядочение команд. Важность этих задач рассмат-

ривается в разделе 8.1. Выбор команд означает выбор машинных команд целевой машины для реализации инструкций промежуточного представления. Распределение и назначение регистров означает принятие решения о том, какие значения в каких регистрах будут храниться. Упорядочение команд предусматривает принятие решения о том, в каком порядке должны выполняться сгенерированные команды.

В этой главе приведены алгоритмы, которые генератор кода может использовать для трансляции промежуточного представления в последовательность команд целевого языка простой регистровой машины. Алгоритмы иллюстрируются с использованием описанной в разделе 8.2 модели машины. В главе 10 проблема генерации кода рассматривается в применении к сложным современным машинам, поддерживающим параллельность на уровне отдельных инструкций.

После основных вопросов разработки генераторов кода мы рассмотрим, какой целевой код должен генерироваться компилятором для поддержки абстракций, воплощенных в типичном исходном языке. В разделе 8.3 будут описаны реализация статических областей данных и выделение памяти для данных в стеке, а также показано, как имена в промежуточном представлении могут быть преобразованы в адреса в целевом коде.

Многие генераторы кода разделяют команды промежуточного представления на базовые блоки (basic block), состоящие из последовательности команд, которые всегда выполняются совместно. Разделение промежуточного представления на базовые блоки является темой раздела 8.4. В следующем разделе приведены простые локальные преобразования, которые могут использоваться для превращения базовых блоков в модифицированные базовые блоки, из которых может генерироваться более эффективный код. Эти преобразования представляют собой рудиментарную форму оптимизации кода; более глубоко теория оптимизации не будет затрагиваться до главы 9. Примером практичной локальной трансформации может служить выявление общих подвыражений на уровне промежуточного кода, приводящее к замене арифметических операций более простыми операциями копирования.

В разделе 8.6 представлен простой алгоритм генерации кода, который поочередно генерирует код для каждой инструкции, храня операнды в регистрах до тех пор, пока это возможно. Выход такого генератора кода легко улучшается при помощи методов локальной оптимизации, подобных рассматриваемым в разделе 8.7.

Оставшиеся разделы посвящены выбору команд и распределению регистров.

8.1 Вопросы проектирования генератора кода

Такие задачи, как выбор команд, распределение регистров и упорядочение команд, стоят практически перед всеми генераторами кода, в то время как их конкретные детали зависят от промежуточного представления, целевого языка и системы времени выполнения.

Наиболее важным критерием оценки генератора кода является корректность получающегося кода. Корректность приобретает особую важность в связи с наличием массы частных случаев, с которыми может столкнуться генератор. При сверхприоритетности корректности получающегося кода генераторы кода должны проектироваться так, чтобы их можно было легко реализовывать, тестировать и поддерживать.

8.1.1 Вход генератора кода

Входной поток генератора кода – это промежуточное представление исходной программы, полученное на начальной стадии компиляции, вместе с информацией в таблице символов, которая используется для определения адресов времени выполнения объектов данных, обозначаемых в промежуточном представлении именами.

Имеется много вариантов промежуточных представлений, включая такие трехадресные представления, как четверки, тройки, косвенные тройки; представление виртуальной машины, такое как байт-код или код стековой машины; линейное представление, например постфиксная запись; и графическое представление в виде синтаксических деревьев и ориентированных графов. Многие алгоритмы в этой главе излагаются с использованием представлений, рассмотренных в главе 6, – трехадресных кодов, деревьев и графов. Методы, которые мы рассмотрим, применимы и к другим промежуточным представлениям.

В этой главе мы считаем, что перед генерацией кода начальной стадией компиляции были выполнены сканирование, разбор и трансляция исходной программы в относительно низкоуровневое промежуточное представление, так что значения имен в промежуточном языке могут быть представлены величинами, с которыми целевая машина может работать непосредственно, такими, например, как целые числа и числа с плавающей точкой. Мы также полагаем, что были выявлены все синтаксические и статические семантические ошибки, выполнены все необходимые проверки типов, так что необходимые операторы преобразования типов находятся на своих местах. Таким образом, стадия генерации кода может работать в предположении, что ее вход не содержит ошибок.

8.1.2 Целевая программа

Архитектура набора команд целевой машины существенно влияет на сложность разработки хорошего генератора кода, дающего высококачественный машинный код. Наиболее распространенными архитектурами являются RISC (reduced instruction set computer — компьютер с сокращенным набором команд), CISC (complex instruction set computer — компьютер со сложным набором команд) и стековая.

RISC-машина обычно имеет много регистров, трехадресные команды, простые режимы адресации и относительно простой набор команд. CISC-машины, напротив, имеют мало регистров, двухадресные команды, множество режимов адресации, несколько классов регистров, команды переменной длины и команды с побочными действиями.

В стековых машинах операции выполняются путем внесения операндов в стек с последующим выполнением операций над операндами на вершине стека. Для достижения высокой производительности вершина стека обычно хранится в регистрах. Стековых машин практически не осталось, поскольку стековая организация слишком ограничена и требует слишком большого количества операций обмена и копирования.

Однако стековая архитектура ожила с появлением виртуальной машины Java (Java Virtual Machine — JVM). JVM представляет собой программный интерпретатор байт-кода Java, промежуточного языка, выдаваемого компиляторами Java. Интерпретатор обеспечивает многоплатформенную совместимость программ, основной фактор успеха Java.

Для преодоления снижения производительности из-за интерпретации, которое может достигать 10 раз, были созданы *оперативные* (just-in-time — JIT) компиляторы Java. Такие компиляторы во время выполнения программы транслируют байт-код в машинные команды целевого компьютера. Еще один способ повышения производительности Java состоит в создании компилятора, который выполняет компиляцию непосредственно в машинные команды, минуя стадию байт-кода.

Генерация абсолютной программы на машинном языке имеет то преимущество, что она может быть помещена в фиксированное место в памяти и тут же выполнена. Программа может быть быстро скомпилирована и выполнена.

Генерация переносимой программы на машинном языке (часто именуемой *объектным модулем* (object module)) обеспечивает возможность раздельной компиляции подпрограмм. Набор переместимых объектных модулей может быть скомпонован в одно целое и загружен для выполнения. Ценой дополнительных действий по компоновке и загрузке объектных модулей мы получаем гибкое решение, которое допускает раздельную компиляцию подпрограмм и вызов из объектного модуля других ранее скомпилированных программ. Если целевая машина не обрабатывает перемещение автоматически, компилятор должен явно предоставить

необходимую информацию загрузчику для компоновки отдельно скомпилированных модулей.

Несколько проще получение в качестве выхода генератора программы на языке ассемблера. При этом можно генерировать символьные команды и использовать возможности макросов ассемблера при генерации кода. Платой за эту простоту является дополнительный шаг ассемблирования по окончании генерации кода.

В этой главе мы будем рассматривать в качестве целевой машины очень простой RISC-образный компьютер. Мы добавим к нему некоторые CISC-образные режимы адресации, чтобы иметь возможность одновременно рассмотреть и методы генерации кода для CISC-машин. Для удобочитаемости в качестве целевого будет использован язык ассемблера. Поскольку адреса могут быть вычислены из смещений и другой информации, хранящейся в таблице символов, генератор кода может производить переносимые или абсолютные адреса для имен так же легко, как и символьные адреса.

8.1.3 Выбор команд

Генератор кода должен отобразить программу в промежуточном представлении на последовательность машинных команд, которые могут быть выполнены целевой машиной. Сложность этого отображения определяется такими факторами, как

- уровень промежуточного представления;
- природа архитектуры набора команд;
- требуемое качество генерируемого кода.

Если используется высокоуровневое промежуточное представление, то генератор кода может транслировать каждую инструкцию промежуточного представления в последовательность машинных команд с использованием шаблонов кода. Однако генерация кода инструкция за инструкцией зачастую дает низкокачественный код, требующий дальнейшей оптимизации. Если промежуточное представление отражает некоторые низкоуровневые особенности машины, то генератор кода может использовать эту информацию для генерации более эффективной последовательности команд.

Природа набора команд целевой машины сильно влияет на сложность выбора инструкций. Например, важными факторами являются единообразие и полнота набора команд. Если целевая машина не поддерживает единообразно все типы данных, то каждое исключение из общего правила требует отдельной обработки. На некоторых машинах, например, операции с числами с плавающей точкой выполняются с использованием отдельных регистров.

Другими важными факторами являются скорость выполнения команд и идиомы языка целевой машины. Если мы не беспокоимся об эффективности целевой программы, выбор инструкций достаточно прост. Для каждого типа трехадресных инструкций можно разработать шаблон целевого кода, генерируемого для данной конструкции. Например, каждая трехадресная инструкция вида `x=y+z`, где `x`, `y` и `z` распределяются статически, может быть транслирована в следующий код.

```
LD   R0, y        // R0 = y          (загрузка y в регистр R0)
ADD  R0, R0, z    // R0 = R0 + z     (прибавление z к R0)
ST   x, R0        // x = R0          (сохранение R0 в x)
```

Такая стратегия часто приводит к избыточным сохранениям и загрузкам. Например, последовательность трехадресных инструкций

```
a = b + c
d = a + e
```

будет транслирована в

```
LD   R0, b        // R0 = b
ADD  R0, R0, c    // R0 = R0 + c
ST   a, R0        // a = R0
LD   R0, a        // R0 = a
ADD  R0, R0, e    // R0 = R0 + e
ST   d, R0        // d = R0
```

Четвертая команда в данном случае совершенно излишня, поскольку она загружает значение, которое только что было сохранено; если в дальнейшем `a` не используется, то излишня и третья команда.

Качество сгенерированного кода обычно определяется его скоростью выполнения и размером. На большинстве машин заданная программа в промежуточном представлении может быть реализована множеством различных последовательностей кодов, существенно отличающихся друг от друга. Непосредственная простейшая трансляция промежуточного кода может, таким образом, давать корректный, но неприемлемо неэффективный код.

Например, если целевая машина имеет команду инкремента (`INC`), то трехадресная инструкция `a=a+1` может быть более эффективно реализована одной командой `INC a` вместо обычной последовательности, состоящей из загрузки `a` в регистр, прибавления к регистру 1 и сохранения результата в `a`:

```
LD   R0, a        // R0 = a
ADD  R0, R0, #1   // R0 = R0 + 1
ST   a, R0        // a = R0
```

Для получения эффективной последовательности команд необходимо знать стоимость каждой команды; к сожалению, получить точную информацию зачастую весьма сложно. Принятие решения о том, какая именно последовательность машинных команд наилучшим образом подходит для данной трехадресной конструкции, может потребовать также знания контекста, в котором появляется эта конструкция.

В разделе 8.9 мы увидим, что такой выбор команд может быть смоделирован с использованием представления машинных команд и инструкций промежуточного представления в виде деревьев. Затем мы пытаемся "покрыть" дерево промежуточного представления множеством поддеревьев, соответствующих машинным командам. Если каждому поддереву машинной команды назначить стоимость, то для генерации оптимальной последовательности команд можно использовать динамическое программирование.

8.1.4 Распределение регистров

Ключевой проблемой при генерации кода является принятие решения о том, какие значения в каких регистрах должны храниться. Регистры представляют собой наиболее быстрые вычислительные модули целевой машины, но обычно их слишком мало, чтобы хранить все значения. Значения, которые не хранятся в регистрах, должны находиться в памяти. Команды, использующие в качестве операндов регистры, обычно короче и выполняются быстрее, чем команды, работающие с операндами, расположенными в памяти. Следовательно, эффективное использование регистров — еще одна важная составляющая генерации хорошего целевого кода.

Использование регистров часто разделяется на две подзадачи.

1. В процессе *распределения регистров* (register allocation) мы выбираем множество переменных, которые будут находиться в регистрах в каждой точке программы.

2. В последующей фазе *назначения регистров* (register assignment) мы выбираем конкретные регистры для размещения в них переменных.

Поиск оптимального назначения регистров переменным представляет собой сложную задачу даже на машине с единственным регистром. Математически эта задача — NP-полная. Проблема усложняется еще и тем, что аппаратное обеспечение и/или операционная система целевой машины может накладывать дополнительные ограничения по использованию регистров.

Пример 8.1. Некоторые машины требуют для некоторых операндов и результатов операций *пар регистров* (регистра с четным номером и регистра со следующим за ним нечетным номером). Например, на некоторых машинах целочисленные

умножение и деление требуют использования пар регистров. Команда умножения, например, имеет вид

```
M x, y
```

Здесь сомножитель x представляет собой нечетный регистр пары, состоящей из четного и нечетного регистров, а сомножитель y может храниться в любом регистре. Произведение занимает пару из четного и нечетного регистров. Команда деления имеет вид

```
D x, y
```

Здесь делимое занимает пару регистров, четный регистр которой — x; y представляет собой делитель. После деления в четном регистре хранится остаток, а в нечетном — частное.

Рассмотрим теперь две последовательности трехадресных кодов на рис. 8.2, в которых единственным отличием между последовательностями *а* и *б* является оператор во второй инструкции. Кратчайшие последовательности ассемблерных кодов для *а* и *б* приведены на рис. 8.3.

```
t = a + b          t = a + b
t = t * c          t = t + c
t = t / d          t = t / d
```

а) б)

Рис. 8.2. Две последовательности трехадресных кодов

```
L   R1, a          L     R0, a
A   R1, b          A     R0, b
M   R0, c          A     R0, c
D   R0, d          SRDA  R0, 32
ST  R1, t          D     R0, d
                   ST    R1, t
```

а) б)

Рис. 8.3. Оптимальные последовательности машинных кодов

R*i* означает *i*-й регистр; SRDA — сдвиг вправо (Shift-Right-Double-Arithmetic); команда SRDA R0,32 сдвигает делимое в R1 и очищает R0, делая все биты равными его знаковому биту. Команды L, ST и A означают соответственно загрузку в регистр, сохранение регистра и суммирование. Заметим, что оптимальный выбор регистра для загрузки a зависит от того, что в конечном счете произойдет с t. □

Стратегии распределения регистров рассматриваются в разделе 8.8. В разделе 8.10 показано, что для некоторых классов машин можно построить последова-

тельности кодов, которые будут выполнять вычисление выражения с использованием минимально возможного количества регистров.

8.1.5 Порядок вычислений

Порядок, в котором выполняются вычисления, также может существенно влиять на эффективность целевого кода. Как мы увидим, изменение порядка вычислений может привести к уменьшению количества регистров, необходимых для хранения промежуточных результатов. Однако в общем случае выбор оптимального порядка вычислений представляет собой сложную NP-полную задачу. Вначале мы избежим этой задачи, генерируя целевой код для трехадресных инструкций в том порядке, в каком они были созданы генератором промежуточного кода. В главе 10 будет рассмотрено планирование кода для конвейерных машин, которые способны выполнять несколько операций за один такт.

8.2 Целевой язык

Одним из непременных требований к построению хорошего генератора кода является близкое знакомство с целевой машиной и ее набором инструкций. К сожалению, при обсуждении общих вопросов генерации кода невозможно описать нюансы той или иной целевой машины достаточно подробно, чтобы иметь возможность генерировать для нее хороший код. В этой главе в качестве целевого языка мы используем ассемблерный код для простого компьютера, являющегося типичным представителем множества регистровых машин. Однако рассматриваемые здесь методы генерации кода применимы и к множеству других классов машин.

8.2.1 Простая модель целевой машины

Модель нашего целевого компьютера представляет собой трехадресную машину с операциями загрузки и сохранения регистров, вычислительными операциями, условными и безусловными переходами. Компьютер представляет собой машину с байтовой адресацией памяти с n регистрами общего назначения $R0, R1, \ldots, Rn-1$. Полнофункциональный язык ассемблера должен иметь множество команд. Чтобы не потерять базовые концепции за мириадами несущественных деталей, мы будем использовать очень ограниченное множество команд и считать, что все операнды представляют собой целые числа. Большинство команд состоит из оператора, за которым следуют приемник и список исходных операндов. Команде может предшествовать метка. Предполагается, что имеются команды следующих видов.

- Операции *загрузки*. Команда LD *dst, addr* загружает значение из ячейки памяти *addr* в *dst*. Эта команда описывает присваивание *dst* = *addr*. Наиболее распространенной формой этой команды является LD r, x, которая загружает значение из ячейки памяти x в регистр r. Команда вида LD r_1, r_2 представляет собой *копирование*, при котором содержимое регистра r_2 копируется в регистр r_1.

- Операция *сохранения*. Команда ST x, r сохраняет значение из регистра r в ячейке памяти x. Эта команда описывает присваивание $x = r$.

- *Вычислительные* операции вида OP dst, src_1, src_2, где OP — оператор наподобие ADD или SUB, а dst, src_1 и src_2 — местоположения данных, не обязательно различные. Результат выполнения команды состоит в применении операции OP к значениям в местоположениях src_1 и src_2 и помещении результата в dst. Например, команда SUB r_1, r_2, r_3 вычисляет $r_1 = r_2 - r_3$. Любое значение, ранее хранившееся в r_1, теряется, но если r_1 совпадает с r_2 или r_3, то сперва считывается старое значение. Унарная операция, принимающая только один операнд, не имеет операнда src_2.

- *Безусловные переходы*. Команда BR L приводит к передаче управления машинной команде с меткой L (BR означает "branch" — "переход").

- *Условные переходы* вида B*cond* r, L, где r — регистр, L — метка, а *cond* означает одну из обычных проверок значения в регистре r. Например, BLTZ r, L приводит к переходу к метке L, если значение в регистре r меньше нуля; в противном случае управление передается очередной машинной команде.

Наша целевая машина имеет несколько режимов адресации.

- В командах адрес может быть именем переменной x, что означает место в памяти, зарезервированное для x (т.е. для l-значения x).

- Адрес может быть индексированным адресом вида $a(r)$, где a — переменная, а r — регистр. Адрес $a(r)$ вычисляется путем прибавления к l-значению a значения из регистра r. Например, команда LD R1, a(R2) выполняет присваивание $R1 = contents\,(a + contents\,(\text{R2}))$, где $contents\,(x)$ означает содержимое регистра или ячейки памяти, представленной x. Этот режим адресации полезен для обращения к массивам, когда a представляет собой базовый адрес массива (т.е. адрес его первого элемента), а в r хранится количество байтов после этого адреса, которые надо пропустить, чтобы обратиться к некоторому элементу массива a.

- Адрес может быть индексирован регистром. Например, команда `LD R1, 100(R2)` выполняет присваивание R1 = *contents* (100 + *contents* (R2)), т.е. в регистр R1 загружается значение из ячейки памяти, адрес которой получается путем прибавления 100 к содержимому регистра R2. Эта адресация полезна при следовании по указателям, как будет видно в приведенном далее примере.

- Целевая машина допускает два режима косвенной адресации: **r* означает ячейку памяти, находящуюся по адресу, представленному содержимым регистра *r*, а `*100(`*r*`)` означает ячейку памяти, находящуюся по адресу, полученному путем прибавления 100 к содержимому *r*. Например, команда `LD R1, *100(R2)` выполняет присваивание R1 = *contents*(*contents*(100+ + *contents*(R2))), т.е. в регистр R1 загружается значение ячейки памяти, адрес которой хранится в ячейке памяти по адресу, равному 100 плюс содержимое регистра R2.

- Наконец, имеется режим адресации с использованием констант (для указания которых используется префикс #). Команда `LD R1, #100` загружает в регистр R1 целое число 100, а `ADD R1, R1, #100` прибавляет 100 к регистру R1.

Комментарии располагаются после команд и начинаются с символов `//`.

Пример 8.2. Трехадресная инструкция `x = y - z` может быть реализована машинными командами

```
LD  R1, y           // R1 = y
LD  R2, z           // R2 = z
SUB R1, R1, R2      // R1 = R1 - R2
ST  x, R1           // x = R1
```

Однако, вероятно, код можно улучшить. Одна из целей хорошего алгоритма генерации кода — по возможности избежать использования как можно большего количества команд. Например, `y` и/или `z` могут быть вычислены в регистрах, и это позволит обойтись без команд загрузки `LD`. Точно так же можно обойтись без команды сохранения, если значение `x` используется в дальнейших вычислениях в регистре и больше в программе не потребуется.

Предположим, что `a` — массив, элементы которого представляют собой 8-байтные значения, скажем, числа с плавающей точкой. Будем также считать, что элементы массива индексируются начиная с нулевого значения. Трехадресная команда `b = a[i]` может быть выполнена при помощи следующих машинных команд:

```
LD  R1, i          // R1 = i
MUL R1, R1, 8      // R1 = R1 * 8
LD  R2, a(R1)      // R2 = contents(a+contents(R1))
ST  b, R2          // b = R2
```

На втором шаге вычисляется $8i$, а на третьем в регистр R2 загружается значение i-го элемента a, который находится на расстоянии $8i$ байт от базового адреса массива a.

Аналогично присваивание элементу массива a, представленное трехадресной командой a[j] = c, реализуется такой последовательностью машинных команд:

```
LD  R1, c          // R1 = c
LD  R2, j          // R2 = j
MUL R2, R2, 8      // R2 = R2 * 8
ST  a(R2), R1      // contents(a+contents(R2)) = R1
```

Для реализации простого косвенного обращения с использованием указателя, такого как трехадресная инструкция x = *p, можно использовать машинные команды

```
LD  R1, p          // R1 = p
LD  R2, 0(R1)      // R2 = contents(0+contents(R1))
ST  x, R2          // x = R2
```

Присваивание с использованием указателя *p = y реализуется аналогичными командами

```
LD  R1, p          // R1 = p
LD  R2, y          // R2 = y
ST  0(R1), R2      // contents(0+contents(R1)) = R2
```

Наконец, рассмотрим трехадресную команду условного перехода наподобие

```
if x < y goto L
```

Эквивалентный машинный код будет выглядеть примерно так:

```
LD   R1, x         // R1 = x
LD   R2, y         // R2 = y
SUB  R1, R1, R2    // R1 = R1 - R2
BLTZ R1, M         // Если R1 < 0, переход к M
```

Здесь M — метка, представляющая первую машинную команду, сгенерированную трехадресной командой с меткой L. Как и в случае любой трехадресной команды, мы надеемся, что сможем сэкономить несколько машинных команд за счет того, что необходимые операнды уже находятся в регистрах или что сохранение результатов не потребуется. □

8.2.2 Стоимость программ и команд

Мы часто связываем стоимость с компиляцией и выполнением программ. В зависимости от того, оптимизация какого именно аспекта программы нас интересует, наиболее распространенными мерами стоимости являются продолжительность компиляции, размер целевой программы и время ее работы.

Определение фактической стоимости компиляции и работы программы представляет собой сложную задачу. Поиск оптимальной целевой программы для данной исходной в общем случае – задача неразрешимая, а многие ее подзадачи NP-сложные. Как уже говорилось, при генерации кода приходится удовлетворяться эвристическими методами, которые дают хорошие, но не обязательно оптимальные целевые программы.

В оставшейся части этой главы мы считаем, что каждая команда целевого языка имеет связанную с ней стоимость. Для простоты будем считать стоимость равной единице плюс стоимости, связанные с режимами адресации операндов. Такая стоимость соответствует длине команды в словах. Режимы адресации с использованием регистров имеют нулевую стоимость, а режимы с использованием ячеек памяти или констант – дополнительную стоимость, равную единице, поскольку такие операнды хранятся в словах, следующих за командой. Вот некоторые примеры.

- Команда `LD R0, R1` копирует содержимое регистра `R1` в регистр `R0`. Стоимость этой команды равна единице, поскольку для нее не требуются никакие дополнительные слова памяти.

- Команда `LD R0, M` загружает содержимое ячейки памяти `M` в регистр `R0`. Стоимость этой команды равна двум, поскольку адрес ячейки памяти `M` содержится в слове, следующем за командой.

- Команда `LD R1, *100(R2)` загружает в регистр `R1` значение *contents*(*contents*(100 + *contents*(`R2`))). Стоимость команды равна трем, так как константа 100 хранится в слове, следующем за командой.

В этой главе мы полагаем стоимость программы на целевом языке равной сумме стоимостей отдельных команд, выполняемых при работе программы с заданными входными данными. Хороший алгоритм генерации кода старается минимизировать сумму стоимостей команд, выполняемых целевой программой для типичных входных данных. Мы увидим, что в некоторых ситуациях для определенных классов регистровых машин возможна генерация оптимального кода для выражений.

8.2.3 Упражнения к разделу 8.2

Упражнение 8.2.1. Сгенерируйте код для следующих трехадресных инструкций в предположении, что все переменные хранятся в ячейках памяти.

а) `x = 1`

б) `x = a`

в) `x = a + 1`

г) `x = a + b`

д) Две инструкции:

```
x = b * c
y = a + x
```

Упражнение 8.2.2. Сгенерируйте код для следующих трехадресных инструкций в предположении, что `a` и `b` — массивы с элементами, размеры которых равны 4 байт.

а) Последовательность из четырех инструкций:

```
x = a[i]
y = b[j]
a[i] = y
b[j] = x
```

б) Последовательность из трех инструкций:

```
x = a[i]
y = b[i]
z = x * y
```

в) Последовательность из трех инструкций:

```
x = a[i]
y = b[x]
a[i] = y
```

Упражнение 8.2.3. Сгенерируйте код для следующих трехадресных инструкций в предположении, что `p` и `q` располагаются в ячейках памяти:

```
y = *q
q = q + 4
*p = y
p = p + 4
```

Упражнение 8.2.4. Сгенерируйте код для следующих трехадресных инструкций в предположении, что x, y и z располагаются в ячейках памяти:

```
    if x < y goto L1
    z = 0
    goto L2
L1: z = 1
```

Упражнение 8.2.5. Сгенерируйте код для следующих трехадресных инструкций в предположении, что n располагается в ячейке памяти:

```
    s = 0
    i = 0
L1: if i > n goto L2
    s = s + i
    i = i + 1
    goto L1
L2:
```

Упражнение 8.2.6. Определите стоимость приведенных ниже последовательностей команд.

а)
```
LD  R0, y
LD  R1, z
ADD R0, R0, R1
ST  x, R0
```

б)
```
LD  R0, i
MUL R0, R0, 8
LD  R1, a(R0)
ST  b, R1
```

в)
```
LD  R0, c
LD  R1, i
MUL R1, R1, 8
ST  a(R1), R0
```

г)
```
LD R0, p
LD R1, 0(R0)
ST x, R1
```

д)
```
LD R0, p
LD R1, x
ST 0(R0), R1
```

е)
```
LD   R0, x
LD   R1, y
SUB  R0, R0, R1
BLTZ *R3, R0
```

8.3 Адреса в целевом коде

В этом разделе мы покажем, каким образом имена в промежуточном представлении могут быть преобразованы в адреса в целевом коде, рассматривая генерацию кода простых вызовов процедур и возвратов из них с использованием статического распределения памяти и выделения памяти в стеке. В разделе 7.1 было рассказано, что каждая выполняющаяся программа работает в собственном логическом адресном пространстве, которое разбивается на четыре области для кода и данных.

1. Статически определенная область *Code*, в которой хранится выполнимый целевой код.

2. Статически определенная область *Static*, в которой хранятся глобальные константы и иные данные, генерируемые компилятором. Размер глобальных констант и данных компилятора также может быть определен в процессе компиляции.

3. Динамически управляемая область памяти *Heap* для хранения объектов данных, память для которых выделяется и освобождается в процессе выполнения программы. Размер области *Heap* не может быть определен во время компиляции.

4. Динамически управляемая область памяти *Stack* для хранения записей активации между их созданием и уничтожением в процессе вызова процедур и возвратов из них. Как и в случае кучи *Heap*, размер области *Stack* не может быть определен во время компиляции.

8.3.1 Статическое выделение памяти

Чтобы проиллюстрировать генерацию кода для упрощенных вызовов процедур и возвратов из них, остановимся на следующих трехадресных инструкциях.

- `call` *callee*

- `return`
- `halt`
- `action` (заменитель прочих трехадресных инструкций)

Размер и размещение записей активации определяется генератором кода на основании информации об именах, хранящейся в таблице символов. Сначала проиллюстрируем, каким образом в записи активации процедуры сохраняется адрес возврата и как передается управление процедуре после ее вызова. Для удобства будем считать, что в первой ячейке записи активации хранится адрес возврата.

Рассмотрим код, необходимый для реализации простейшего случая — статического выделения памяти. Инструкция `call` *callee* в промежуточном коде может быть реализована последовательностью двух команд целевой машины:

```
ST   callee.staticArea,  #$here + 20$
BR   callee.codeArea
```

Команда `ST` сохраняет адрес возврата в начале записи активации *callee*, а `BR` передает управление целевому коду вызываемой процедуры. Атрибут перед *callee.staticArea* представляет собой константу, которая дает адрес начала записи активации *callee*, а атрибут *callee.codeArea* представляет собой константу, указывающую адрес первой команды вызываемой процедуры *callee* в области *Code* памяти времени выполнения.

Операнд #*here*+20 в команде `ST` представляет собой адрес возврата, т.е. адрес команды, следующий за командой `BR`. Мы считаем, что #*here* — адрес текущей команды и что три константы и две команды в вызывающей последовательности составляют 5 слов, или 20 байт.

Код процедуры заканчивается возвратом в вызывающую процедуру, за исключением первой процедуры, не имеющей вызывающей ее процедуры. В этом случае конечная команда — `HALT`, которая передает управление операционной системе. Инструкция `return` *callee* может быть реализована простой командой перехода

```
BR   *callee.staticArea
```

Она передает управление по адресу, сохраненному в начале записи активации для *callee*.

Пример 8.3. Предположим, что имеется следующий трехадресный код:

```
                    // Код c
action_1
call p
action_2
```

```
halt
                    // Код p
action_3
return
```

На рис. 8.4 показана целевая программа для данного трехадресного кода. В ней использована псевдокоманда ACTION, представляющая последовательность машинных команд для выполнения инструкции action, которая, в свою очередь, представляет трехадресный код, не представляющий в настоящий момент для нас никакого интереса. Код процедуры *c* произвольным образом начинается с адреса 100, а процедуры *p* — с адреса 200. Считаем также, что каждая команда ACTION занимает 20 байт. Затем предположим, что память, необходимая для записи активации для этих процедур, статически выделяется начиная с ячеек соответственно 300 и 364.

```
                              // Код c
100:  ACTION1                 // Код action1
120:  ST 364, #140            // Сохранение адреса возврата 140 в ячейке 364
132:  BR 200                  // Вызов p
140:  ACTION2
160:  HALT                    // Возврат в операционную систему
      ...
                              // Код p
200:  ACTION3
220:  BR *364                 // Возврат к адресу, сохраненному в ячейке 364
      ...
                              // 300–363 хранит запись активации c
300:                          // Адрес возврата
304:                          // Локальные данные c
      ...
                              // 364–451 хранит запись активации p
364:                          // Адрес возврата
368:                          // Локальные данные p
```

Рис. 8.4. Целевой код статического распределения

Команды, начинающиеся с адреса 100, реализуют инструкции

```
action1; call p; action2; halt
```

первой процедуры *c*. Таким образом, выполнение программы начинается с команды $\texttt{ACTION}_1$ по адресу 100. Команда ST по адресу 120 сохраняет адрес возврата 140 в поле состояния машины, которое представляет собой первое слово

записи активации p. Команда BR по адресу 132 передает управление первой команде в целевом коде вызываемой процедуры p.

После $ACTION_3$ выполняется команда перехода по адресу 220. Поскольку в ячейке по адресу 364 вызывающей последовательностью записан адрес 140, при выполнении команды BR по адресу 220 операнд *364 представляет собой этот адрес. Таким образом, по завершении процедуры p управление возвращается по адресу 140 и продолжается выполнение процедуры c. □

8.3.2 Выделение памяти в стеке

Статическое распределение может стать выделением памяти в стеке при использовании в записи активации относительных адресов. Однако в случае стека положение записи активации процедуры во время компиляции неизвестно. Это положение обычно хранится в регистре, так что обратиться к словам в записи активации можно с использованием смещения относительно значения в этом регистре. Для этой цели вполне подходит индексированный режим адресации.

Относительные адреса в записи активации могут быть, как говорилось в главе 7, смещениями относительно любой известной позиции в записи активации. Для удобства мы будем использовать положительные смещения, храня в регистре SP указатель на начало записи активации на вершине стека. При вызове процедуры вызывающая процедура увеличивает значение SP и передает управление вызываемой процедуре. После того, как управление возвращается в вызывающую процедуру, она уменьшает значение SP, тем самым освобождая память, выделенную для записи активации вызываемой процедуры.

Код первой процедуры инициализирует стек, устанавливая SP на начало области стека в памяти:

```
LD   SP, #stackStart           // Инициализация стека
Код первой процедуры
HALT
```

Последовательность вызова процедуры увеличивает значение SP, сохраняет адрес возврата и передает управление вызываемой процедуре:

```
ADD  SP, SP, #caller.recordSize   // Увеличение указателя стека
ST   0(SP), #here + 16            // Сохранение адреса возврата
BR   callee.codeArea              // Передача управления вызываемой
                                  // процедуре
```

Операнд *#caller.recordSize* представляет размер записи активации, так что команда ADD делает регистр SP указывающим на следующую запись активации. Операнд

#here + 16 в команде ST представляет собой адрес команды, следующей за командой BR; он сохраняется по адресу, на который указывает SP.

Последовательность возврата состоит из двух частей. Вызываемая процедура передает управление по адресу возврата командой

```
BR    *0(SP)                        // Возврат в вызывающую процедуру
```

Причина использования операнда *0(SP) в команде BR в том, что нам требуется два уровня косвенности: 0(SP) — адрес первого слова в записи активации, а *0(SP) — хранящийся там адрес возврата.

Вторая часть последовательности возврата находится в вызывающей процедуре и заключается в уменьшении SP и восстановлении его предыдущего значения, так что после выполнения вычитания SP указывает на начало записи активации вызывающей процедуры:

```
SUB   SP, SP, #caller.recordSize    // Уменьшение указателя стека
```

В главе 7 более подробно рассмотрены как вызывающие последовательности, так и разделение задач между вызывающей и вызываемой процедурами.

Пример 8.4. Программа на рис. 8.5 представляет собой абстрагированную программу быстрой сортировки из предыдущей главы. Процедура q рекурсивная, так что в один и тот же момент может существовать несколько активаций q.

```
                 // Код m
action1
call q
action2
halt
                 // Код p
action3
return
                 // Код q
action4
call p
action5
call q
action6
call q
return
```

Рис. 8.5. Код к примеру 8.4

Предположим, что размеры записей активации для процедур m, p и q равны соответственно *msize*, *psize* и *qsize*. Первое слово в каждой записи активации

хранит адрес возврата. Произвольным образом мы полагаем, что коды этих процедур начинаются соответственно в адресах 100, 200 и 300 и что стек начинается с адреса 600. Целевая программа показана на рис. 8.6.

```
                                 // Код m
100:  LD SP, #600                // Инициализация стека
108:  ACTION1                    // Код action1
128:  ADD SP, SP, #msize         // Начало вызывающей последовательности
136:  ST 0(SP), #152             // Внесение адреса возврата в стек
144:  BR 300                     // Вызов q
152:  SUB SP, SP, #msize         // Восстановление SP
160:  ACTION2
180:  HALT
      ...
                                 // Код p
200:  ACTION3
220:  BR *0(SP)                  // Возврат
      ...
                                 // Код q
300:  ACTION4                    // Содержит условный переход по адресу 456
320:  ADD SP, SP, #qsize
328:  ST 0(SP), #344             // Внесение адреса возврата в стек
336:  BR 200                     // Вызов p
344:  SUB SP, SP, #qsize
352:  ACTION5
372:  ADD SP, SP, #qsize
380:  ST 0(SP), #396             // Внесение адреса возврата в стек
388:  BR 300                     // Вызов q
396:  SUB SP, SP, #qsize
404:  ACTION6
424:  ADD SP, SP, #qsize
432:  ST 0(SP), #440             // Внесение адреса возврата в стек
440:  BR 300                     // Вызов q
448:  SUB SP, SP, #qsize
456:  BR *0(SP)                  // Возврат
      ...
600:                             // Начало стека
```

Рис. 8.6. Целевой код для выделения памяти в стеке

Предполагается, что ACTION_4 содержит условный переход к адресу 456 последовательности возврата из q; в противном случае рекурсивная процедура q будет вечно вызывать саму себя.

Пусть *msize*, *psize* и *qsize* равны соответственно 20, 40 и 60. Первая команда по адресу 100 инициализирует SP значением 600, начальным адресом стека. Перед передачей управления от m к q в регистре SP хранится значение 620, так как *msize* равно 20. Далее, когда q вызывает p, команда по адресу 320 увеличивает значение SP до 690, места, где начинается запись активации p. После возврата управления процедуре q значение SP снова уменьшается до 620. Если из последующих двух рекурсивных вызовов q тут же выполняется возврат, то максимальное значение SP в процессе работы равно 680. Заметим, однако, что последняя использованная ячейка стека — 739, поскольку запись активации для q начинается с адреса 680 и занимает 60 байт. □

8.3.3 Адреса имен времени выполнения

Стратегия выделения памяти и схема размещения локальных данных в записи активации процедуры определяет, каким образом выполняется обращение к памяти, предназначенной для имен. В главе 6 считается, что имя в трехадресной команде в действительности является указателем на запись в таблице символов для данного имени. Такой подход обладает важным преимуществом; он делает компилятор более переносимым, поскольку не требуется изменение начальной стадии компилятора даже при переносе на другую машину с иной организацией времени выполнения. С другой стороны, генерация конкретной последовательности шагов обращения при генерации промежуточного кода может существенно помочь в случае оптимизирующего компилятора, поскольку обеспечивает компилятор детальной информацией, которой нет в простых трехадресных командах.

В любом случае в конечном счете имена должны быть заменены кодом для доступа к ячейкам памяти. Давайте рассмотрим простую трехадресную инструкцию присваивания x = 0. Предположим, что после обработки объявления в процедуре запись для x в таблице символов содержит относительный адрес 12. Рассмотрим случай, когда x находится в статически выделенной области памяти, начинающейся по адресу *static*. Тогда фактический адрес x времени выполнения равен *static* + 12. Хотя в конечном счете компилятор может определить значение *static* + 12 во время компиляции, позиция статической области может не быть известна, когда генерируется промежуточный код для доступа к имени. В таком случае имеет смысл генерировать трехадресный код для "вычисления" *static* + 12, отдавая себе отчет в том, что это вычисление будет выполнено на стадии генерации целевого кода или, возможно, загрузчиком перед запуском программы. Присваивание x = 0 транслируется в

```
static[12] = 0
```

Если статическая область начинается с адреса 100, то целевой код для этой инструкции имеет вид

```
LD 112, #0
```

8.3.4 Упражнения к разделу 8.3

Упражнение 8.3.1. Сгенерируйте код для следующих трехадресных инструкций в предположении выделения памяти в стеке, причем регистр SP указывает на вершину стека:

```
call p
call q
return
call r
return
return
```

Упражнение 8.3.2. Сгенерируйте код для следующих трехадресных инструкций в предположении выделения памяти в стеке, причем регистр SP указывает на вершину стека.

а) `x = 1`

б) `x = a`

в) `x = a + 1`

г) `x = a + b`

д) Две инструкции:

```
x = b * c
y = a + x
```

Упражнение 8.3.3. Сгенерируйте код для следующих трехадресных инструкций в предположении выделения памяти в стеке, а также в предположении, что a и b — массивы элементов, представляющих собой четырехбайтовые значения.

а) Последовательность из четырех инструкций:

```
x = a[i]
y = b[j]
a[i] = y
b[j] = x
```

б) Последовательность из трех инструкций:

```
x = a[i]
y = b[i]
z = x * y
```

в) Последовательность из трех инструкций:

```
x = a[i]
y = b[x]
a[i] = y
```

8.4 Базовые блоки и графы потоков

В этом разделе вводится представление промежуточного кода в виде графа, полезное при рассмотрении генерации кода (даже если алгоритм генерации кода не строит граф явным образом). Генерация кода выигрывает от знания контекста. Как мы увидим в разделе 8.8, знание того, какие значения определены и используются, помогает наилучшим образом выполнить распределение регистров; из раздела 8.9 вы узнаете, что просмотр последовательности трехадресных инструкций позволяет наилучшим образом решить задачу выбора команд.

Представление строится следующим образом.

1. Промежуточный код разделяется на *базовые блоки* (basic blocks), представляющие собой максимальные последовательности следующих друг за другом трехадресных команд, обладающие приведенными ниже свойствами:

   а) поток управления может входить в базовый блок только через первую команду блока, т.е. переходы в средину блока отсутствуют;

   б) управление покидает блок без останова или ветвления, за исключением, возможно, в последней команде блока.

2. Базовые блоки становятся узлами *графа потока* (flow graph), ребра которого указывают порядок следования блоков.

Начиная с главы 9 мы будем рассматривать трансформации графов потоков, которые преобразуют исходный промежуточный код в "оптимизированный" промежуточный код, позволяющий генерировать более качественный целевой код. "Оптимизированный" промежуточный код превращается в машинный код с помощью методов генерации кода из этой главы.

Влияние прерываний

Представление о том, что управление, достигнув начала базового блока, обязательно дойдет до его конца, требует пояснений. Имеется множество причин для прерываний, явно не отражаемых в коде, которые могут заставить управление покинуть блок, возможно, навсегда. Например, команда наподобие `x=y/z` кажется не влияющей на поток управления, но если `z` равно 0, то она может привести к завершению программы.

Мы не будем беспокоиться о таких возможностях. Причина этого в следующем. Цель построения базовых блоков — оптимизация кода. В общем случае при генерации исключения либо оно будет обработано и управление вернется к команде, вызвавшей его, как если бы никакого исключения не было, либо программа завершится с кодом ошибки. В последнем случае оптимизация кода не имеет значения, поскольку в любом случае программа не даст требуемого результата.

8.4.1 Базовые блоки

Наша первая задача состоит в разбиении последовательности трехадресных команд на базовые блоки. Мы начинаем новый базовый блок с первой команды и добавляем команды до тех пор, пока не встретим условный или безусловный переход или метку у следующей команды. При отсутствии переходов и меток управление последовательно проходит от одной команды к другой. Эта идея формализована в следующем алгоритме.

Алгоритм 8.5. Разбиение трехадресных команд на базовые блоки

Вход: последовательность трехадресных команд.

Выход: список базовых блоков для данной последовательности, в которой каждая команда принадлежит ровно одному базовому блоку.

Метод: сначала мы определяем команды промежуточного кода, являющиеся *лидерами* (leader), т.е. первыми командами в некоторых базовых блоках. Команда сразу после окончания промежуточной программы в лидеры не включается. Вот правила поиска лидеров.

1. Первая трехадресная команда промежуточного кода является лидером.

2. Любая команда, являющаяся целевой для условного или безусловного перехода, является лидером.

3. Любая команда, следующая непосредственно за условным или безусловным переходом, является лидером.

Затем базовый блок каждого лидера определяется как содержащий самого лидера и все команды до (но не включая) следующего лидера или до конца промежуточной программы. □

Пример 8.6. Промежуточный код на рис. 8.7 превращает матрицу 10×10 в единичную. Хотя происхождение этого кода не важно, он может быть результатом трансляции псевдокода на рис. 8.8. При генерации промежуточного кода предполагается, что элементы массива являются числами с плавающей точкой и имеют размер 8 байт, и что матрица хранится построчно.

```
 1)  i = 1
 2)  j = 1
 3)  t1 = 10 * i
 4)  t2 = t1 + j
 5)  t3 = 8 * t2
 6)  t4 = t3 - 88
 7)  a[t4] = 0.0
 8)  j = j + 1
 9)  if j <= 10 goto (3)
10)  i = i + 1
11)  if i <= 10 goto (2)
12)  i = 1
13)  t5 = i - 1
14)  t6 = 88 * t5
15)  a[t6] = 1.0
16)  i = i + 1
17)  if i <= 10 goto (13)
```

Рис. 8.7. Промежуточный код для установки матрицы 10×10 равной единичной

for i от 1 до 10 **do**
    for j от 1 до 10 **do**
        $a[i, j] = 0.0$;
for i от 1 до 10 **do**
    $a[i, i] = 1.0$;

Рис. 8.8. Исходный код для промежуточного представления на рис. 8.7

Команда 1 является лидером в соответствии с правилом 1 алгоритма 8.5. Чтобы найти других лидеров, мы сначала находим команды переходов. В данном примере имеются три такие команды — 9, 11 и 17. В соответствии с правилом 2 целевые команды этих переходов являются лидерами; это команды 3, 2 и 13 соответственно.

В соответствии с правилом 3 каждая команда, следующая за переходом, является лидером; это команды 10 и 12. Заметим, что в этом коде нет команд, следующих за командой 17, но если бы такая команда была, то команда 18 также была бы лидером.

В результате мы заключаем, что лидерами являются команды 1, 2, 3, 10, 12 и 13. Базовые блоки каждого лидера содержат все команды от него самого до команды перед следующим лидером включительно. Таким образом, базовый блок лидера 1 состоит только из одной команды 1, блок лидера 2 — из одной команды 2. Блок лидера 3 состоит из команд 3–9 включительно. Командами блока лидера 10 являются команды 10 и 11. Блок лидера 12 состоит из единственной команды 12, а в блок лидера 13 входят команды 13–17. □

8.4.2 Информация о дальнейшем использовании

Для генерации хорошего кода необходима информация о том, когда значение переменной будет использовано в следующий раз. Если значение переменной, в настоящий момент находящееся в регистре, в дальнейшем использоваться не будет, то этот регистр можно назначить другой переменной.

Использование имени в трехадресной инструкции определяется следующим образом. Предположим, что трехадресная команда i присваивает значение переменной x. Если команда j имеет x в качестве операнда и управление может перейти от i к j по пути, на котором нет промежуточных присваиваний x, то мы говорим, что команда j *использует* значение x, вычисленное в команде i. Мы также говорим, что x "*живая*", или "*активная*" (live) в команде i.

Для каждой трехадресной инструкции $x = y + z$ мы хотим определить следующие использования x, y и z. Пока что мы не будем рассматривать использования вне базового блока, содержащего эту трехадресную инструкцию.

Наш алгоритм для определения информации о живучести и последующем использовании выполняет обратный проход по каждому базовому блоку. Информация сохраняется в таблице символов. Можно легко сканировать поток трехадресных инструкций для поиска окончаний базовых блоков, как в алгоритме 8.5. Поскольку процедуры могут иметь произвольные побочные эффекты, для удобства мы считаем, что вызов каждой процедуры начинает новый базовый блок.

Алгоритм 8.7. Определение информации о живучести и последующем использовании для каждой инструкции базового блока

ВХОД: базовый блок трехадресных инструкций B. Считаем, что изначально таблица символов указывает все не временные переменные в блоке B, которые остаются живыми на выходе из блока.

ВЫХОД: в каждой инструкции $i: \quad x = y + z$ в блоке B к значению i добавляется информация о живучести и последующем использовании x, y и z.

Метод: начинаем с последней инструкции B и сканируем блок к началу. В каждой инструкции $i: \quad x = y + z$ в блоке B мы делаем следующее.

1. Добавляем к инструкции i информацию о живучести и последующем использовании x, y и z, найденную в таблице символов.

2. Устанавливаем в таблице символов x как "не живая" и "не используемая в дальнейшем".

3. Устанавливаем в таблице символов y и z как "живые", а в качестве последующего их использования указываем i.

Здесь символ $+$ использован просто как оператор. В трехадресной инструкции i вида $x = +y$ или $x = y$ все шаги будут те же, что и выше, просто в них игнорируется z. Заметим, что порядок шагов (2) и (3) не может быть изменен, поскольку x может выступать в роли y или z. □

8.4.3 Графы потоков

После того как программа разбита на базовые блоки, поток управления между ними представляется в виде графа. Узлами графа потока являются базовые блоки. Ребро из блока B в блок C идет тогда и только тогда, когда первая команда блока C может следовать непосредственно за последней командой блока B. Имеются две ситуации, когда могут существовать такие ребра.

- Существует условный или безусловный переход от конца блока B к началу блока C.

- C следует непосредственно за B в исходном порядке трехадресных команд, а блок B не заканчивается безусловным переходом.

Мы говорим, что B — *предшественник* C, а C — *преемник* B.

Зачастую к графу добавляются два узла, именуемые *входом* и *выходом* и не соответствующие выполнимым промежуточным командам. Существует ребро от входа к первому выполнимому узлу графа, т.е. к базовому блоку, который начинается с первой команды промежуточного кода. Если последняя команда программы не является безусловным переходом, то выходу предшествует блок, содержащий эту последнюю команду программы. В противном случае таковым предшествующим блоком является любой базовый блок, имеющий переход к коду, не являющемуся частью программы.

Пример 8.8. Множество базовых блоков, построенное в примере 8.6, дает граф потока, показанный на рис. 8.9. Вход указывает на блок B_1, поскольку именно блок B_1 содержит первую команду программы. Единственным преемником блока

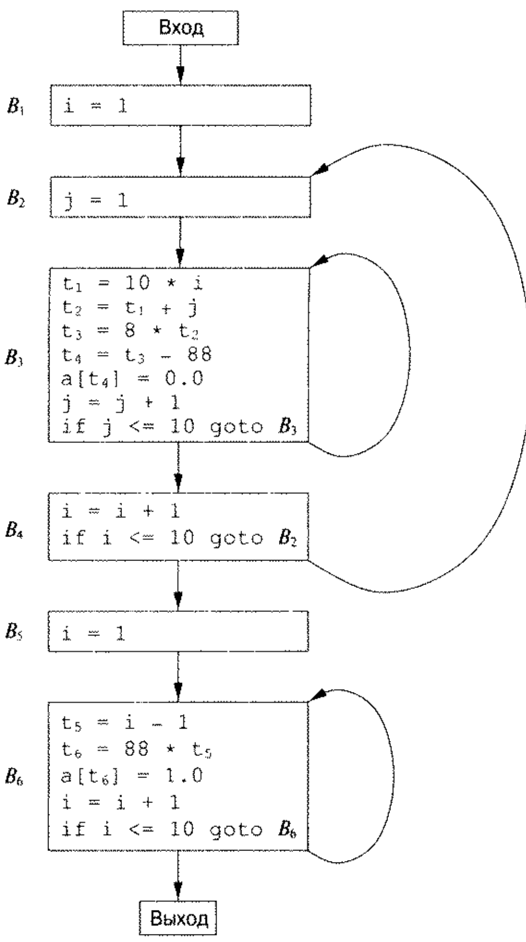


Рис. 8.9. Граф потока для рис. 8.7

B_1 является блок B_2, поскольку B_1 не заканчивается безусловным переходом, а лидер B_2 следует непосредственно за концом B_1.

У блока B_3 два преемника. Одним из них является сам блок B_3, поскольку лидер этого блока — команда 3 — является целевой командой условного перехода 9 в конце блока B_3. Вторым преемником блока B_3 является блок B_4, поскольку управление из той же команды условного перехода в блоке B_3 может быть передано лидеру блока B_4.

На выход из графа потока указывает только базовый блок B_6, поскольку единственный путь к коду, следующему за программой, для которой мы строим граф потока, следует через условный переход, которым завершается базовый блок B_6.□

8.4.4 Представление графов потоков

Обратите внимание, как на рис. 8.9 в графе потока переходы к номерам команд или меткам заменены переходами к базовым блокам. Вспомните, что любой условный или безусловный переход выполняется к лидеру некоторого базового блока, и это именно те блоки, которые указаны в командах переходов. Причина такого изменения заключается в том, что обычно после построения графа потока в команды разных базовых блоков вносятся существенные изменения. Если оставить переходы к командам, то придется вносить изменения в целевые адреса переходов всякий раз при изменении даже одной из команд.

Графы потоков, будучи обычными графами, могут быть представлены любыми структурами данных, подходящими для представления графов. Содержимое узлов (базовые блоки) требует собственного представления. Содержимое узлов можно представить при помощи указателя на лидера блока в массиве трехадресных команд вместе с количеством команд или указателем на последнюю команду блока. Однако, поскольку внесение изменений может приводить к частому изменению количества команд, пожалуй, более эффективным способом будет создание связанного списка команд для каждого базового блока.

8.4.5 Циклы

Конструкции языков программирования наподобие циклов while, do-while и for, естественным образом приводят к циклам в программах. Поскольку почти каждая программа затрачивает основную часть времени работы на выполнение циклов, особенно важным становится вопрос генерации эффективного высококачественного кода для циклов. Многие преобразования кода зависят от идентификации "циклов" в графах потоков. Мы говорим, что множество узлов L в графе потока является *циклом*, если выполняются следующие условия.

1. В L существует узел, именуемый *входом в цикл*, обладающий тем свойством, что никакие другие узлы не имеют предшественников вне L, т.е. любой путь из входа всего графа потока в любой узел в L проходит через вход в цикл, не совпадающий со входом в весь граф потока.

2. Каждый узел в L имеет непустой полностью содержащийся в L путь ко входу в L.

Пример 8.9. Граф потока на рис. 8.9 имеет три цикла:

1. состоящий из единственного блока B_3;

2. состоящий из единственного блока B_6;

3. $\{B_2, B_3, B_4\}$.

Первые два цикла представляют собой отдельные узлы с петлями, т.е. ребрами, выходящими из узла и входящими в тот же узел. Например, B_3 образует цикл с B_3 в качестве входа в цикл. Вспомним, что второе требование заключается в наличии непустого пути из B_3 в себя же. Таким образом, отдельный узел, например, B_2, у которого нет петли $B_2 \to B_2$, циклом не является, поскольку не существует непустого пути из B_2 в B_2, полностью лежащего в $\{B_2\}$.

Третий цикл $L = \{B_2, B_3, B_4\}$ имеет вход в цикл B_2. Обратите внимание, что среди указанных трех узлов только у узла B_2 есть предшественник B_1, не входящий в множество L. Далее, каждый из этих трех узлов имеет непустой путь к B_2, полностью лежащий в L. Например, B_2 имеет путь $B_2 \to B_3 \to B_4 \to B_2$ с заданным свойством. □

8.4.6 Упражнения к разделу 8.4

Упражнение 8.4.1. На рис. 8.10 приведена простая программа умножения матриц.

```
for (i=0; i<n; i++)
    for (j=0; j<n; j++)
        c[i][j] = 0.0;
for (i=0; i<n; i++)
    for (j=0; j<n; j++)
        for (k=0; k<n; k++)
            c[i][j] = c[i][j] + a[i][k]*b[k][j];
```

Рис. 8.10. Алгоритм умножения матриц

а) Транслируйте эту программу в трехадресные инструкции, подобные рассмотренным в этом разделе. Считается, что элементы матрицы являются 8-байтными числами и что матрица хранится построчно.

б) Постройте граф потока для вашего кода из части *а* упражнения.

в) Укажите циклы в графе потока из части *б* упражнения.

Упражнение 8.4.2. На рис. 8.11 приведен код для подсчета количества простых чисел от 2 до n с использованием алгоритма решета Эратосфена для достаточно большого массива a. Иначе говоря, в конечном счете $a[i]$ равно TRUE тогда и только тогда, когда не существует простых чисел, не превышающих $\sqrt{i}$, являющихся делителем i. Массив a инициализируется значениями TRUE, после чего элемент массива $a[j]$ устанавливается равным FALSE, если мы находим делитель j.

а) Транслируйте эту программу в трехадресные инструкции, подобные рассмотренным в этом разделе. Считается, что размер целого числа — 4 байт.

```
for (i=2; i<=n; i++)
    a[i] = TRUE;
count = 0;
s = sqrt(n);
for (i=2; i<=s; i++)
    if (a[i]) {              /* i --- простое число */
        count++;
        for (j=2*i; j<=n; j = j+i)
            a[j] = FALSE;    /* Простое число не может */
                             /* быть кратно i */
    }
```

Рис. 8.11. Код решета Эратосфена

б) Постройте граф потока для вашего кода из части *а* упражнения.

в) Укажите циклы в графе потока из части *б* упражнения.

8.5 Оптимизация базовых блоков

Зачастую можно существенно снизить время работы кода, просто выполнив *локальную* оптимизацию внутри каждого блока. Большее можно получить при помощи *глобальной* оптимизации, которая рассматривает потоки информации между базовыми блоками программы. Эта сложная оптимизация, включающая множество различных методов, будет рассматриваться начиная с главы 9.

8.5.1 Представление базовых блоков с использованием ориентированных ациклических графов

Многие важные методы локальной оптимизации начинаются с преобразования базового блока в ориентированный ациклический граф. В разделе 6.1.1 ориентированные ациклические графы использовались для представления отдельных выражений. Естественным образом эта идея распространяется и на набор выражений, образующих базовый блок. Ориентированный ациклический граф для базовых блоков строится следующим образом.

1. В ориентированном ациклическом графе имеются узлы для всех начальных значений переменных, появляющихся в базовом блоке.

2. Для каждой инструкции s в базовом блоке имеется связанный с ней узел N ориентированного ациклического графа. Дочерними узлами N являются

узлы, соответствующие инструкциям, являющимся последними до s определениями операндов, использующихся в s.

3. Узел N помечается оператором из s; кроме того, к узлу присоединяется список переменных, последнее определение которых находится в блоке.

4. Ряд узлов помечается как *выходные* (output nodes). Это узлы, переменные в которых *живые при выходе* из базового блока, т.е. их значения могут использоваться позже, в другом блоке графа потока. Вычисления этих "живых переменных" являются предметом анализа глобального потока, рассматривающегося в разделе 9.2.5.

Представление базового блока при помощи ориентированного ациклического графа позволяет нам выполнить ряд улучшающих преобразований кода, представленного блоком.

а) Можно устранить *локальные общие подвыражения* (local common subexpression), т.е. команды, которые вычисляют уже вычисленные значения.

б) Можно устранить *неиспользуемый*, или *мертвый*, *код* (dead code), т.е. команды, вычисляющие никогда не использующиеся значения.

в) Можно переупорядочить инструкции, не зависящие друг от друга; такое переупорядочение может снизить время хранения временного значения в регистре.

г) Можно применить алгебраические законы для переупорядочения операндов трехадресных команд, что иногда приводит к упрощению вычислений.

8.5.2 Поиск локальных общих подвыражений

Общие подвыражения могут быть обнаружены путем выяснения при добавлении нового узла M, нет ли уже существующего узла N с теми же дочерними узлами в том же порядке и с теми же операторами. Если такой узел есть, то N вычисляет то же самое значение, что и M, и может быть использован вместо него. Эта технология уже рассматривалась в разделе 6.1.1 в качестве метода "номера значения" для поиска общих подвыражений.

Пример 8.10. Ориентированный ациклический граф для блока

```
a = b + c
b = a - d
c = b + c
d = a - d
```

приведен на рис. 8.12. При построении узла для третьей инструкции c = b + c мы обнаруживаем, что использование b в b+c обращается к узлу с меткой − на рис. 8.12, поскольку это последнее определение b. Таким образом, нас не должны волновать значения, вычисляемые в первой и третьей инструкциях.

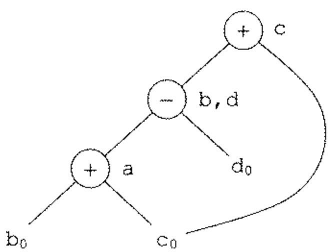


Рис. 8.12. Ориентированный ациклический граф для базового блока из примера 8.10

Однако узел, соответствующий четвертой инструкции, d = a − d, имеет оператор − и узлы a и d_0 в качестве дочерних. Поскольку оператор и дочерние узлы те же, что и у узла, соответствующего второй инструкции, мы не создаем этот узел, а просто добавляем d к узлу с меткой −. □

Может показаться, что, поскольку в ориентированном ациклическом графе на рис. 8.12 имеется только три узла, не являющихся листьями, базовый блок на рис. 8.10 можно заменить блоком только из трех инструкций. В действительности если b на выходе из блока не является живой, то нам не надо вычислять значение этой переменной и можно использовать d для получения значения, представленного узлом с меткой − на рис. 8.12. В этом случае блок превращается в

```
a = b + c
d = a - d
c = d + c
```

Однако если на выходе из базового блока живыми являются и b, и d, то четвертой инструкцией должно быть копирование значения из одной переменной в другую[1].

Пример 8.11. При поиске общих подвыражений мы ищем выражения, которые гарантированно вычисляют одно и то же значение, независимо от того, как это

[1]В общем случае следует быть осторожным при выборе имен переменных во время реконструкции кода из ориентированного ациклического графа. Если переменная x определена дважды или если выполняется одно присваивание и при этом используется и ее начальное значение x_0, то следует убедиться, что значение x не изменяется до тех пор, пока не будут выполнены все использования узла, хранящего предыдущее значение x.

значение вычисляется. Таким образом, метод использования ориентированного ациклического графа не в состоянии обнаружить, что первая и четвертая инструкции в последовательности

```
a = b + c
b = b - d
c = c + d
e = b + c
```

одинаковы и равны $b_0 + c_0$. Иначе говоря, несмотря на то, что и b, и c между первой и последней инструкциями изменяются, их сумма остается той же, поскольку $b + c = (b - d) + (c + d)$. Ориентированный ациклический граф для этой последовательности показан на рис. 8.13, но в нем не видно ни одного общего подвыражения. Однако применение к ориентированному ациклическому графу алгебраических тождеств, рассматривающееся в разделе 8.5.4, может выявить такую эквивалентность. □

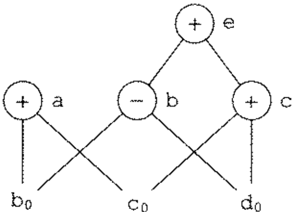


Рис. 8.13. Ориентированный ациклический граф для базового блока из примера 8.11

8.5.3 Устранение неиспользуемого кода

Операции над ориентированным ациклическим графом, соответствующие устранению неиспользуемого кода, могут быть выполнены следующим образом. Мы удаляем из ориентированного ациклического графа любой корень (узел без предков), с которым не связаны живые переменные. Повторное применение этого преобразования удалит из ориентированного ациклического графа все узлы, соответствующие неиспользуемому коду.

Пример 8.12. Если на рис. 8.13 a и b — живые, а c и e — нет, то можно тут же удалить корень с меткой e. После этого узел с меткой c становится корнем и также может быть удален. Корни с метками a и b остаются, поскольку с каждым из них связана живая переменная. □

8.5.4 Применение алгебраических тождеств

Алгебраические тождества представляют еще один важный класс оптимизации базовых блоков. Например, для устранения вычислений из базового блока могут применяться такие тождества, как

$$x + 0 = 0 + x = x \qquad x - 0 = x$$
$$x \times 1 = 1 \times x = x \qquad x/1 = x$$

Еще один класс алгебраических оптимизаций включает локальное *снижение стоимости вычислений*, т.е. заменяет более дорогостоящие с вычислительной точки зрения операторы более дешевыми, как, например,

Дорогой		Дешевый
x^2	=	$x \times x$
$2 \times x$	=	$x + x$
$x/2$	=	$x \times 0.5$

Третий класс оптимизаций — *свертывание констант* (constant folding), заключающееся в вычислении констант в процессе компиляции и замене константных выражений их значениями[2]. Так, выражение $2 * 3.14$ можно заменить значением 6.28. На практике многие константные выражения возникают из-за частого использования в программах символьных констант.

Процесс построения ориентированного ациклического графа может помочь нам в применении этих и других более общих алгебраических преобразований, таких как коммутативность и ассоциативность. Предположим, например, что в спецификации языка программирования указано, что операция умножения коммутативна, т.е. что $x * y = y * x$. Перед тем как создать новый узел с меткой *, левым дочерним узлом M и правым дочерним узлом N, мы проверяем, не существует ли уже такой узел. Однако в силу коммутативности умножения мы должны проверить также наличие узла с оператором *, левым дочерним узлом N и правым дочерним узлом M.

Операторы отношений, такие как $<$ и $=$, иногда генерируют неожиданные общие подвыражения. Например, условие $x > y$ может быть также проверено путем вычитания аргументов и проверки кода условия, установленного при вычитании[3].

[2]Арифметические выражения должны вычисляться во время компиляции так же, как они бы вычислялись во время выполнения программы. К. Томпсон (K. Thompson) предложил элегантное решение этой проблемы, заключающееся в том, что константное выражение компилируется, его целевой код выполняется и заменяется полученным результатом. Таким образом, компилятору не надо включать в себя интерпретатор.

[3]Однако вычитание может привести к переполнению или опустошению, чего не может произойти при применении команды сравнения.

Таким образом, для $x - y$ и $x > y$ может быть сгенерирован единственный узел ориентированного ациклического графа.

Ассоциативные законы также могут применяться для выявления общих подвыражений. Например, если в исходном коде имеются присваивания

```
a = b + c;
e = c + d + b;
```

то может быть сгенерирован следующий промежуточный код:

```
a = b + c
t = c + d
e = t + b
```

Если t вне этого блока не используется, то, воспользовавшись ассоциативностью и коммутативностью сложения, данную последовательность можно заменить следующей:

```
a = b + c
e = a + d
```

Разработчик компилятора должен внимательно изучить спецификацию языка программирования, чтобы знать, какие изменения вычислений допустимы, поскольку (из-за возможных переполнений и опустошений) компьютерная арифметика не всегда подчиняется алгебраическим тождествам математики. Например, стандарт языка программирования Fortran указывает, что компилятор может вычислять любое математически эквивалентное выражение, обеспечивая неизменность примененных программистом скобок. Так, компилятор может вычислить $x * y - x * z$ как $x * (y - z)$, но не может вычислить $a + (b - c)$ как $(a + b) - c$. Таким образом, компилятор языка программирования Fortran должен отслеживать наличие скобок в исходном тексте программы, чтобы оптимизировать ее в соответствии с определением языка программирования.

8.5.5 Представление обращений к массивам

На первый взгляд может показаться, что команды с индексами массивов ничем не отличаются от других команд и могут рассматриваться как любые другие операторы. Рассмотрим, например, последовательность трехадресных инструкций

```
x = a[i]
a[j] = y
z = a[i]
```

Если рассматривать `a[i]` как операцию, включающую `a` и i, подобную $a + i$, то может показаться, что два использования `a[i]` представляют собой общие подвыражения. В этом случае можно попытаться "оптимизировать" код, заменяя третью команду `z = a[i]` более простым присваиванием `z = x`. Но поскольку `j` может оказаться равным `i`, средняя команда может на самом деле изменить значение `a[i]`, так что такая замена некорректна.

Правильный способ представления обращения к элементам массива в ориентированном ациклическом графе выглядит следующим образом.

1. Присваивание элемента массива наподобие `x = a[i]` представляется при помощи создания узла с оператором =[] и двумя дочерними узлами, представляющими начальное значение массива, в данном случае a_0, и индекс `i`. Переменная `x` становится меткой нового узла.

2. Присваивание элементу массива наподобие `a[i] = y` представляется новым узлом с оператором []= и тремя дочерними узлами, представляющими a_0, `j` и `y`. Никакая переменная не выступает в качестве метки данного узла. Главное же в том, что создание этого узла *аннулирует* все созданные к этому моменту узлы, значения которых зависят от a_0. Аннулированный узел не может получать никакие метки, т.е. он не может стать общим подвыражением.

Пример 8.13. Ориентированный ациклический граф для базового блока

```
x = a[i]
a[j] = y
z = a[i]
```

показан на рис. 8.14. Сначала создается узел N для `x`, но, когда создается узел с меткой []=, N аннулируется. Таким образом, когда создается узел для z, он не может быть отождествлен с N, и в результате должен быть создан новый узел с теми же операндами a_0 и i_0. □

Пример 8.14. Иногда узел должен быть аннулирован, несмотря на то что ни один из его дочерних узлов не имеет в качестве связанной с ним переменной массив наподобие a_0 в примере 8.13. Узел может быть аннулирован, если имеет потомка, являющегося массивом, даже если ни один из его дочерних узлов не является узлом массива. Рассмотрим, например, трехадресный код

```
b = 12 + a
x = b[i]
b[j] = y
```

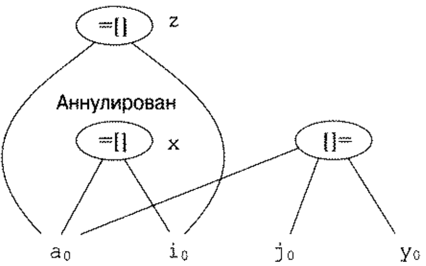


Рис. 8.14. Ориентированный ациклический граф для последовательности присваиваний с участием массива

Здесь для повышения эффективности b определена как позиция в массиве a. Например, если элементы a имеют размер 4 байт, то b представляет четвертый элемент a. Если i и j имеют одно и то же значение, то b[i] и b[j] представляют собой один и тот же элемент. Следовательно, третья команда, b[j] = y, аннулирует узел, с которым связана переменная x. Однако, как видно из рис. 8.15, как аннулированный, так и аннулирующий узлы не имеют дочернего узла a_0 — он является "внучатым" узлом для обоих указанных. □

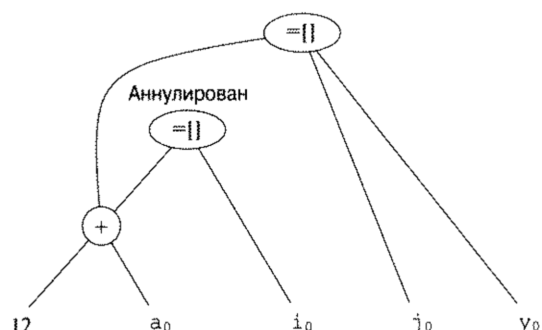


Рис. 8.15. Узел, аннулирующий использование массива, не обязан иметь его в качестве дочернего узла

8.5.6 Присваивание указателей и вызовы процедур

Нам неизвестно, куда указывают p и q при косвенном присваивании с использованием указателя, как в следующих инструкциях:

```
x = *p
*q = y
```

В сущности, x = *p представляет собой использование любой переменной, а *q = y – возможное присваивание любой переменной. Следовательно, оператор =* должен принимать в качества аргументов все узлы, связанные в настоящий момент с идентификаторами, что оказывает существенное влияние на устранение неиспользуемого кода. И, что еще более важно, оператор *= аннулирует все прочие узлы в ориентированном ациклическом графе.

Глобальный анализ указателей иногда может ограничить множество переменных, на которые в данном месте кода может ссылаться некоторый указатель. Даже локальный анализ может ограничить область видимости указателя. Например, в случае последовательности

```
p = &x
*p = y
```

мы знаем, что только x, и никакая иная переменная, может получить значение y, так что нет необходимости в аннулировании узлов, кроме тех, с которыми связана переменная x.

Вызовы процедур ведут себя во многом подобно присваиваниям с использованием указателей. При отсутствии глобальной информации о потоках данных мы должны считать, что процедура использует и изменяет любые данные, к которым обращается. Таким образом, если процедура P находится в области видимости переменной x, то вызов P как использует узел с присоединенной переменной x, так и аннулирует его.

8.5.7 Сборка базового блока из ориентированного ациклического графа

После выполнения всех возможных оптимизаций при построении ориентированного ациклического графа (или при работе с построенным ориентированным ациклическим графом) можно восстановить трехадресный код базового блока, для которого был построен этот ориентированный ациклический граф. Для каждого узла с одной или несколькими связанными переменными мы строим трехадресную инструкцию, которая вычисляет значение одной из этих переменных. Предпочтительно вычисление результата в переменную, которая будет живой при выходе из блока. Однако если глобальная информация о живых переменных отсутствует, то надо считать, что любая переменная программы (но не временные переменные, генерируемые компилятором для обработки выражений) является живой при выходе из базового блока.

Если у узла несколько присоединенных живых переменных, то следует добавить команды копирования, которые присвоят корректное значение каждой из этих переменных. Иногда глобальная оптимизация позволяет избежать копирования, если одна из двух переменных может быть использована вместо другой.

Пример 8.15. Вернемся к ориентированному ациклическому графу на рис. 8.12. В обсуждении после примера 8.10 мы решили, что если переменная b не является живой, то для реконструкции базового блока достаточно трех инструкций:

```
a = b + c
d = a - d
c = d + c
```

Третья команда, `c = d + c`, должна использовать в качестве операнда d, а не b, поскольку оптимизированный базовый блок никогда не вычисляет b.

Если при выходе из базового блока живыми являются и b, и d или если мы не знаем, какие именно переменные живые при выходе из базового блока, то мы должны вычислять как d, так и b. Это можно сделать при помощи последовательности

```
a = b + c
d = a - d
b = d
c = d + c
```

Этот базовый блок все равно остается более эффективным, чем исходный. Хотя количество инструкций в нем остается тем же, что и в исходном блоке, вычитание заменено копированием, которое на большинстве машин является более выгодной операцией. Далее, при проведении глобального анализа может выясниться, что использования b вне блока можно будет заменить использованиями d. В таком случае позже можно будет вернуться к данному базовому блоку и убрать из него копирование `b = d`. Интуитивно это копирование можно устранить, если везде, где используется значение переменной b, переменная d все еще хранит то же самое значение. Это может быть так, а может и не быть в зависимости от того, когда и как программа заново вычисляет значение d. □

При восстановлении базового блока из ориентированного ациклического графа мы должны не только беспокоиться о том, какие переменные используются для хранения значений узлов ориентированного ациклического графа, но и о порядке, в котором мы перечисляем команды, вычисляющие значения различных узлов. Следует помнить следующие правила.

1. Порядок команд должен отражать порядок узлов в ориентированном ациклическом графе, т.е. мы не можем вычислить значение узла до тех пор, пока не вычислим значения всех его дочерних узлов.

2. Присваивания массиву должны следовать за всеми присваиваниями этому массиву или за вычислениями с использованием его элементов, которые в исходном базовом блоке выполняются ранее.

3. Вычисление элементов массива должно следовать за всеми более ранними (в соответствии с исходным базовым блоком) присваиваниями элементам того же массива. Единственная разрешенная перестановка заключается в том, что два вычисления с использованием элементов одного и того же массива могут быть выполнены в любом порядке, если между ними нет ни одного присваивания элементам этого массива.

4. Любое использование переменной должно следовать за всеми более ранними (в соответствии с исходным базовым блоком) вызовами процедур или косвенными присваиваниями с использованием указателей.

5. Любой вызов процедуры или косвенное присваивание с использованием указателя должно следовать за всеми более ранними (в соответствии с исходным базовым блоком) вычислениями переменных.

Иными словами, при переупорядочении кода ни одна инструкция не должна пересекаться с вызовом процедуры или присваиванием с использованием указателя, а использования одного и того же массива могут пересекаться только в том случае, если оба они являются обращениями к массиву, но не присваиваниями значений его элементам.

8.5.8 Упражнения к разделу 8.5

Упражнение 8.5.1. Постройте ориентированный ациклический граф для базового блока

```
d = b * c
e = a + b
b = b * c
a = e - d
```

Упражнение 8.5.2. Упростите трехадресный код из упражнения 8.5.1, в предположении, что

а) a — единственная живая переменная на выходе из блока;

б) на выходе из блока живыми являются переменные a, b и c.

Упражнение 8.5.3. Постройте ориентированный ациклический граф для кода блока B_6 на рис. 8.9. Не забудьте включить в него сравнение $i \leqslant 10$.

Упражнение 8.5.4. Постройте ориентированный ациклический граф для кода блока B_3 на рис. 8.9.

Упражнение 8.5.5. Доработайте алгоритм 8.7 так, чтобы он мог обрабатывать трехадресные инструкции вида

а) `a[i] = b`

б) `a = b[i]`

в) `a = *b`

г) `*a = b`

Упражнение 8.5.6. Постройте ориентированный ациклический граф для базового блока

```
a[i] = b
*p = c
d = a[j]
e = *p
*p = a[i]
```

в предположении, что

а) `p` может указывать куда угодно;

б) `p` может указывать только на `b` или `d`.

! **Упражнение 8.5.7.** Если выражение с указателем или массивом, такое как `a[i]` или `*p`, сначала получает значение, а затем используется, причем без возможности изменения в промежутке между этими событиями, то этой ситуацией можно воспользоваться для упрощения ориентированного ациклического графа. Например, в коде упражнения 8.5.6, поскольку присваивания `p` между второй и четвертой инструкциями не происходит, инструкцию `e = *p` можно заменить инструкцией `e = c`, независимо от того, куда именно указывает `p`[4]. Пересмотрите алгоритм построения ориентированного ациклического графа с тем, чтобы он использовал преимущества такой ситуации, и примените этот алгоритм к коду из упражнения 8.5.6.

Упражнение 8.5.8. Предположим, что базовый блок образован инструкциями на языке программирования C

```
x = a + b + c + d + e + f;
y = a + c + e;
```

а) Приведите трехадресные инструкции (по одному сложению в инструкции) для этого блока.

б) Воспользуйтесь коммутативным и ассоциативным законами для того, чтобы этот блок использовал минимально возможное количество команд, в предположении, что и `x`, и `y` живые на выходе из блока.

---

[4]Лишь бы этот указатель не указывал на `d` (т.е. между второй и четвертой инструкциями не должна меняться переменная, на которую указывает `p`). — *Прим. ред.*

8.6 Простой генератор кода

В этом разделе мы рассмотрим алгоритм генерации кода для отдельного базового блока. Он поочередно рассматривает трехадресные команды и отслеживает, какие значения находятся в регистрах, так что позволяет избежать излишних загрузок и сохранений.

Одним из основных вопросов в процессе генерации кода является принятие решения о том, как наилучшим образом использовать регистры. Существует четыре основных применения регистров.

- В большинстве архитектур некоторые или все операнды должны находиться в регистрах для выполнения операции.
- Регистры представляют собой хорошие временные переменные — места для хранения результатов подвыражений при вычислениях больших выражений или, в общем случае, для размещения переменных, использующихся в пределах только одного базового блока.
- Регистры используются для хранения (*глобальных*) значений, которые вычисляются в одном базовом блоке и используются в других базовых блоках, например, индекса цикла, который увеличивается на каждой итерации цикла и неоднократно используется в пределах цикла.
- Регистры зачастую используются для помощи в управлении памятью времени выполнения, например для управления стеком времени выполнения, включая поддержку указателя стека и, возможно, элементов на вершине стека.

Это конкурирующие использования регистров, поскольку их количество весьма ограничено.

Алгоритм в данном разделе предполагает наличие некоторого множества регистров, доступных для хранения использующихся в блоке значений. Обычно это множество не включает в себя все имеющиеся в машине регистры, поскольку некоторые из них зарезервированы для глобальных переменных и управления стеком. Мы считаем, что базовые блоки уже преобразованы в идеальные последовательности трехадресных команд при помощи таких трансформаций, как объединение общих подпоследовательностей и др. Мы также предполагаем, что для каждого оператора имеется ровно одна машинная команда, которая получает необходимые операнды в регистрах и выполняет операцию, причем ее результат также находится в регистре. Машинные команды имеют вид

- LD *reg, mem*
- ST *mem, reg*
- *OP reg, reg, reg*

8.6.1 Дескрипторы регистров и адресов

Наш алгоритм генерации кода поочередно рассматривает каждую трехадресную команду и определяет, какие загрузки необходимо выполнить, чтобы требуемые операнды находились в регистрах. После генерации загрузок генерируется сама операция, а затем при необходимости сохранения результата в памяти — команды сохранения.

Для принятия необходимых решений нужна структура данных, которая говорит нам, какие переменные программы находятся в регистрах и в каких именно регистрах. Нам также надо знать, содержит ли в настоящий момент ячейка памяти для данной переменной корректное значение, — дело в том, что может оказаться так, что новое вычисленное значение переменной находится в регистре и еще не сохранено в памяти. Требуемая структура данных имеет следующие дескрипторы.

1. *Дескриптор регистра* отслеживает для всех доступных регистров имена переменных, значения которых хранятся в настоящее время в регистрах. Поскольку мы будем использовать только регистры, доступные для локального использования в базовом блоке, мы полагаем, что изначально все дескрипторы регистров пустые. В процессе генерации кода каждый регистр будет хранить значение нуля или некоторого количества имен.

2. *Дескриптор адреса* отслеживает для каждой переменной программы местоположение, где можно найти текущее значение этой переменной. Местоположение может быть регистром, адресом памяти, положением в стеке или некоторым множеством из этих данных. Эта информация может храниться в записи таблицы символов для имени переменной.

8.6.2 Алгоритм генерации кода

Важнейшей частью алгоритма является функция *getReg* (I), которая выбирает регистры для каждой ячейки памяти, связанной с трехадресной командой I. Функция *getReg* имеет доступ к дескрипторам регистров и адресов для всех переменных базового блока и может также иметь доступ к некоторой полезной информации о потоках данных, такой как переменные, остающиеся живыми при выходе из базового блока. Мы рассмотрим эту функцию после того, как представим основной алгоритм. Пока мы не знаем общее количество регистров, доступных для локальных данных, принадлежащих базовому блоку, мы полагаем, что регистров достаточно для того, чтобы после освобождения всех доступных регистров путем сохранения их значений в памяти можно было выполнить любую трехадресную операцию.

В трехадресной команде наподобие `x = y + z` мы рассматриваем `+` как обобщенный оператор, а `ADD` — как эквивалентную машинную команду. Таким

образом, мы не используем коммутативность сложения. Поэтому, когда мы реализуем операцию, значение y должно находиться во втором регистре в команде ADD и никогда в третьем. Возможное усовершенствование алгоритма заключается в генерации кода для x = y + z и x = z + y для коммутативного оператора + и выборе лучшей последовательности.

Машинные команды для операций

Для трехадресной команды, такой как $x = y + z$, выполняются следующие действия.

1. Используем функцию *getReg* $(x = y + z)$ для выбора регистров для x, y и z. Назовем их R_x, R_y и R_z.

2. Если y не находится в R_y (согласно дескриптору регистра для R_y), то генерируем команду LD R_y, y', где y' — местоположение y в памяти (в соответствии с дескриптором адреса для y).

3. Аналогично, если z находится не в регистре R_z, то генерируем команду LD R_z, z', где z' — местоположение z.

4. Генерируем команду ADD R_x, R_y, R_z.

Машинные команды для инструкций копирования

Имеется важный частный случай — трехадресная инструкция копирования вида $x = y$. Мы считаем, что функция *getReg* всегда выбирает одни и те же регистры как для x, так и для y. Если y еще не находится в регистре R_y, то генерируем машинную команду LD R_y, y. Если же y уже находится в регистре R_y, то не делаем ничего. Все, что необходимо сделать, — это обновить дескриптор регистра для R_y так, чтобы он включал x как одно из находящихся в нем значений.

Завершение базового блока

Как уже говорилось, переменные, используемые блоком, могут находиться только в регистрах. Если это переменная, временно используемая только внутри блока, то все в порядке, — когда блок завершается, мы можем просто забыть об этой переменной и считать регистр пустым. Однако если при выходе из блока переменная живая или нам неизвестно, какие переменные живые при выходе из блока, то следует считать, что значение этой переменной потребуется позже. В этом случае для каждой переменной x, дескриптор адреса которой не гласит, что ее значение хранится в ячейке памяти, выделенной для x, требуется генерация команды ST x, R, где R — регистр, в котором находится значение x в конце базового блока.

Управление дескрипторами регистров и адресов

При генерации алгоритмом команд загрузки, сохранения и прочих машинных команд необходимо обновление дескрипторов регистров и адресов. Вот правила такого обновления.

1. Для команды `LD` R, x:

   а) изменяем дескриптор регистра R так, чтобы он указывал, что в R хранится только переменная x;

   б) изменяем дескриптор адреса для x, добавляя регистр R как дополнительное местоположение.

2. Для команды `ST` x, R изменяем дескриптор адреса для x, чтобы он включал местоположение переменной в памяти.

3. Для операции, такой как `ADD` R_x, R_y, R_z, реализующей трехадресную команду $x = y + z$:

   а) изменяем дескриптор регистра для R_x так, чтобы он указывал, что в R_x хранится только переменная x;

   б) изменяем дескриптор адреса для x так, чтобы он указывал, что единственное местоположение этой переменной — регистр R_x (обратите внимание, что теперь ячейки памяти, выделенные для x, *не* входят в дескриптор адреса для x);

   в) удаляем R_x из дескрипторов адресов всех переменных, кроме x.

4. При обработке инструкции копирования $x = y$ после генерации при необходимости загрузки y в регистр R_y и после изменения дескрипторов, как того требует инструкция загрузки (правило 1),

   а) добавляем x в дескриптор регистра R_y;

   б) изменяем дескриптор адреса для x так, чтобы он указывал, что единственное местоположение этой переменной — регистр R_y.

Пример 8.16. Транслируем базовый блок, состоящий из трехадресных инструкций

```
t = a - b
u = a - c
v = t + u
a = d
d = v + u
```

Мы считаем, что t, u и v — временные переменные, локальные для данного блока, в то время как a, b, c и d — переменные, живые при выходе из блока. Поскольку мы еще не рассматривали работу функции *getReg*, будем просто считать, что у нас столько регистров, сколько нам нужно, но что, когда значение регистра более не нужно (например, в нем хранится временная переменная, все использования которой позади), такой регистр используется повторно.

Все сгенерированные машинные команды показаны на рис. 8.16. Там же показаны дескрипторы регистров и адресов до и после трансляции каждой трехадресной команды.

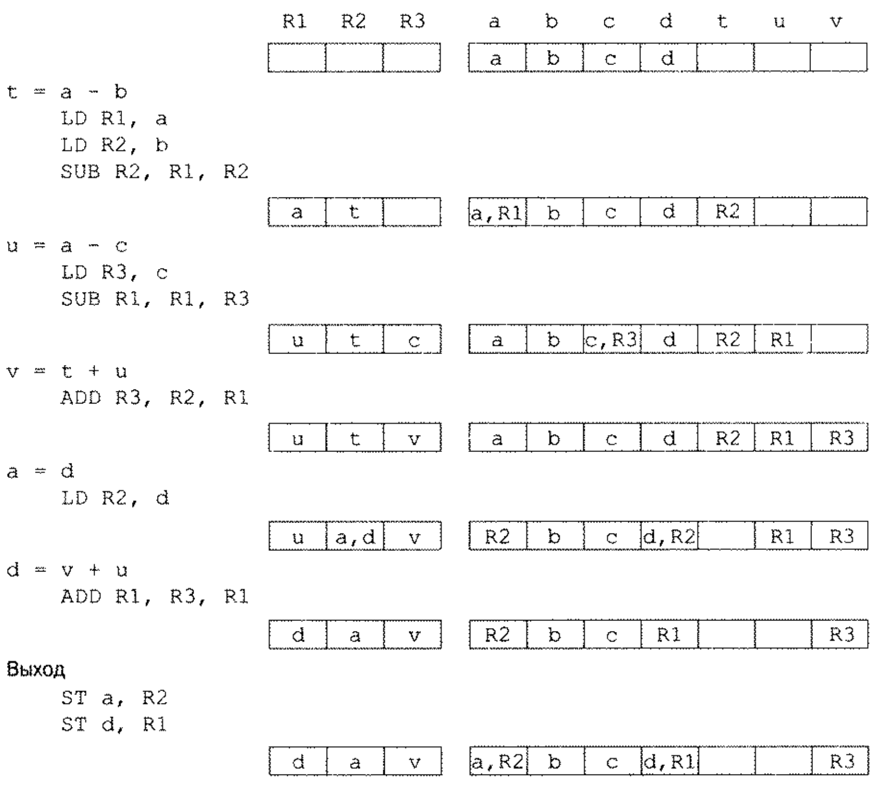


Рис. 8.16. Сгенерированные команды и изменения в дескрипторах регистров и адресов

Для первой трехадресной команды, t = a - b, необходимо сгенерировать три команды, поскольку изначально в регистрах не находятся никакие значения. Таким образом, a и b загружаются в регистры R1 и R2 и значение t вычисляется в регистр R2. Заметим, что мы используем регистр R2 для переменной t, поскольку значение b, находившееся ранее в R2, больше в блоке не используется. Поскольку переменная b живая на выходе из блока, то, если бы она не

находилась в своей ячейке памяти (на что указывает ее дескриптор адреса), нам потребовалось бы сначала сохранить значение регистра R2 в памяти, отведенной для переменной b. Такие решения принимаются функцией *getReg*.

Вторая команда, u = a - c, не требует загрузки a, поскольку данное значение уже находится в регистре R1. Далее, можно использовать регистр R1 для результата операции u, поскольку значение a, ранее располагавшееся в этом регистре, больше внутри блока не используется, и если a потребуется вне блока, то ее значение имеется в отведенном ей месте в памяти. Обратите внимание на изменения в дескрипторе адреса для a, отражающие тот факт, что a больше не находится в R1, а только в ячейках памяти, выделенных для переменной.

Третья команда, v = t + u, требует только одного сложения. Далее, для результата v можно использовать регистр R3, поскольку значение c, хранящееся в этом регистре, больше внутри блока не требуется, а актуальное значение c хранится в памяти, выделенной для данной переменной.

Команда копирования a = d требует загрузки d, поскольку это значение находится только в памяти. Мы указываем, что в регистре R2 теперь хранятся и d, и a. Добавление a в дескриптор регистра является результатом обработки инструкции копирования, но не результатом какой-либо машинной команды.

Пятая команда, d = v + u, использует два значения, находящиеся в регистрах. Поскольку u — временная переменная, значение которой больше не нужно, мы выбираем повторное использование регистра R1 для хранения нового значения переменной d. Обратите внимание, что теперь d хранится только в регистре R1, но не в выделенной для этой переменной памяти. То же самое можно сказать и об a, значение которой находится в регистре R2, но не в ячейке памяти для a. В результате нам требуется добавление к машинному коду базового блока, которое сохранит живые переменные a и d в их ячейках памяти. Это — две последние машинные команды базового блока. □

8.6.3 Разработка функции *getReg*

Наконец, рассмотрим, как реализовать функцию *getReg* (I) для трехадресной команды I. Имеется много вариантов, как это сделать, при наличии строжайшего требования, что выбор не должен привести к некорректному коду из-за потери одной или нескольких живых переменных. Начнем наше рассмотрение с операции, в качестве обобщенного примера которой выберем $x = y + z$. Сначала выбираем регистр для y и регистр для z. Оба выбора выполняются одинаково, так что рассмотрим выбор регистра R_y для y. Вот как это делается.

1. Если y в настоящее время хранится в регистре, то в качестве R_y выбирается регистр, который уже содержит y.

2. Если y не хранится в регистре, но имеется регистр, в настоящий момент являющийся пустым, он выбирается в качестве регистра R_y.

3. Наиболее сложный случай, когда y не находится в регистре и в настоящий момент пустых регистров нет. Нам в любом случае надо выбрать один из доступных регистров, так что мы должны обеспечить безопасность повторного использования регистра. Пусть R — регистр-кандидат, и предположим, что v — одна из переменных, о которых дескриптор регистра R говорит, что они хранятся в данном регистре. Мы должны убедиться, что значение v в действительности более не требуется, либо значение v можно получить из другого места. Вот возможные ситуации.

   а) Если дескриптор адреса для v гласит, что v располагается где-то вне R, то все в порядке, R можно использовать.

   б) Если v представляет собой x — переменную, вычисляемую командой I, и x не является одновременно одним из операндов команды I (в данном примере — z), то все в порядке, R можно использовать. В этом случае мы знаем, что данное значение x больше не будет использовано, так что его можно игнорировать.

   в) В противном случае, если v больше не используется (т.е. после команды I нет иных использований v, и если v — живая переменная на выходе из блока, то v вычисляется позже в блоке), то все в порядке, R можно использовать.

   г) Если же приведенные ситуации не осуществляются, то требуется сгенерировать команду сохранения $ST\ \ v, R$ для сохранения копии v в ее местоположении в памяти. Эта операция называется *сбросом* (spill).

   Поскольку R может одновременно хранить несколько переменных, приведенные выше шаги следует повторить для всех таких переменных v. Среди регистров выбирается один из тех, у которых количество одновременно хранимых переменных минимально.

Теперь рассмотрим выбор регистра R_x. Почти все варианты при этом те же, что и для y, так что упомянем только имеющиеся отличия.

1. Поскольку вычисляется новое значение переменной x, регистр, в котором хранится единственная переменная x, всегда годится в качестве R_x. Это утверждение справедливо даже тогда, когда x является одним из y или z, поскольку наша машинная команда допускает совпадение двух регистров в одной команде.

2. Если y не используется после команды I в смысле, описанном для переменной v в пункте 3*в*, и R_y хранит только одну переменную y, то R_y также может использоваться в роли R_x. То же самое относится и к z и R_z.

Последним рассмотрим частный случай, когда I представляет собой команду копирования $x = y$. Регистр R_y выбирается, как и ранее, после чего мы всегда выбираем $R_x = R_y$.

8.6.4 Упражнения к разделу 8.6

Упражнение 8.6.1. Для каждой из приведенных ниже инструкций присваивания на языке программирования C сгенерируйте трехадресный код в предположении, что все элементы массива — целые числа размером по 4 байт. В частях *г* и *д* считаем, что a, b и c — константы, указывающие положения первых (с индексом 0) элементов массивов с данными именами, как и во всех предыдущих примерах обращения к массивам в данной главе.

а) `x = a + b*c;`

б) `x = a/(b+c) - d*(e+f);`

в) `x = a[i] + 1;`

г) `a[i] = b[c[i]];`

д) `a[i][j] = b[i][k] + c[k][j];`

е) `*p++ = *q++;`

Упражнение 8.6.2. Повторите части *г* и *д* упражнения 8.6.1 в предположении, что массивы a, b и c доступны через указатели соответственно pa, pb и pc, указывающие на положения первых элементов массивов.

Упражнение 8.6.3. Преобразуйте трехадресный код из упражнения 8.6.1 в машинный код для модели машины из этого раздела. Вы можете использовать столько регистров, сколько вам надо.

Упражнение 8.6.4. Преобразуйте трехадресный код из упражнения 8.6.1 в машинный код с помощью простого алгоритма генерации кода из данного раздела в предположении доступности трех регистров. Приведите дескрипторы регистров и адресов после каждого шага алгоритма.

Упражнение 8.6.5. Повторите упражнение 8.6.4 при условии доступности только двух регистров.

8.7 Локальная оптимизация

В то время как большинство промышленных компиляторов дают хороший код путем тщательного выбора команд и распределения регистров, некоторые компиляторы используют иную стратегию: они генерируют простейший код, а затем повышают его качество, применяя "оптимизирующие" преобразования целевой программы. Термин "оптимизирующий" в данном случае несколько неверен, поскольку не гарантируется, что полученный в результате код будет более оптимален с той или иной точки зрения. Тем не менее многие простые преобразования могут существенно улучшить время работы или размер целевой программы.

Простым, но эффективным методом локального улучшения целевого кода является *локальная оптимизация* (peephole optimization[5]), которая выполняется путем перемещения по целевой программе окна ("peephole" — "глазка") и изучения команд в его пределах с дальнейшей заменой последовательностей команд более быстрыми или более короткими там, где это возможно. Локальная оптимизация зачастую применима непосредственно после генерации промежуточного кода для усовершенствования промежуточного представления.

Глазок представляет собой небольшое перемещающееся по программе окно. Код в этом окне не обязательно непрерывен, хотя некоторые реализации и выдвигают такое требование. Характерным для локальной оптимизации является то, что каждое улучшение кода может создать условия для дополнительных улучшений. Вообще говоря, для получения максимального эффекта необходимы повторяющиеся проходы по целевому коду. В этом разделе мы приведем следующие примеры преобразования программ, характеризующие локальную оптимизацию:

- устранение излишних инструкций;
- оптимизация потока управления;
- алгебраические упрощения;
- использование машинных идиом.

8.7.1 Устранение излишних загрузок и сохранений

Если в целевой программе нам встречается последовательность команд

```
LD R0, a
ST a, R0
```

то можно удалить команду сохранения, поскольку при ее выполнении первая команда гарантирует, что значение a уже загружено в регистр R0. Заметим, что если

[5]Дословно — "оптимизация смотрового отверстия". — *Прим. пер.*

бы команда сохранения имела метку, то мы не могли бы быть уверены, что первая команда всегда выполняется немедленно перед второй, так что удаление второй команды было бы ошибочным. Иными словами, чтобы такое преобразование было безопасным, обе команды должны находиться в одном базовом блоке.

Излишние загрузки и сохранения такого вида не генерируются при использовании простого алгоритма из предыдущего раздела. Однако простейший наивный алгоритм генерации кода наподобие алгоритма из раздела 8.1.3 может генерировать излишние последовательности, подобные рассмотренной.

8.7.2 Устранение недостижимого кода

Еще одной возможностью локальной оптимизации является удаление недостижимых команд. Команда без метки, следующая непосредственно за безусловным переходом, может быть удалена. Эта операция может быть повторена для удаления последовательности команд. Например, в целях отладки большая программа может использовать фрагменты кода, которые выполняются только тогда, когда переменная `debug` равна 1. В промежуточном представлении такой код может выглядеть следующим образом:

```
    if debug == 1 goto L1
    goto L2
L1: Вывод отладочной информации
L2:
```

Очевидной локальной оптимизацией является устранение переходов через переходы. Независимо от того, какое значение имеет переменная `debug`, приведенную последовательность можно заменить такой последовательностью:

```
    if debug != 1 goto L2
    Вывод отладочной информации
L2:
```

Если в начале программы переменная `debug` установлена равной 0, то распространение констант преобразует эту последовательность в

```
    if 0 != 1 goto L2
    Вывод отладочной информации
L2:
```

Теперь аргумент первой инструкции всегда истинен, так что вся инструкция может быть заменена безусловным переходом `goto L2`. Тогда все инструкции, выводящие отладочную информацию, являются недостижимыми и могут быть удалены.

8.7.3 Оптимизация потока управления

Алгоритмы генерации промежуточного кода часто приводят к коду, в котором имеются условные и безусловные переходы к командам-переходам. Такие излишние переходы могут быть устранены либо в промежуточном, либо в целевом коде путем локальной оптимизации следующих видов. Мы можем заменить последовательность переходов

```
    goto L1
    ...
L1: goto L2
```

последовательностью

```
    goto L2
    ...
L1: goto L2
```

Если после такой замены переходы к `L1` отсутствуют, команду `L1: goto L2` можно убрать из целевого кода, если ей предшествует безусловный переход. Аналогично последовательность

```
    if a < b goto L1
    ...
L1: goto L2
```

может быть заменена последовательностью

```
    if a < b goto L2
    ...
L1: goto L2
```

Наконец, предположим, что есть только один переход к `L1` и что `L1` предшествует безусловный переход. Тогда последовательность

```
    goto L1
    ...
L1: if a < b goto L2
L3:
```

может быть заменена последовательностью

```
    if a < b goto L2
    goto L3
    ...
L3:
```

Хотя число инструкций в этих двух последовательностях и одинаково, иногда безусловный переход во второй последовательности может оказаться пропущен, но в первой последовательности он выполняется всегда. Таким образом, код второй последовательности эффективнее кода первой с точки зрения времени выполнения.

8.7.4 Алгебраические упрощения и снижение стоимости

В разделе 8.5 мы рассматривали алгебраические тождества, которые могут использоваться для упрощения ориентированных ациклических графов. Эти алгебраические тождества могут использоваться и при локальной оптимизации для устранения трехадресных команд наподобие `x := x + 0` или `x := x * 1` в пределах окна.

Аналогично снижение стоимости может применяться локально, заменяя дорогие с точки зрения времени выполнения, используемых ресурсов и других характеристик команды целевой машины эквивалентными более дешевыми. Одни машинные команды значительно дешевле других и часто используются как частный случай более дорогих операторов. Например, x^2 всегда дешевле вычислить как $x * x$, чем путем вызова подпрограммы возведения в степень. Умножение или деление чисел с фиксированной точкой на степень двойки дешевле реализовать как сдвиг. Деление чисел с плавающей точкой на константу, реализованное как умножение на другую константу, также может оказаться более дешевым.

8.7.5 Использование машинных идиом

Целевая машина может иметь аппаратные команды, эффективно реализующие ряд специфических операций. Определение ситуаций, позволяющих использовать специальные машинные команды, может существенно сократить время выполнения кода. Например, ряд машин имеет режимы адресации с автоувеличением или автоуменьшением, при которых к операнду прибавляется (или вычитается из него) единица до или после использования его значения. Использование таких режимов существенно повышает качество кода для внесения в стек или снятия с него, выполняемых, например, при передаче параметров процедуре. Эти режимы могут также использоваться в коде для инструкций наподобие `x = x + 1`.

8.7.6 Упражнения к разделу 8.7

Упражнение 8.7.1. Разработайте алгоритм для устранения излишних команд в окне, перемещающемся по целевому коду.

Упражнение 8.7.2. Разработайте алгоритм для выполнения оптимизации потока управления в окне, перемещающемся по целевому коду.

Упражнение 8.7.3. Разработайте алгоритм, который будет выполнять простые алгебраические упрощения и снижение стоимости в окне, перемещающемся по целевому коду.

8.8 Распределение и назначение регистров

Команды, включающие в качестве операндов только регистры, выполняются быстрее, чем аналогичные команды с ячейками памяти в качестве операндов. На современных машинах эта разница в скорости может достигать порядка по величине. Следовательно, эффективное использование регистров — жизненно важная составляющая генерации хорошего целевого кода. В этом разделе представлены различные стратегии принятия решения о том, какие значения в программе должны находиться в регистрах (распределение регистров) и в каком регистре должно храниться каждое значение (назначение регистров).

Один из подходов к распределению и назначению регистров состоит в выделении конкретных регистров для определенных значений в целевой программе. Например, решение может состоять в назначении базовых адресов одной группе регистров; арифметических вычислений — второй; вершины стека времени выполнения — некоторому фиксированному регистру и т.д.

Преимуществом такого подхода является упрощение разработки генератора кода. Недостаток же состоит в том, что слишком строгое следование правилу ограничивает эффективность использования регистров; ряд регистров на некотором участке кода может оставаться неиспользованным, в то время как из-за нехватки других регистров будут производиться излишние обмены значениями между регистрами и памятью. Тем не менее в большинстве вычислительных сред имеет смысл зарезервировать несколько регистров в качестве базовых, указателя стека и других и позволить компилятору использовать остальные регистры по своему усмотрению.

8.8.1 Глобальное распределение регистров

Алгоритм генерации кода из раздела 8.6 использовал регистры для хранения значений во время выполнения отдельного базового блока. Однако все живые переменные в конце каждого блока сохранялись в памяти. Для того чтобы сэкономить на сохранениях в памяти и соответствующих им загрузках, мы можем назначить регистры часто используемым переменным и хранить их там при переходе границ между блоками (т.е. *глобально*). Поскольку программы тратят бо́льшую часть времени на выполнение внутренних циклов, естественным подходом к глобальному назначению регистров будет попытка хранить часто используемое значение в фиксированном регистре в процессе выполнения всего цикла. Предположим, что нам известна циклическая структура графа потока и мы знаем,

какие значения, вычисляемые в базовом блоке, используются вне его. В следующей главе вы познакомитесь с методами получения этой информации.

Одна из стратегий глобального распределения регистров состоит в назначении некоторого фиксированного числа регистров для хранения наиболее активно используемых значений в каждом внутреннем цикле. Эти значения могут отличаться от цикла к циклу. Нераспределенные после этого регистры могут использоваться для хранения локальных значений в блоке, как это делалось в разделе 8.6. Недостаток такого подхода в том, что фиксированного количества регистров не всегда хватает для глобального распределения регистров. Однако этот метод прост в реализации и был использован в Fortran H, оптимизирующем компиляторе Fortran для машин серии IBM-360 в конце 1960-х годов.

В ранних компиляторах C программист мог выполнять часть работы по распределению регистров, используя соответствующее явное объявление для хранения переменных в регистрах во время выполнения процедуры. Разумное использование таких объявлений ускоряло выполнение программ, но программист не должен заниматься распределением регистров до того, как выполнит профилирование программы и определит места в коде, которые могут выиграть от таких действий[6].

8.8.2 Счетчики использований

В этом разделе мы предполагаем, что экономия, получаемая при хранении переменной x в регистре в процессе выполнения цикла L, составляет единицу стоимости кода для каждого обращения к x, если эта переменная находится в регистре. Однако при использовании подхода из раздела 8.6 для генерации кода блока весьма вероятно, что вычисленное в блоке значение x останется в регистре, если используется далее в этом блоке. Таким образом, мы подсчитываем суммарную экономию от хранения x в регистре, прибавляя единицу за каждое использование x в цикле L, если такому использованию не предшествует присваивание x в том же блоке. Кроме того, мы сэкономим две единицы, если избежим сохранения x в конце блока. Таким образом, если x хранится в регистре, мы добавляем к суммарной экономии по две единицы для каждого блока L, в котором x получает значение и остается живым при выходе из блока.

Вместе с тем, если переменная x является живой при входе в заголовок цикла, мы должны загрузить ее в регистр перед входом в цикл L. Такая загрузка стоит две единицы. Кроме того, для каждого из выходных блоков B цикла L, в которых

[6]Объявление переменных в качестве регистровых в языке программирования C является всего лишь пожеланием программиста, но никак не директивой компилятору, который при генерации кода может пренебречь этими объявлениями (например, когда регистровыми объявлено больше переменных, чем имеется регистров у целевой машины). В современных компиляторах эти возможности оставлены в целях совместимости, но их использование рассматривается как крайне нежелательное вмешательство в работу компилятора. — *Прим. ред.*

x остается живым при входе в преемник блока B вне цикла L, мы должны сохранить значение x, что также стоит две единицы. Однако в предположении, что цикл выполняется много раз, этой потерей экономии можно пренебречь (она однократна при входе в цикл). Таким образом, приближенная формула для выгоды от выделения регистра для хранения переменной x в цикле L выглядит следующим образом:

$$\sum_{\text{Блоки } B \text{ в } L} (use\,(x, B) + 2 * live\,(x, B)) \tag{8.1}$$

Здесь $use\,(x, B)$ — число использований x в B до любого определения x, а $live(x, B)$ равно 1, если переменная x остается живой при выходе из блока B, в котором ей присваивается значение, и 0 — в противном случае. Отметим приближенность (8.1), поскольку не все блоки в цикле выполняются с одинаковой частотой. Кроме того, формула (8.1) основана на предположении, что цикл выполняется много раз. Для конкретных машин можно разработать формулу, аналогичную (8.1), хотя, возможно, и существенно отличающуюся от нее.

Пример 8.17. Рассмотрим базовые блоки во внутреннем цикле, изображенном на рис. 8.17, на котором опущены инструкции безусловных и условных переходов. Предположим, что регистры R0, R1 и R2 выделены для хранения значений в цикле. Переменные, живые при входе в блок и выходе из него, показаны на рисунке непосредственно над и под блоком. Здесь имеется несколько тонкостей, относящихся к живым переменным, о которых мы поговорим в следующей главе. Например, обратите внимание, что и e, и f живы при выходе из блока B_1, но при входе в блок B_2 жива только переменная e, а при входе в блок B_3 — толь-

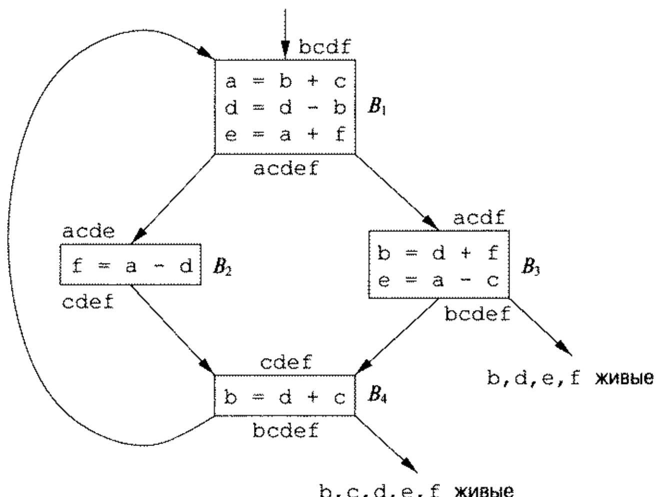


Рис. 8.17. Граф потока внутреннего цикла

ко f. В общем случае переменные, живые в конце блока, представляют собой объединение переменных, живых при входе в блоки-преемники.

При вычислении (8.1) для $x = a$ заметим, что переменная a жива при выходе из B_1 и получает в этом блоке свое значение, но не жива при выходе из блока B_2, B_3 или B_4. Следовательно, $\sum_{B \text{ в } L} use(a, B) = 2$. Следовательно, значение (8.1) для $x = a$ равно 4, а значит, выбирая a для хранения в одном из глобальных регистров, мы получаем экономию в 4 единицы. Значения (8.1) для b, c, d, e и f равны соответственно 5, 3, 6, 4 и 4, а потому для глобальных регистров R0, R1 и R2 мы можем выбрать переменные a, b и d. Использование R0 для хранения e или f вместо a дает тот же конечный результат. На рис. 8.18 показан ассемблерный код, сгенерированный по рис. 8.17 с применением стратегии из раздела 8.6 для генерации кода каждого базового блока. Мы не приводим здесь код для опущенных условных и безусловных переходов в конце каждого блока, а также не показываем сгенерированный код в качестве единого непрерывного потока, как это осуществляется на практике. □

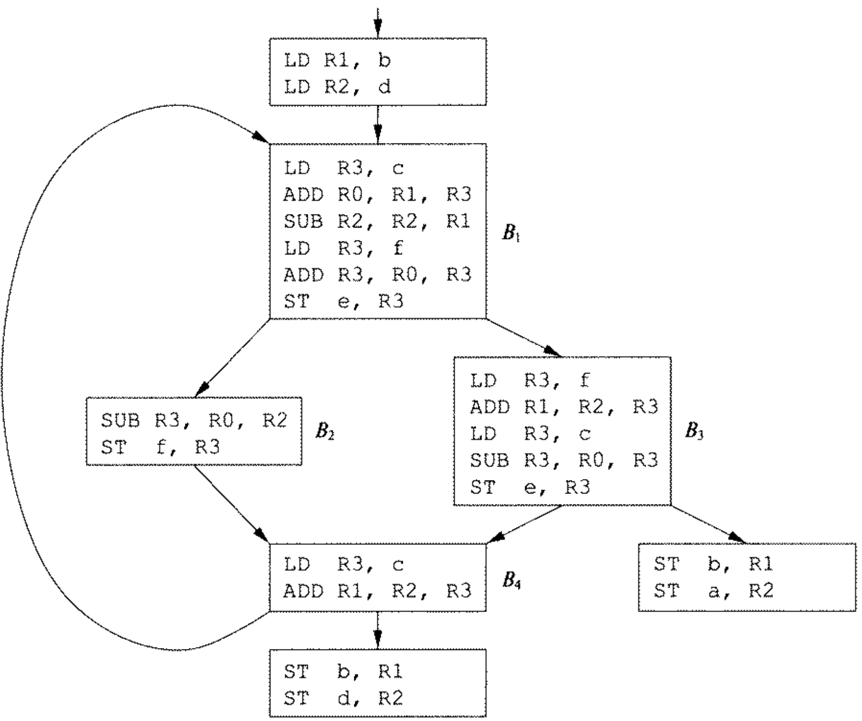


Рис. 8.18. Последовательность, использующая глобальное назначение регистров

8.8.3 Назначение регистров для внешних циклов

Назначив регистры и сгенерировав код для внутренних циклов, мы можем применить ту же идею и для бóльших циклов. Если внешний цикл L_1 содержит внутренний цикл L_2, имена, хранящиеся в регистрах в L_2, не обязаны храниться в регистрах в $L_1 - L_2$. Если мы храним x в регистре в L_2, но не в L_1, следует принять меры по загрузке x при входе в L_2 и сохранению при выходе из него. Мы оставляем читателю в качестве упражнения вывод критерия для выбора имен, хранимых в регистрах во внешнем цикле L, по уже выбранным для хранения в регистрах именам во всех вложенных в L циклах.

8.8.4 Распределение регистров путем раскраски графа

Когда для вычисления требуется регистр, а все доступные регистры уже используются, содержимое одного из них должно быть сохранено (*сброшено* — spilled) в ячейке памяти для его освобождения. Раскраска графа — это простой систематичный метод распределения регистров и управления их сбросом.

Этот метод требует двух проходов. При первом проходе команды целевой машины выбираются так, как будто у нас есть бесконечное множество символических регистров; по сути, имена, используемые в промежуточном представлении, становятся именами регистров, а трехадресные команды — командами машинного языка. Если обращение к переменным требует применения команд, использующих стековые указатели, указатели дисплея, базовые регистры или другие вспомогательные величины, мы предполагаем, что они также хранятся в регистрах, зарезервированных для этого. Обычно их использование непосредственно транслируется в режим обращения к адресам, использованным в машинных командах. Если требуется более сложное обращение к адресу памяти, то оно разбивается на несколько машинных команд и, возможно, придется создать временный символический регистр (или ряд таких регистров).

После выбора команд выполняется второй проход, в процессе которого и происходит назначение физических регистров символическим. Цель назначения состоит в минимизации стоимости сбросов.

При втором проходе для каждой процедуры строится *граф взаимодействия регистров* (register-interference graph), узлы которого представляют собой символические регистры, а дуги соединяют два узла в том случае, если в момент определения одного из регистров второй оказывается жив. Например, граф взаимодействия регистров для представленного на рис. 8.17 кода будет иметь узлы для имен a и d. Поскольку в блоке B_1 в момент определения d во второй инструкции a является живой переменной, эти узлы графа взаимодействия регистров будут соединены дугой.

Далее предпринимается попытка раскрасить граф k цветами, где k — количество доступных для назначения регистров. Граф называется *раскрашенным*, если

каждому его узлу назначен некоторый цвет таким образом, что никакие два соседних узла не имеют один и тот же цвет. Цвет в данном случае представляет регистр, а раскраска гарантирует, что никаких два символических регистра не будут мешать друг другу при назначении им одного и того же физического регистра.

Хотя в общем случае задача определения, может ли граф быть раскрашен с использованием k цветов, является NP-полной, обычно на практике для раскраски графов используется простая и быстрая эвристическая методика. Предположим, что узел n графа G имеет меньше, чем k соседей (узлов, соединенных дугами с n). Удалим n и его дуги из G, получив тем самым новый граф G'. k-раскраска G' может быть преобразована в k-раскраску G путем назначения узлу n цвета, не назначенного ни одному из его соседей.

Многократно удаляя таким образом узлы с менее чем n соседями из графа взаимодействия регистров, мы получим либо пустой граф (в этом случае мы выполняем k-раскраску исходного графа, раскрашивая узлы в порядке, обратном порядку удаления), либо граф, в котором каждый узел имеет k или более смежных узлов. В этом последнем случае k-раскраска графа невозможна, и требуется сброс узла путем добавления кода для сохранения и перезагрузки регистра. Чаитин (Chaitin) разработал ряд эвристических приемов для выбора сбрасываемого регистра. Общее правило состоит в том, чтобы избегать добавления кода для сброса во внутренний цикл.

8.8.5 Упражнения к разделу 8.8

Упражнение 8.8.1. Постройте граф взаимодействия регистров для программы, представленной на рис. 8.17.

Упражнение 8.8.2. Разработайте стратегию распределения регистров в предположении, что мы автоматически сохраняем все регистры в стеке перед каждым вызовом процедуры и восстанавливаем их после возвращения из нее.

8.9 Выбор команд путем переписывания дерева

Выбор команд может представлять собой комбинаторную задачу большого размера, в особенности на машинах с богатым набором режимов адресации, таких как машины CISC, или на машинах с командами специального назначения, например, для обработки сигналов. Даже если считать, что порядок вычислений задан и что регистры распределены с использованием отдельного механизма, выбор команд — задача выбора команд целевого языка для реализации операторов промежуточного представления — остается большой комбинаторной задачей.

В этом разделе мы рассмотрим выбор команд как задачу изменения дерева. Представление целевых команд с использованием дерева эффективно использовалось в генераторах генераторов кода, которые автоматически создавали фазу выбора команд генератора кода на основании высокоуровневой спецификации целевой машины. Для некоторых машин лучший код можно получить при использовании ориентированных ациклических графов, а не деревьев, но работа с ориентированными ациклическими графами более сложна, чем работа с деревьями.

8.9.1 Схемы трансляции деревьев

В этом разделе входом процесса генерации кода будет последовательность деревьев на семантическом уровне целевой машины. Эти деревья представляют собой результат вставки адресов времени выполнения в промежуточное представление, как описано в разделе 8.3. Кроме того, листья деревьев содержат информацию о типах хранилищ для их меток.

Пример 8.18. На рис. 8.19 показано дерево для инструкции присваивания a[i]=b+1, где массив a хранится в стеке времени выполнения, а переменная b — в глобальной ячейке памяти M_b. Адреса времени выполнения локальных переменных a и i заданы как константные смещения C_a и C_i относительно SP — регистра, содержащего указатель на начало текущей записи активации.

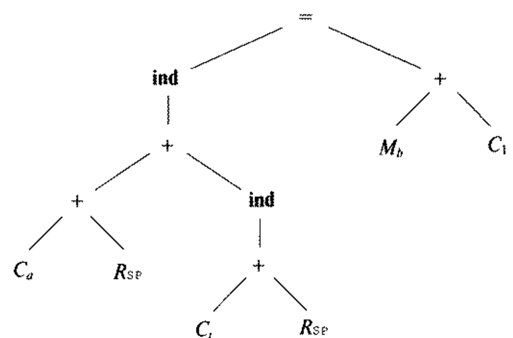


Рис. 8.19. Дерево промежуточного кода для a[i]=b+1

Присваивание элементу a[i] является косвенным присваиванием, в котором r-значение ячейки памяти a[i] устанавливается равным r-значению выражения b+1. Адреса массива a и переменной i определяются путем сложения значений констант C_a и C_i соответственно с содержимым регистра SP. Для простоты мы считаем, что все значения представляют собой однобайтные символы (некото-

рые наборы команд содержат специальные средства для умножений в процессе вычисления адресов на константы, такие как 2, 4 и 8).

В приведенном дереве оператор **ind** рассматривает свой аргумент как адрес памяти. В качестве левого дочернего узла оператора присваивания **ind** дает адрес памяти, по которому будет сохранено r-значение правой части оператора присваивания. Если аргумент операторов + или **ind** представляет собой ячейку памяти или регистр, то в качестве значения принимается содержимое этой ячейки памяти или регистра. Листья дерева представляют собой атрибуты (константа, регистр, ячейка памяти) с индексами, указывающими значения этих атрибутов. □

Целевой код генерируется в процессе свертки входного дерева в единый узел путем последовательного применения правил преобразования дерева. Каждое правило преобразования представляет собой инструкцию вида

$$replacement \leftarrow template \quad \{action\}$$

Здесь *replacement* — отдельный узел, *template* — дерево, а *action*, как и в случае схемы синтаксически управляемой трансляции, является фрагментом кода.

Множество правил преобразования дерева именуется *схемой трансляции дерева* (tree-translation scheme).

Каждое правило преобразования дерева представляет собой трансляцию части дерева, определяемой шаблоном. Трансляция состоит из (возможно, пустой) последовательности машинных команд, которые генерируются действием, связанным с шаблоном. Листья шаблона являются атрибутами с индексами, как и во входном дереве. Иногда к значениям индексов в шаблонах применяется ряд ограничений, определяемых в виде семантических предикатов, которым должен удовлетворять шаблон перед тем, как можно будет говорить о его соответствии. Например, предикат может требовать, чтобы значение константы находилось в определенном диапазоне.

Схема трансляции дерева представляет собой удобный способ представления фазы выбора инструкций генератора кода. В качестве примера рассмотрим правило для инструкции сложения одного регистра с другим:

Это правило применяется следующим образом. Если входное дерево содержит поддерево, соответствующее приведенному шаблону, т.е. поддерево, корень которого помечен оператором +, а его левый и правый потомки представляют собой величины, хранящиеся в регистрах i и j, то мы можем заменить это поддерево одним узлом с меткой R_i и сгенерировать команду ADD Ri, Ri, Rj. Назовем

такую замену *замощением* (tiling) поддерева. Одно поддерево может соответствовать нескольким шаблонам; далее мы вкратце опишем механизм выбора правила в случае возникновения конфликтов.

Пример 8.19. На рис. 8.20 показаны правила преобразования для некоторых команд нашей целевой машины. Эти правила будут использоваться в последующих примерах данного раздела. Первые два правила соответствуют инструкциям загрузки, следующие два — инструкциям сохранения, а остальные — индексированным загрузкам и сложениям. Обратите внимание, что правило 8 требует, чтобы значение константы было равно 1. Это условие может определяться семантическим предикатом. □

1)	$R_i \leftarrow C_a$	`{ LD Ri, #a }`
2)	$R_i \leftarrow M_x$	`{ LD Ri, x }`
3)	$M \leftarrow$ = M_x R_i	`{ ST x, Ri }`
4)	$M \leftarrow$ = **ind** R_j R_i	`{ ST *Ri, Rj }`
5)	$R_i \leftarrow$ **ind** + C_a R_j	`{ LD Ri, a(Rj) }`
6)	$R_i \leftarrow$ + R_i **ind** + C_a R_j	`{ ADD Ri, Ri, a(Rj) }`
7)	$R_i \leftarrow$ + R_i R_j	`{ ADD Ri, Ri, Rj }`
8)	$R_i \leftarrow$ + R_i C_1	`{ INC Ri }`

Рис. 8.20. Правила преобразования дерева для некоторых команд целевой машины

8.9.2 Генерация кода путем замощения входного дерева

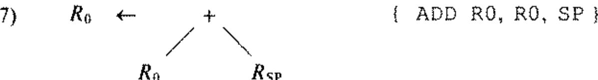
Схема трансляции дерева используется следующим образом. В данном входном дереве выполняется поиск поддеревьев, соответствующих имеющимся шаблонам правил преобразования. При соответствии шаблону поддерево заменяется узлом, определяемым правилом преобразования, и выполняются связанные с правилом действия. Если действие содержит последовательность машинных команд, они выводятся в выходной поток. Этот процесс повторяется до тех пор, пока в дереве имеются поддеревья, соответствующие шаблонам, и исходное дерево не свернуто в единственный узел. Последовательность машинных команд, генерируемая в процессе свертки входного дерева в один узел, образует выход схемы трансляции для данного входного дерева.

Процесс определения генератора кода аналогичен использованию схемы синтаксически управляемой трансляции для определения транслятора. Мы создаем схему трансляции дерева для описания набора команд целевой машины. На практике мы должны найти такую схему, которая приведет к генерации последовательности команд с минимальной стоимостью для каждого входного дерева. Существуют специальные инструментальные средства, облегчающие автоматическое построение генератора кода по схеме трансляции дерева.

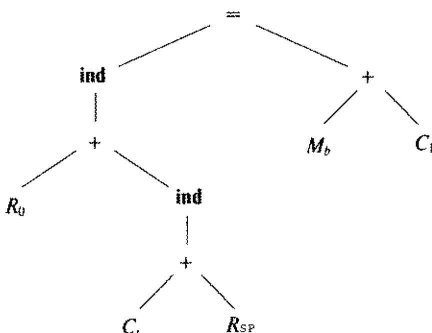
Пример 8.20. Воспользуемся схемой трансляции дерева на рис. 8.20 для генерации кода, соответствующего дереву, приведенному на рис. 8.19. Предположим, что правило 1 применяется к загрузке константы C_a в регистр `R0`:

1) $R_0 \leftarrow C_a$ `{ LD R0, #a }`

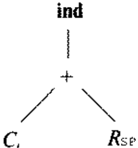
При этом метка крайнего слева листа C_a заменяется меткой R_0 и генерируется команда `LD R0, #a`. Теперь правило 7



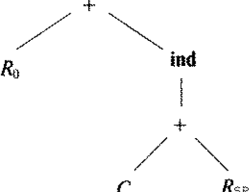
соответствует крайнему слева поддереву с меткой корня +. Используя это правило, мы заменим указанное поддерево одним узлом с меткой R_0 и сгенерируем команду `ADD R0,R0,SP`. В результате дерево будет выглядеть следующим образом:



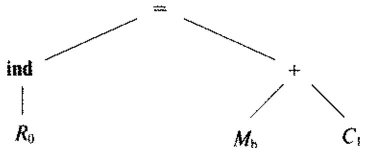
Теперь можно применить правило 5 для свертки поддерева



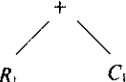
в единственный узел с меткой R_1. Однако можно воспользоваться правилом 6 для свертки большего дерева



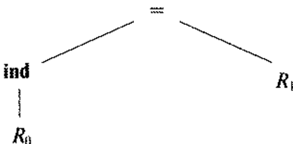
в единственный узел с меткой R_0 и генерации команды `ADD R0,R0,i(SP)`. Считая, что более эффективным является использование одной команды для вычисления большего дерева, мы выбираем последний вариант и получаем после свертки дерево



В правом поддереве к листу M_b мы применим правило 2, которое сгенерирует команду загрузки `b` в регистр, для определенности — регистр `R1`. В результате поддерево



соответствует шаблону правила 8 и при свертке дает команду INC R1. Теперь входное дерево свернуто в дерево



□

Оно соответствует поддереву из правила 4, применение которого дает в результате единственный узел и генерирует команду ST *R0,R1. Итак, в процессе свертки исходного дерева в единственный узел мы получили следующий код:

```
LD  R0,  #a
ADD R0,  R0, SP
ADD R0,  R0, i(SP)
LD  R1,  b
INC R1
ST  *R0, R1
```

Для реализации процесса свертки дерева из примера 8.18 нам надо решить пару вопросов, связанных с поиском соответствия деревьев шаблону.

- Каким образом выполняется поиск соответствия? Эффективность процесса генерации кода в процессе компиляции зависит от того, насколько эффективен применяющийся алгоритм.
- Что делать, если имеется более одного соответствия шаблону? Эффективность сгенерированного кода может зависеть от порядка, в котором выявлялось соответствие шаблонам, поскольку различные последовательности соответствий будут приводить к разным последовательностям машинных команд, одни из которых более эффективны, чем другие.

Если соответствия шаблонам не найдено, процесс генерации кода блокируется. Другим предельным случаем является бесконечное перезаписывание одного узла, что приводит к генерации бесконечной последовательности перемещения значения между регистрами или бесконечной последовательности загрузок и сохранений.

Чтобы избежать блокировки, мы полагаем, что любой оператор промежуточного кода может быть реализован при помощи одной или нескольких машинных команд. Далее, мы предполагаем, что имеется достаточно регистров для вычисления каждого узла. Тогда независимо от того, как именно выявляется соответствие шаблонам, остающееся дерево всегда может быть транслировано в команды целевой машины.

8.9.3 Поиск соответствий с использованием синтаксического анализа

Перед тем как рассматривать обобщенный поиск соответствий, рассмотрим специализированный подход, состоящий в использовании LR-синтаксического анализатора для выполнения поиска соответствия шаблонам. Входное дерево можно рассматривать как строку при использовании его префиксного представления. Так, префиксное представление дерева на рис. 8.19 выглядит следующим образом:

$$= \ \textbf{ind} \ + \ + \ C_a \ R_{\texttt{SP}} \ \textbf{ind} \ + \ C_i \ R_{\texttt{SP}} \ + \ M_b \ C_1$$

Схема трансляции дерева может быть преобразована в схему синтаксически управляемой трансляции путем замены правил преобразования дерева продукциями контекстно-свободной грамматики, в которых правые части являются префиксными представлениями шаблонов инструкций.

Пример 8.21. Синтаксически управляемая схема трансляции на рис. 8.21 основана на схеме трансляции дерева на рис. 8.20.

1)	$R_i \to \mathbf{c}_a$	`{ LD Ri, #a }`
2)	$R_i \to M_x$	`{ LD Ri, x }`
3)	$M \to = M_x\ R_i$	`{ ST x, Ri }`
4)	$M \to = \textbf{ind}\ R_i\ R_j$	`{ ST *Ri, Rj }`
5)	$R_i \to \textbf{ind} + \mathbf{c}_a\ R_j$	`{ LD Ri, a(Rj) }`
6)	$R_i \to +R_i\ \textbf{ind} + \mathbf{c}_a\ R_j$	`{ ADD Ri, Ri, a(Rj) }`
7)	$R_i \to +R_i\ R_j$	`{ ADD Ri, Ri, Rj }`
8)	$R_i \to +R_i\ \mathbf{c}_1$	`{ INC Ri }`
9)	$R \to \mathbf{sp}$	
10)	$M \to \mathbf{m}$	

Рис. 8.21. Синтаксически управляемая схема трансляции, построенная на основе рис. 8.20

Нетерминалами лежащей в основе такой схемы грамматики являются R и M. Терминал **m** представляет конкретную ячейку памяти, как, например, местоположение для глобальной переменной b в примере 8.18. Продукция $M \to \mathbf{m}$ в правиле 10 может рассматриваться как соответствие нетерминала M ячейке памяти **m** перед использованием одного из шаблонов, применяющих M. Аналогично вводятся терминал **sp** для регистра SP и продукция $R \to \mathbf{sp}$. Наконец, терминал **c** представляет константы.

При использовании этих терминалов строка для входного дерева на рис. 8.19 принимает вид

$$= \ \textbf{ind} \ + \ + \ \mathbf{c}_a \ \mathbf{sp} \ \textbf{ind} \ + \ \mathbf{c}_i \ \mathbf{sp} \ + \ \mathbf{m}_b \ \mathbf{c}_1$$

□

На основе продукций схемы трансляции мы строим LR-синтаксический анализатор с применением одного из методов построения LR-синтаксических анализаторов, представленных в главе 4. Целевой код генерируется путем вывода машинных инструкций, соответствующих каждой свертке.

Обычно грамматика генерации кода весьма неоднозначна и поэтому должны быть предприняты дополнительные меры по разрешению конфликтов в процессе создания синтаксического анализатора. При отсутствии информации о стоимости общее правило состоит в предпочтении бóльших сверток меньшим. Это означает, что в случае конфликта "свертка — свертка" выбирается более длинная свертка, а при конфликте "перенос — свертка" — перенос. Такой подход "максимального разжевывания" приводит к выполнению большего числа операций с помощью одной машинной инструкции.

Существует ряд преимуществ использования LR-синтаксического анализа для генерации кода. Во-первых, метод синтаксического анализа эффективен и хорошо изучен, а использование алгоритмов, описанных в главе 4, обеспечивает надежность и эффективность генератора кода. Во-вторых, при этом облегчается перенастройка генератора кода для другой целевой машины. Создание генератора кода для новой машины сводится к разработке грамматики, описывающей инструкции новой машины. В-третьих, качество генерируемого кода может быть сделано весьма высоким за счет добавления продукций для особых случаев, чтобы использовать преимущества машинных идиом.

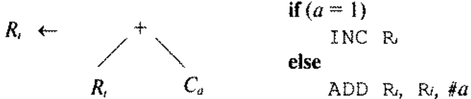
Однако при этом подходе имеется и ряд трудностей. При синтаксическом анализе фиксирован порядок вычислений слева направо. Кроме того, для некоторых машин с большим количеством режимов адресации грамматика, описывающая целевую машину (и, соответственно, сам синтаксический анализатор), становится необычайно большой. Как следствие необходимы специальные методы для работы с грамматикой описания целевой машины. При разработке грамматики следует быть крайне осторожным, чтобы полученный анализатор не мог быть заблокирован в процессе разбора дерева выражения из-за того, что грамматика не обрабатывает некоторые шаблоны операторов или синтаксический анализатор выполнил неверное разрешение возникшего конфликта. Следует также гарантировать невозможность входа синтаксического анализатора в бесконечный цикл сверток продукций с одним символом в правой части. Проблема циклов может быть разрешена путем использования технологии расщепления состояний при генерации таблиц синтаксического анализатора.

8.9.4 Программы семантической проверки

В схеме трансляции для генерации кода встречаются те же атрибуты, что и во входном дереве, но зачастую с ограничениями на значения их индексов. Например, машинная команда может требовать, чтобы значения атрибута находились

в определенном диапазоне или чтобы два значения атрибутов были связаны некоторым соотношением.

Такие ограничения на значения атрибутов могут быть указаны как предикаты, вызываемые перед выполнением свертки. Использование семантических действий и предикатов может обеспечить бóльшую гибкость и простоту описания по сравнению с чисто грамматической спецификацией генератора кода. Для представления классов инструкций могут использоваться обобщенные шаблоны, а семантические действия при этом могут использоваться для выбора команд в частных случаях. Например, с помощью одного шаблона могут быть представлены два варианта команд сложения:



Конфликты действий синтаксического анализа могут быть разрешены с помощью предикатов устранения неоднозначностей, которые обеспечивают использование различных стратегий выбора в разных контекстах. Поскольку некоторые аспекты архитектуры целевой машины, например режимы адресации, могут быть выражены с помощью атрибутов, для целевой машины можно получить описание меньшего размера. Однако при этом трудно проверить, точно ли грамматика атрибутов описывает целевую машину (хотя в той или иной степени эта проблема касается всех генераторов кода).

8.9.5 Обобщенный поиск соответствий

Подход к поиску соответствий с использованием LR-синтаксического анализа основан на префиксном представлении левого операнда бинарного оператора. В префиксном представлении **op** E_1 E_2 решения LR-синтаксического анализа с ограниченным предпросмотром должны приниматься на основе некоторого префикса E_1, поскольку операнд E_1 может быть произвольной длины. Таким образом, поиск соответствий может пропустить ряд моментов целевых команд, связанных с правым операндом.

Вместо префиксного представления можно использовать постфиксное, но при этом все сказанное выше останется в силе, просто будет относиться к правому операнду.

В случае генератора кода, написанного вручную, можно в качестве руководства при написании воспользоваться шаблонами наподобие показанных на рис. 8.20 и написать узкоспециализированный генератор. Например, если корнем дерева является узел с меткой **ind**, то единственный шаблон, соответствующий такому

дереву, — шаблон из правила 5. Если же корень имеет метку +, то соответствие следует искать среди шаблонов из правил 6–8.

В случае генератора генераторов кода нам требуется более общий алгоритм. Можно разработать эффективный нисходящий алгоритм путем расширения методов поиска соответствия строковым шаблонам из главы 3. Идея заключается в представлении каждого шаблона в виде набора строк, где строки соответствуют путям от корня к листу в шаблоне. Все операнды рассматриваются одинаково при помощи включения в строки номеров позиций дочерних узлов слева направо.

Пример 8.22. При построении множества строк для набора команд мы опускаем индексы, поскольку соответствие шаблонам основано только на самих атрибутах, но не на их значениях.

Шаблоны на рис. 8.22 имеют следующий набор строк от корня к листьям:

$$
\begin{array}{l}
C \\
+ \ 1 \ R \\
+ \ 2 \ \textbf{ind} \ 1 \ + \ 1 \ C \\
+ \ 2 \ \textbf{ind} \ 1 \ + \ 2 \ R \\
+ \ 2 \ R
\end{array}
$$

Строка C представляет шаблон с C в качестве корня. Строка $+ \ 1 \ R$ — + и его левый операнд R в двух шаблонах, у которых корень помечен как +. □

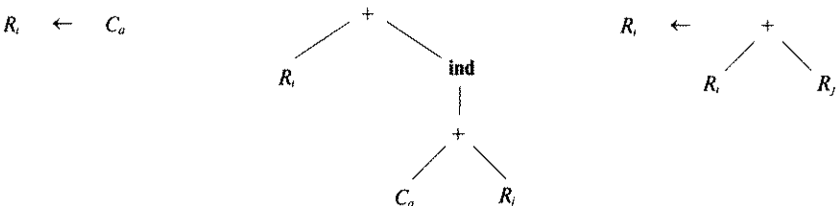


Рис. 8.22. Набор команд для поиска соответствий

Используя наборы строк наподобие показанного в примере 8.22, можно построить программу поиска соответствия деревьям с использованием методов эффективного параллельного поиска строк.

На практике переписывание дерева может быть реализовано при помощи поиска шаблонов деревьев в процессе обхода входного дерева в глубину и выполнения сверток при последнем посещении узлов.

Учесть стоимости команд можно, связав с каждым правилом преобразования дерева стоимость последовательности машинных команд, генерируемых при применении этого правила. В разделе 8.11 мы рассмотрим алгоритм динамического

программирования, который можно использовать вместе с поиском соответствия шаблону.

При работе алгоритма динамического программирования можно выбрать оптимальную последовательность соответствий с использованием информации о стоимости каждого правила. Может оказаться необходимым отложить принятие решения до того момента, когда будут известны стоимости всех альтернатив. При таком подходе на основании схемы преобразования дерева может быть быстро построен небольшой эффективный генератор кода. Кроме того, алгоритм динамического программирования освобождает разработчика генератора кода от необходимости разрешать конфликты или принимать решения о порядке вычислений.

8.9.6 Упражнения к разделу 8.9

Упражнение 8.9.1. Постройте синтаксические деревья для каждой из следующих инструкций в предположении, что все неконстантные операнды располагаются в ячейках памяти.

а) `x = a * b + c * d;`

б) `x[i] = y[j] * z[k];`

в) `x = x + 1;`

Воспользуйтесь схемой, представленной на рис. 8.20, для генерации кода для каждой инструкции.

Упражнение 8.9.2. Повторите упражнение 8.9.1 с использованием схемы синтаксически управляемой трансляции на рис. 8.21 вместо схемы преобразования дерева.

! **Упражнение 8.9.3.** Дополните схему преобразования дерева на рис. 8.20, чтобы она была применима к инструкциям while.

! **Упражнение 8.9.4.** Как следует изменить преобразование деревьев, чтобы оно стало применимым к ориентированным ациклическим графам?

8.10 Генерация оптимального кода для выражений

Если базовый блок состоит из вычисления одного выражения или если мы принимаем, что достаточно генерировать код блока по одному выражению, то можно выбрать регистры оптимальным образом. В приведенном далее алгоритме представлена схема нумерации узлов в дереве выражения (синтаксическом дереве

для выражения), которая позволяет сгенерировать оптимальный код для дерева выражения при наличии фиксированного количества регистров для вычисления этого выражения.

8.10.1 Числа Ершова

Начнем с назначения узлам дерева выражения чисел, которые указывают, сколько регистров требуется для вычисления этого узла без сохранения временных переменных. Эти числа иногда называют числами *Ершова*, после того как А. Ершов (A. Ershov) использовал подобную схему для машин с единственным арифметическим регистром. В нашей модели мы пользуемся следующими правилами.

1. Все листья имеют метку 1.
2. Метка внутреннего узла с одним дочерним узлом равна метке дочернего узла.
3. Метка внутреннего узла с двумя дочерними равна:
   а) если метки дочерних узлов различны — наибольшей из меток дочерних узлов;
   б) если метки дочерних узлов совпадают — метке дочернего узла, увеличенной на 1.

Пример 8.23. На рис. 8.23 показано дерево выражения (с опущенными операторами), которое может быть деревом для выражения $(a - b) + e \times (c + d)$ или для соответствующего трехадресного кода

```
t1 = a - b
t2 = c + d
t3 = e * t2
t4 = t1 + t3
```

Каждый из пяти листьев получает метку 1 в соответствии с первым правилом. Затем можно пометить внутренний узел для `t1=a-b`, поскольку оба его дочерних узла помечены. Здесь используется правило 3*б*, так что данный узел получает метку, на 1 бóльшую метки его дочерних узлов, т.е. 2. То же правило применимо и к внутреннему узлу `t2=c+d`.

Теперь можно перейти к узлу `t3=e*t2`. Его дочерние узлы имеют метки 1 и 2, так что в соответствии с правилом 3*а* метка узла `t3` равна максимальной из них, т.е. 2. Наконец, корень — узел для `t4=t1+t3` — имеет два дочерних узла с меткой 2, так что он получает метку 3. □

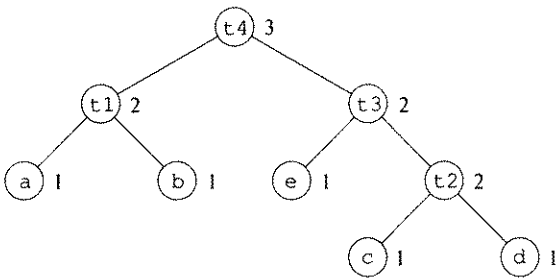


Рис. 8.23. Дерево, помеченное числами Ершова

8.10.2 Генерация кода на основе помеченных деревьев выражений

Можно доказать, что в нашей модели машины (где все операнды должны располагаться в регистрах и регистры могут использоваться и как операнды, и как результаты операции) метка узла указывает наименьшее количество регистров, необходимое для вычисления выражения без сохранения временных результатов. Поскольку в этой модели мы должны загружать каждый операнд и вычислять результат для каждого внутреннего узла, единственная вещь, которая может сделать код неоптимальным, — излишние сохранения временных значений. Аргументом в пользу этого утверждения служит приведенный далее алгоритм для генерации кода без сохранения временных значений с использованием количества регистров, равного метке корня.

Алгоритм 8.24. Генерация кода для помеченного дерева выражения

ВХОД: помеченное дерево, в котором каждый операнд появляется по одному разу (т.е. отсутствуют общие подвыражения).

ВЫХОД: оптимальная последовательность машинных команд для вычисления корня в регистр.

МЕТОД: приведенный далее рекурсивный алгоритм для генерации машинного кода. Описанные ниже шаги применяются к корню дерева. Если алгоритм применяется к узлу с меткой k, будут использоваться ровно k регистров. В алгоритме имеется "база" $b \geqslant 1$ для используемых регистров, т.е. фактически используемыми регистрами будут $R_b, R_{b+1}, \ldots, R_{b+k-1}$. Результат всегда вычисляется в регистр R_{b+k-1}.

1. Для генерации машинного кода для внутреннего узла с меткой k и двумя дочерними узлами с равными метками (которые должны быть равны $k-1$) выполняем следующее.

а) Рекурсивно генерируем код для правого дочернего узла с базой $b + 1$. Результат правого дочернего узла находится в регистре R_{b+k-1}.

б) Рекурсивно генерируем код для левого дочернего узла с базой b. Результат левого дочернего узла находится в регистре R_{b+k-2}.

в) Генерируем команду `OP` $R_{b+k-1}, R_{b+k-2}, R_{b+k-1}$, где `OP` — операция в рассматриваемом внутреннем узле.

2. Предположим, что у нас имеется внутренний узел с меткой k и дочерние узлы с разными метками. Тогда один из дочерних узлов, назовем его "большим", имеет метку k, а второй — "маленький" — некоторую метку $m < k$. Для генерации кода для этого внутреннего узла с использованием базы b делаем следующее.

   а) Рекурсивно генерируем код для большого дочернего узла с использованием базы b; результат находится в регистре R_{b+k-1}.

   б) Рекурсивно генерируем код для маленького узла с использованием базы b; результат находится в регистре R_{b+m-1}. Заметим, что, поскольку $m < k$, при вычислениях не используются ни регистр R_{b+k-1}, ни какой-либо иной регистр с бо́льшим номером.

   в) Генерируем команду `OP` $R_{b+k-1}, R_{b+m-1}, R_{b+k-1}$ или `OP` $R_{b+k-1}, R_{b+k-1}, R_{b+m-1}$ в зависимости от того, является большой дочерний узел соответственно правым или левым дочерним узлом.

3. Для листа, представляющего операнд x, при использовании базы b генерируем команду `LD` R_b, x. □

Пример 8.25. Применим алгоритм 8.24 к дереву на рис. 8.23. Поскольку метка корня равна 3, результат будет находиться в регистре R_3, а использоваться при вычислениях будут только регистры R_1, R_2 и R_3. База для корня — $b = 1$. Поскольку корень имеет два дочерних узла с равными метками, сначала генерируем код для правого дочернего узла с базой 2.

При генерации кода для правого дочернего узла корня с меткой `t3` мы выясняем, что его правый дочерний узел — большой, а левый — маленький. Соответственно, сначала мы генерируем код для правого дочернего узла с базой $b = 2$. Применяя правила для дочерних узлов с одинаковыми метками и для листьев, генерируем следующий код для узла с меткой `t2`:

```
LD  R3, d
LD  R2,c
ADD R3, R2, R3
```

Теперь сгенерируем код для левого дочернего узла, который представляет собой узел с меткой e. Поскольку $b = 2$, генерируется команда

```
LD  R2, e
```

После этого код для правого дочернего узла корня можно завершить, добавив команду

```
MUL R3, R2, R3
```

Далее алгоритм генерирует код для левого дочернего узла корня, получая результат в регистре R_2 и используя базу $b = 1$. Полностью сгенерированная последовательность команд показана на рис. 8.24. □

```
LD   R3, d
LD   R2,c
ADD  R3, R2, R3
LD   R2, e
MUL  R3, R2, R3
LD   R2, b
LD   R1,a
SUB  R2, R1, R2
ADD  R3, R2, R3
```

Рис. 8.24. Оптимальный код, использующий три регистра, для дерева на рис. 8.23

8.10.3 Вычисление выражений при недостаточном количестве регистров

Если количество доступных регистров меньше метки корня дерева, то непосредственно применить алгоритм 8.24 невозможно. Нам потребуется ввести в код несколько команд, которые сбрасывают значения поддеревьев в память, а затем при необходимости загружают их оттуда. Далее приведен модифицированный алгоритм, принимающий во внимание ограниченное количество регистров.

Алгоритм 8.26. Генерация кода для помеченного дерева выражения

ВХОД: помеченное дерево, в котором каждый операнд появляется по одному разу (т.е. отсутствуют общие подвыражения) и количество регистров $r \geqslant 2$.

ВЫХОД: оптимальная последовательность машинных команд для вычисления корня в регистр с использованием не более чем r регистров (регистры $R_1, R_2, \ldots, R_r$).

МЕТОД: приведенный далее рекурсивный алгоритм для генерации машинного кода. Описанные ниже шаги применяются к корню дерева с использованием базы

$b = 1$. Для узла N с меткой r или меньшей алгоритм совпадает с алгоритмом 8.24, и здесь эти шаги описаны не будут. Однако для внутренних узлов с метками $k > r$ мы должны работать с каждой стороной дерева отдельно и сохранять результат большего поддерева. Этот результат затем загружается из памяти непосредственно перед вычислением N, и на последнем шаге мы работаем с регистрами R_{r-1} и R_r. Базовый алгоритм модифицируется следующим образом.

1. Узел N имеет как минимум один дочерний узел с меткой r или выше. Выбираем больший дочерний узел (любой, если их метки одинаковы) как "большой", а оставшийся — как "маленький".

2. Рекурсивно генерируем код для большого дочернего узла с использованием базы $b = 1$. Результат вычислений будет находиться в регистре R_r.

3. Генерируем машинную команду $ST \;\; t_k, R_r$, где t_k — временная переменная, используемая для вычисления узлов с меткой k.

4. Генерируем код для маленького дочернего узла следующим образом. Если маленький узел имеет метку r или бóльшую, выбираем $b = 1$. Если метка маленького узла $j < r$, то выбираем базу $b = r - j$. Затем рекурсивно применяем этот алгоритм к маленькому узлу; результат вычислений будет находиться в регистре R_r.

5. Генерируем команду $LD \;\; R_{r-1}, t_k$.

6. Если большой узел является правым дочерним узлом N, генерируем команду OP R_r, R_r, R_{r-1}; если же левым — команду OP R_r, R_{r-1}, R_r. □

Пример 8.27. Вновь обратимся к выражению, представленному на рис. 8.23, но теперь будем считать, что $r = 2$, т.е. что для хранения временных значений при вычислении выражения у нас есть только два регистра — R1 и R2. При применении алгоритма 8.26 к рис. 8.23 мы видим, что корень имеет метку 3, большую, чем $r = 2$. Таким образом, нам надо идентифицировать один из дочерних узлов как "большой". Поскольку у обоих дочерних узлов метки одинаковы, можно выбрать в качестве большого любой. Предположим, что нами выбран правый узел.

Поскольку метка большого дочернего узла равна 2, регистров для его вычисления достаточно. Поэтому к данному поддереву применяем алгоритм 8.24 с $b = 1$ и двумя регистрами. Результат выглядит очень похожим на показанный на рис. 8.24, но с регистрами R1 и R2 вместо R2 и R3:

```
LD  R2, d
LD  R1, c
ADD R2, R1, R2
```

```
LD  R1, e
MUL R2, R1, R2
```

Теперь, поскольку для вычисления левого дочернего узла корня нам нужны оба регистра, генерируем команду

```
ST  t3, R2
```

Приступим к левому дочернему узлу корня. Здесь вновь регистров хватает для вычислений, так что генерируется код

```
LD  R2, b
LD  R1, a
SUB R2, R1, R2
```

И наконец загружаем временное значение правого поддерева корня при помощи команды

```
LD  R1, t3
```

и выполняем операцию в корне дерева с помощью команды

```
ADD R2, R2, R1
```

Полностью последовательность команд показана на рис. 8.25. □

```
LD   R2, d
LD   R1, c
ADD  R2, R1, R2
LD   R1, e
MUL  R2, R1, R2
ST   t3, R2
LD   R2, b
LD   R1, a
SUB  R2, R1, R2
LD   R1, t3
ADD  R2, R2, R1
```

Рис. 8.25. Оптимальный код для дерева на рис. 8.23, использующий только два регистра

8.10.4 Упражнения к разделу 8.10

Упражнение 8.10.1. Вычислите числа Ершова для следующих выражений:

а) $a/(b+c) - d*(e+f)$;

б) $a + b * (c * (d + e))$;

в) $(-a + *p) * ((b - *q)/(-c + *r))$.

Упражнение 8.10.2. Сгенерируйте оптимальный код с использованием двух регистров для каждого выражения из упражнения 8.10.1.

Упражнение 8.10.3. Сгенерируйте оптимальный код с использованием трех регистров для каждого выражения из упражнения 8.10.1.

Упражнение 8.10.4. Обобщите вычисление чисел Ершова для деревьев выражений со внутренними узлами с тремя и более дочерними узлами.

Упражнение 8.10.5. Присваивание элементу массива, такое как `a[i]=x`, кажется имеющим три операнда: a, i и x. Каким образом можно модифицировать схему дерева с метками для генерации оптимального кода для нашей модели машины?

Упражнение 8.10.6. Исходно числа Ершова использовались на машине, у которой правый операнд выражения мог располагаться в памяти, а не в регистре. Каким образом можно модифицировать схему дерева с метками для генерации оптимального кода для этой модели машины?

Упражнение 8.10.7. Некоторые машины требуют двух регистров для некоторых значений однократной точности. Предположим, что результат умножения величин, для хранения которых требуется по одному регистру, размещается в двух смежных регистрах, а при делении a/b значение a должно находиться в двух смежных регистрах. Каким образом можно модифицировать схему дерева с метками для генерации оптимального кода для этой модели машины?

8.11 Генерация кода с использованием динамического программирования

Алгоритм 8.26 из раздела 8.10 генерирует оптимальный код из дерева выражения за время, линейно зависящее от размера дерева. Эта процедура работает для машин, у которых все вычисления выполняются в регистрах, а команды состоят из операторов, применяемых к двум регистрам или к регистру и ячейке памяти.

Для расширения класса машин, для которых возможно построение оптимального кода на основе деревьев выражений за линейно зависящее от размера дерева время, можно воспользоваться алгоритмом на основе динамического программирования.

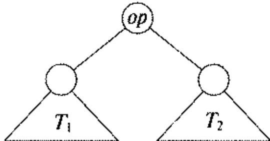
Алгоритм динамического программирования может использоваться для генерации кода для любой машины с r взаимозаменяемыми регистрами `R0`, `R1`, ..., $Rr - 1$ и командами загрузки, сохранения и операций. Для простоты мы полагаем,

что стоимость любой команды равна единице, хотя алгоритм динамического программирования можно легко модифицировать для случая, когда каждая команда обладает собственной стоимостью.

8.11.1 Последовательные вычисления

Алгоритм динамического программирования разбивает задачу генерации оптимального кода для выражения на подзадачи генерации оптимального кода для подвыражений исходного выражения. В качестве простого примера рассмотрим выражение E вида $E_1 + E_2$. Оптимальная программа для E образуется путем комбинации оптимальных программ для E_1 и E_2 в том или ином порядке, за которой следует код вычисления оператора +. Подзадачи генерации оптимального кода для E_1 и E_2 решаются аналогично.

Оптимальная программа, порожденная алгоритмом динамического программирования, имеет важное свойство, заключающееся в том, что она вычисляет выражение $E = E_1$ **op** E_2 "последовательно". Чтобы понять, что это означает, можно рассмотреть синтаксическое дерево T для E.



Здесь T_1 и T_2 — деревья для E_1 и E_2 соответственно.

Мы говорим, что программа P вычисляет дерево T *последовательно* (contiguously), если она вначале вычисляет те поддеревья T, которые должны быть вычислены в память. Затем программа вычисляет оставшуюся часть T либо в порядке T_1, T_2 и затем корень, либо — T_2, T_1 и затем корень. В любом случае при необходимости используются предварительно вычисленные значения, хранящиеся в памяти. В качестве примера непоследовательного вычисления P сначала можно вычислить часть T_1, оставив полученное значение в регистре, а не в памяти, затем вычислить T_2 и вновь возвратиться к оставшейся части T_1.

Можно доказать, что для регистровой машины из данного раздела для любой программы P, вычисляющей дерево выражения T, можно найти эквивалентную программу P', такую, что

1) стоимость P' не больше стоимости P;

2) P' использует не больше регистров, чем P;

3) P' вычисляет дерево последовательно.

Из этого результата следует, что каждое дерево выражения можно вычислить оптимальным образом с помощью последовательной программы.

Кстати, для машин с четно-нечетными парами регистров не всегда существует оптимальное последовательное вычисление; архитектура x86 использует пары регистров для умножения и вычитания. Для таких машин можно привести примеры деревьев выражений, по которым оптимальная программа на машинном языке должна вначале вычислить в регистр часть левого поддерева корня, затем — часть правого поддерева, затем — еще одну часть левого поддерева, затем — еще одну часть правого и т.д. Такие осцилляции не требуются для оптимального вычисления какого бы то ни было дерева выражения на машине, используемой в данном разделе.

Свойство последовательного вычисления, определенное выше, говорит о том, что для любого дерева выражения T всегда существует оптимальная программа вычисления, состоящая из оптимальных программ вычисления поддеревьев корня, за которыми следует команда вычисления корня. Это свойство позволяет нам использовать алгоритм динамического программирования для создания оптимальной программы вычисления T.

8.11.2 Алгоритм динамического программирования

Данный алгоритм состоит из трех фаз. Предполагается, что целевая машина содержит r регистров.

1. В восходящем порядке для каждого узла n дерева выражения T вычисляется массив стоимости C, i-й элемент которого $C[i]$ представляет собой оптимальную стоимость вычисления в регистр поддерева S с корнем n, при условии, что для вычисления доступныi регистров, $1 \leqslant i \leqslant r$.

2. Обход дерева выражения T с использованием векторов стоимости для определения, какие из поддеревьев T должны быть вычислены в память.

3. Обход каждого дерева с использованием векторов стоимости и связанных с ними команд для генерации конечного целевого кода. Первым генерируется код для поддеревьев, вычисляемых в память.

Каждая из фаз может быть выполнена за время, линейно пропорциональное размеру дерева выражения.

Стоимость вычисления узла n включает загрузки и сохранения, необходимые для вычисления S с данным количеством регистров. Она также включает стоимость вычисления оператора в корне S. Нулевой компонент вектора стоимости — оптимальная стоимость вычисления поддерева S в память. Свойство последовательного вычисления гарантирует, что оптимальная программа для S может быть сгенерирована путем рассмотрения комбинаций только оптимальных программ для поддеревьев с корнем S. Такое ограничение снижает количество случаев, которые необходимо рассмотреть.

Для вычисления стоимости $C[i]$ в узле n будем рассматривать команды как правила преобразования дерева, как в разделе 8.9. Рассмотрим каждый шаблон E, который соответствует входному дереву в узле n. Изучая векторы стоимости в соответствующих наследниках n, определим стоимости вычисления операндов в листьях E. Для операндов E, представляющих собой регистры, рассмотрим все возможные порядки вычислений поддеревьев T в регистры. Для каждого из рассматриваемых порядков вычисления первое поддерево, соответствующее регистровому операнду, можно вычислить с использованием i доступных регистров, второе — с помощью $i - 1$ и т.д. При подсчете стоимости для узла n добавим стоимость команды, связанной с шаблоном E. Значение $C[i]$ в таком случае представляет собой минимальную стоимость среди всех возможных порядков вычисления.

Векторы стоимости для всего дерева T могут быть вычислены в восходящем порядке (снизу вверх) за время, линейно зависящее от количества узлов в дереве T. Команды, используемые для получения наилучшей стоимости $C[i]$ для каждого значения i, удобно хранить в узлах дерева; при этом наименьшая стоимость в векторе корневого узла T дает минимальную стоимость вычисления T.

Пример 8.28. Рассмотрим машину с регистрами R0 и R1 и со следующими командами, каждая из которых имеет единичную стоимость:

```
LD   Ri, Mj         // Ri = Mj
op   Ri, Ri, Rj     // Ri = Ri op Rj
op   Ri, Ri, Mj     // Ri = Ri op Mj
LD   Ri, Rj         // Ri = Rj
ST   Mi, Rj         // Mi = Rj
```

Здесь Ri представляет собой либо R0, либо R1, а Mj — адрес в памяти. Оператор *op* соответствует арифметическому оператору.

Применим алгоритм динамического программирования, чтобы сгенерировать оптимальный код для синтаксического дерева, приведенного на рис. 8.26. В первой фазе алгоритма мы вычисляем векторы стоимости, показанные около каждого узла. Чтобы проиллюстрировать вычисление стоимости, рассмотрим вектор стоимости у листа a. $C[0]$, стоимость вычисления a в память, равна нулю, поскольку a и так находится в памяти. Стоимость $C[1]$ вычисления a в регистр равна 1, поскольку мы можем загрузить это значение в регистр с помощью одной команды LD R0,a. $C[2]$, стоимость загрузки a в регистр при двух доступных регистрах, та же, что и при одном. Таким образом, вектор стоимости в листе a представляет собой (0,1,1).

Рассмотрим вектор стоимости в корне. Сначала мы определяем минимальную стоимость вычисления корня при одном или двух доступных регистрах. Поскольку корень помечен оператором +, корню соответствует машинная команда

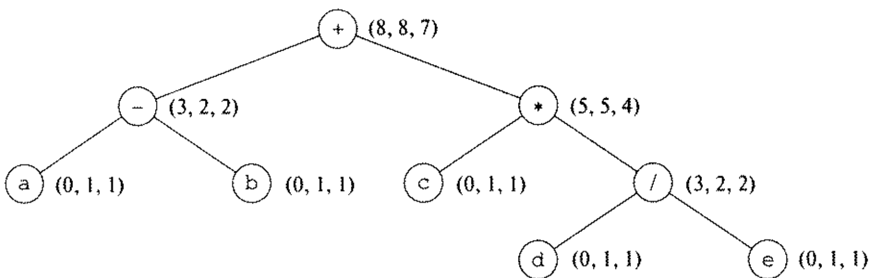


Рис. 8.26. Синтаксическое дерево для `(a-b)+c*(d/e)` с векторами стоимости в каждом узле

`ADD R0,R0,M`. При использовании этой команды минимальная стоимость вычисления корня с одним доступным регистром представляет собой сумму минимальной стоимости вычисления правого поддерева в память, минимальной стоимости вычисления левого поддерева в регистр и еще одной единицы для команды сложения. Других способов вычисления нет. Векторы стоимости правого и левого потомков корня показывают, что минимальная стоимость вычисления корня при одном доступном регистре равна $5 + 2 + 1 = 8$.

Теперь рассмотрим минимальную стоимость вычисления корня с двумя доступными регистрами. Здесь возможны три варианта, зависящие от того, какая из команд используется для вычисления корня и в каком порядке вычисляются левые и правые поддеревья.

1. Вычисляем левое поддерево в регистр `R0` при двух доступных регистрах; после этого вычисляем правое поддерево в регистр `R1` при одном доступном регистре и для вычисления корня используем команду `ADD R0,R0,R1`. Такая последовательность действий имеет стоимость $2 + 5 + 1 = 8$.

2. Вычисляем правое поддерево в регистр `R1` при двух доступных регистрах; после этого вычисляем левое поддерево в регистр `R0` при одном доступном регистре и для вычисления корня используем команду `ADD R0,R0,R1`. Такая последовательность действий имеет стоимость $4 + 2 + 1 = 7$.

3. Вычисляем правое поддерево в память по адресу `M`, вычисляем левое поддерево в регистр `R0` при двух доступных регистрах и для вычисления корня применяем команду `ADD R0,R0,M`. Такая последовательность действий имеет стоимость $5 + 2 + 1 = 8$.

Второй вариант дает минимальную стоимость, равную 7.

Минимальная стоимость вычисления корня в память определяется прибавлением 1 к минимальной стоимости вычисления корня при всех доступных регистрах, т.е. мы вычисляем корень в регистр, а затем сохраняем полученный результат. Таким образом, вектор стоимости в корне представляет собой (8,8,7).

По векторам стоимости мы можем легко построить код путем обхода дерева. Для дерева на рис. 8.26 в предположении доступности двух регистров оптимальный код имеет следующий вид:

```
LD   R0, c          // R0 = c
LD   R1, d          // R1 = d
DIV  R1, R1, e      // R1 = R1 / e
MUL  R0, R0, R1     // R0 = R0 * R1
LD   R1, a          // R1 = a
SUB  R1, R1, b      // R1 = R1 - b
ADD  R1, R1, R0     // R1 = R1 + R0
```
□

Методы динамического программирования использовались в ряде компиляторов, включая вторую версию переносимого компилятора C, PCC2. Такая методика упрощает перенос на другую целевую машину в силу применимости методов динамического программирования для широкого класса машин.

8.11.3 Упражнения к разделу 8.11

Упражнение 8.11.1. Расширьте схему, представленную на рис. 8.20, стоимостями и используйте динамическое программирование для генерации кода для инструкций из упражнения 8.9.1.

!! **Упражнение 8.11.2.** Каким образом можно применить динамическое программирование для генерации оптимального кода на основе ориентированных ациклических графов?

8.12 Резюме к главе 8

- *Генерация кода* представляет собой завершающую стадию компилятора. Генератор кода отображает промежуточное представление, произведенное начальной стадией (или, при наличии фазы оптимизации кода, оптимизатором) в целевую программу.

- *Выбор команд* — процесс выбора команд целевого языка для каждой инструкции промежуточного представления.

- *Распределение регистров* — процесс принятия решения о том, какие значения промежуточного представления должны храниться в регистрах. Эффективным методом распределения регистров в компиляторе служит раскраска графа.

- *Назначение регистров* представляет собой процесс принятия решения о том, в каком регистре должно храниться данное значение промежуточного представления.

- *Перенастраиваемым* называется компилятор, который может генерировать код для нескольких наборов команд.

- *Виртуальная машина* представляет собой интерпретатор промежуточного языка — байт-кода, генерируемого для таких языков программирования, как Java и C#.

- *CISC-компьютер* обычно представляет собой двухадресную машину с относительно небольшим количеством регистров, несколькими классами регистров и командами переменной длины со сложными режимами адресации.

- *RISC-компьютер* обычно представляет собой трехадресную машину с большим количеством регистров и операциями, выполняемыми над регистрами.

- *Базовый блок* представляет собой максимальную последовательность следующих друг за другом трехадресных инструкций, входить в которую поток управления может только через первую инструкцию последовательности, а выходить — только через последнюю, без останова и ветвления, за исключением, возможно, последней инструкции базового блока.

- *Граф потока* — графическое представление программы, в котором узлами графа являются базовые блоки, а ребра графа указывают возможные пути перехода управления между блоками.

- *Циклом* графа потока является сильно связанная область с единственной точкой входа, именуемой входом в цикл.

- Представление базового блока в виде *ориентированного ациклического графа* является графом, узлы которого представляют инструкции базового блока, и каждый дочерний узел некоторого узла соответствует инструкции, являющейся последним определением операнда, используемого в инструкции узла-родителя.

- *Локальная оптимизация* представляет собой локально улучшающие код преобразования, которые могут применяться к программе обычно с использованием перемещающегося по коду окна.

- *Выбор команд* может быть выполнен при помощи процесса преобразования дерева, в котором для замощения синтаксического дерева используются

шаблоны, соответствующие машинным командам. С правилами преобразования дерева можно связать стоимости и применить динамическое программирование для получения оптимального замощения для определенных классов машин и выражений.

- *Число Ершова* говорит о том, сколько регистров необходимо для вычисления выражения без сохранения временных значений в памяти.

- *Сброс* представляет собой последовательность команд, которые сохраняют значение из регистра в ячейке памяти, чтобы освободить регистр для хранения другого значения.

8.13 Список литературы к главе 8

Многие методы, описанные в этой главе, были разработаны в первых компиляторах. Алгоритм меток Ершова (Ershov) создан еще в 1958 году [7]. Сети (Sethi) и Ульман (Ullman) [16] использовали этот метод в алгоритме, который, как они доказали, генерирует оптимальный код для арифметических выражений. Ахо (Aho) и Джонсон (Johnson) [1] использовали динамическое программирование для генерации оптимального кода для деревьев выражений на CISC-машинах. В книге Хеннеси (Hennessy) и Паттерсона (Patterson) [12] можно найти неплохое описание эволюции CISC- и RISC-архитектур и компромиссов при проектировании хороших наборов команд.

Архитектуры RISC стали популярны после 1990 года, хотя их родословная ведет начало от компьютеров типа CDC 6000 1964 года выпуска. Большинство машин, разработанных до 1990 года, были CISC-машинами, но и после 1990 года соотношение не изменилось в силу того, что машины на основе архитектуры Intel 80x86 и их наследники, такие как Pentium, являются CISC-машинами. Разработанная в 1963 году Burroughs B5000 являлась стековой машиной.

Многие эвристики для генерации кодов, предложенные в этой главе, использованы в разных компиляторах. Наша стратегия распределения фиксированного количества регистров для хранения переменных при выполнении цикла использовалась в реализации Fortran H Лоури (Lowry) и Медлоком (Medlock) [13].

Эффективные методы распределения регистров также изучались еще при разработке первых компиляторов. Раскраска графа как метод распределения регистров была предложена Коком (Cocke), Ершовым (Ershov) [8] и Шварцем (Schwartz) [15]. Для распределения регистров предлагались различные варианты задачи о раскраске графа. Мы следовали в своем изложении Чаитину (Chaitin) [3 и 4]. Чоу (Chow) и Хеннеси (Hennessy) описали свой алгоритм раскраски графа с приоритетами для распределения регистров в [5]. В [6] можно найти обзор

последних методов разделения графов и переписывания для распределения регистров.

Генераторы лексических и синтаксических анализаторов стимулировали разработку выбора команд, управляемую шаблонами. Гланвилль (Glanville) и Грехем (Graham) [11] использовали методы генерации LR-синтаксического анализатора для автоматизации выбора команд. Генераторы кода, управляемые таблицами, эволюционировали в различный инструментарий генерации кода [14]. Ахо (Aho), Ганапати (Ganapathi) и Тжянг (Tjiang) [2] скомбинировали эффективные методы поиска соответствия шаблонам и динамического программирования в инструменте для генерации кода под названием *twig*. Фразер (Fraser), Хенсон (Hanson) и Пробстинг (Proebsting) [10] усовершенствовали эти идеи в своем простом и эффективном генераторе генераторов кода.

1. Aho, A. V. and S. C. Johnson, “Optimal code generation for expression trees”, *J. ACM* **23**:3, pp. 488–501.

2. Aho, A. V., M. Ganapathi, and S. W. K. Tjiang, “Code generation using tree matching and dynamic programming”, *ACM Trans. Programming Languages and Systems* **11**:4 (1989), pp. 491–516.

3. Chaitin, G. J., M. A. Auslander, A. K. Chandra, J. Cocke, M. E. Hopkins, and P. W. Markstein, “Register allocation via coloring”, *Computer Languages* **6**:1 (1981), pp. 47–57.

4. Chaitin, G. J., “Register allocation and spilling via graph coloring”, *ACM SIGPLAN Notices* **17**:6 (1982), pp. 201–207.

5. Chow, F. and J. L. Hennessy, “The priority-based coloring approach to register allocation”, *ACM Trans. Programming Languages and Systems* **12**:4 (1990), pp. 501–536.

6. Cooper, K. D. and L. Torczon, *Engineering a Compiler*, Morgan Kaufmann, San Francisco CA, 2004.

7. Ershov, A. P., “On programming of arithmetic operations”, *Comm. ACM* **1**:8 (1958), pp. 3–6. Also, *Comm. ACM* 1:9 (1958), p. 16.

8. Ershov, A. P., *The Alpha Automatic Programming System*, Academic Press, New York, 1971.

9. Fischer, C. N. and R. J. LeBlanc, *Crafting a Compiler with C*, Benjamin-Cummings, Redwood City, CA, 1991.

10. Fraser, C. W., D. R. Hanson, and T. A. Proebsting, "Engineering a simple, efficient code generator generator", *ACM Letters on Programming Languages and Systems* **1**:3 (1992), pp. 213–226.

11. Glanville, R. S. and S. L. Graham, "A new method for compiler code generation", *Conf. Rec. Fifth ACM Symposium on Principles of Programming Languages* (1978), pp. 231–240.

12. Hennessy, J. L. and D. A. Patterson, *Computer Architecture: A Quantitative Approach*, Third Edition, Morgan Kaufman, San Francisco, 2003.

13. Lowry, E. S. and C. W. Medlock, "Object code optimization", *Comm. ACM* **12**:1 (1969), pp. 13–22.

14. Pelegri-Llopart, E. and S. L. Graham, "Optimal code generation for expressions trees: an application of BURS theory", *Conf. Rec. Fifteenth Annual ACM Symposium on Principles of Programming Languages* (1988), pp. 294–308.

15. Schwartz, J. T., *On Programming: An Interim Report on the SETL Project*, Technical Report, Courant Institute of Mathematical Sciences, New York, 1973.

16. Sethi, R. and J. D. Ullman, "The generation of optimal code for arithmetic expressions", *J. ACM* **17**:4 (1970), pp. 715–728.

Глава 9

Машинно-независимые оптимизации

Если независимо транслировать каждую конструкцию высокоуровневого языка программирования в машинный код, то это может приводить к существенным накладным расходам времени выполнения программы. В данной главе рассматривается, как можно этого избежать. Устранение излишних команд в объектном коде или замена последовательности команд более быстрой, выполняющей те же действия последовательностью обычно называется улучшением кода или оптимизацией кода.

Локальная оптимизация кода (улучшения кода в пределах базового блока) рассматривалась в разделе 8.5. В этой главе мы поговорим о *глобальной* оптимизации, улучшения которой учитывают происходящее между базовыми блоками. Начнем с рассмотрения в разделе 9.1 основных возможностей улучшения кода.

Большинство глобальных оптимизаций основано на *анализах потоков данных*, которые представляют собой алгоритмы для сбора информации о программе. Все результаты анализов потоков данных имеют один и тот же вид: для каждой команды программы они указывают некоторое свойство, которое должно выполняться всякий раз при выполнении этой команды. Разные анализы отличаются вычисляемыми ими свойствами. Например, анализ распространения констант вычисляет для каждой точки программы и для каждой переменной, используемой в программе, имеет ли эта переменная в данной точке программы единственное константное значение или нет. Эта информация может, например, использоваться для замены обращений к переменным непосредственными константными значениями. В качестве другого примера анализ живучести переменных для каждой точки программы определяет, будет ли перезаписано значение, хранящееся в переменной, перед его прочтением. Если будет, то мы не должны заботиться о сохранении этого значения в регистре или ячейке памяти.

Анализ потоков данных будет рассматриваться в разделе 9.2 и включать несколько важных примеров видов глобально собираемой информации, используемой затем для улучшения кода. В разделе 9.3 рассматривается обобщенная идея схемы потока данных, для которой анализ потоков данных из раздела 9.2 является частным случаем. Во всех рассматриваемых случаях анализа потоков дан-

ных можно использовать, по сути, одинаковые алгоритмы, производительность которых может быть измерена, а корректность показана для всех случаев. Раздел 9.4 представляет собой пример обобщенной схемы, которая выполняет более мощный анализ, чем в рассмотренных ранее примерах. Затем в разделе 9.5 будет рассмотрен мощный метод оптимизации размещения вычислений выражений в программе под названием "устранение частичной избыточности". Решение этой задачи требует решения ряда различных задач, связанных с потоками данных.

В разделе 9.6 речь пойдет о циклах в программах. Идентификация циклов приводит к другому семейству алгоритмов для решения задач потоков данных, основанному на иерархической структуре циклов в хорошо согласованной ("приводимой") программе. Этот подход к анализу потоков данных рассматривается в разделе 9.7. Наконец, в разделе 9.8 используется иерархический анализ для устранения переменных индукции (в первую очередь, переменных, подсчитывающих количество итераций цикла). Такое улучшение кода — одно из наиболее важных для программ, написанных на распространенных языках программирования.

9.1 Основные источники оптимизации

Оптимизация, выполняемая компилятором, должна сохранять семантику исходной программы. За исключением очень редких случаев, если программист выбрал и реализовал конкретный алгоритм, компилятор не в состоянии достаточно хорошо разобраться в программе, чтобы заменить его совершенно иным более эффективным алгоритмом. Компилятор знает только о том, как применять относительно низкоуровневые семантические преобразования с использованием обобщенных фактов, таких как алгебраические тождества наподобие $i + 0 = i$ или семантика программы наподобие того факта, что выполнение одинаковых операций над одинаковыми значениями даст одинаковые результаты.

9.1.1 Причины избыточности

В типичной программе имеется масса избыточных операций. Иногда избыточность проявляется на уровне исходного текста программы. Например, программист может счесть более удобным вычислить некоторый результат заново, оставляя компилятору распознать, что одно из вычислений излишне. Но более часто избыточность оказывается побочным действием написания программ на языке программирования высокого уровня. В большинстве языков программирования (отличных от C и C++, в которых разрешены арифметические действия с указателями), у программиста нет выбора вариантов обращения к элементам массива или полям структуры наподобие $A[i][j]$ или $X \to f1$.

При компиляции программы каждое из таких обращений к высокоуровневым структурам данных разворачивается в ряд низкоуровневых арифметических операций, таких как вычисление (i, j)-го элемента матрицы A. Обращения к одной и той же структуре данных часто используют много одинаковых низкоуровневых операций. Программист не осведомлен об этих операциях и не может устранить их избыточность самостоятельно. Более того, с точки зрения инженерии программного обеспечения предпочтительно, чтобы программист обращался к элементам данных только по их высокоуровневым именам; такие программы проще писать и, что более важно, проще понимать и развивать. Обладая компилятором с устранением избыточности, мы получаем лучшее из двух миров — программы оказываются одновременно эффективными и легко поддерживаемыми.

9.1.2 Конкретный пример: быстрая сортировка

Далее для иллюстрации некоторых важных улучшающих код преобразований мы будем использовать фрагмент программы быстрой сортировки *quicksort*. Программа на C на рис. 9.1 взята у Седжвика (Sedgewick)[1], который рассматривал варианты ручной оптимизации этой программы. Мы не будем рассматривать все тонкости этой программы, например тот факт, что элемент $a[0]$ должен содержать наименьший из сортируемых элементов, а $a[max]$ — наибольший.

Перед тем как мы сможем оптимизировать код путем удаления избыточности в вычислениях адресов, операции с адресами должны быть разбиты на низкоуровневые арифметические операции для выявления избыточности. В оставшейся части этой главы мы полагаем, что промежуточное представление состоит из трехадресных инструкций, а для хранения всех результатов промежуточных выражений используются временные переменные. Промежуточный код для помеченного фрагмента программы на рис. 9.1 приведен на рис. 9.2.

В этом примере мы полагаем, что размер целого числа — 4 байт. Присваивание `x = a[i]` транслируется, как в разделе 6.4.4, в две приведенные в шагах 14 и 15 на рис. 9.2 трехадресные инструкции:

```
t6 = 4*i
x = a[t6]
```

Аналогично `a[j] = x` в шагах 20 и 21 превращается в

```
t10 = 4*j
a[t10] = x
```

Обратите внимание, что каждое обращение к массиву в исходной программе транслируется в пару шагов, состоящих из умножения и оператора индексирова-

[1] R. Sedgewick, "Implementing Quicksort Programs", *Comm. ACM*, **21**, 1978, pp. 847–857.

```
void quicksort(int m, int n)
    /* Рекурсивная сортировка диапазона от a[m] до a[n] */
{
    int i, j;
    int v, x;
    if (n <= m) return;
    /* Начало фрагмента */
    i = m-1; j = n; v = a[n];
    while (1) {
        do i = i+1; while (a[i] < v);
        do j = j-1; while (a[j] > v);
        if (i >= j) break;
        x = a[i]; a[i] = a[j]; a[j] = x;
            /* Обмен a[i], a[j] */
    }
    x = a[i]; a[i] = a[n]; a[n] = x;
        /* Обмен a[i], a[n] */
    /* Конец фрагмента */
    quicksort(m,j); quicksort(i+1,n);
}
```

Рис. 9.1. Код быстрой сортировки на языке программирования С

```
 (1)  i = m-1                 (16)  t7 = 4*i
 (2)  j = n                   (17)  t8 = 4*j
 (3)  t1 = 4*n                (18)  t9 = a[t8]
 (4)  v = a[t1]               (19)  a[t7] = t9
 (5)  i = i+1                 (20)  t10 = 4*j
 (6)  t2 = 4*i                (21)  a[t10] = x
 (7)  t3 = a[t2]              (22)  goto (5)
 (8)  if t3<v goto (5)        (23)  t11 = 4*i
 (9)  j = j-1                 (24)  x = a[t11]
(10)  t4 = 4*j                (25)  t12 = 4*i
(11)  t5 = a[t4]              (26)  t13 = 4*n
(12)  if t5>v goto (9)        (27)  t14 = a[t13]
(13)  if i>=j goto (23)       (28)  a[t12] = t14
(14)  t6 = 4*i                (29)  t15 = 4*n
(15)  x = a[t6]               (30)  a[t15] = x
```

Рис. 9.2. Трехадресный код для фрагмента на рис. 9.1

ния массива. В результате короткий фрагмент исходной программы транслируется в существенно более длинную последовательность трехадресных операций.

На рис. 9.3 показан граф потока для программы на рис. 9.2. Блок B_1 представляет собой входной узел. Все условные и безусловные переходы к инструкциям на рис. 9.2 заменены на рис. 9.3 переходами к блокам, в которых эти инструкции являются лидерами, как в разделе 8.4. На рис. 9.3 имеется три цикла. Блоки B_2 и B_3 являются циклами сами по себе; еще один цикл с единственной точкой входа B_2 образуют блоки B_2, B_3, B_4 и B_5.

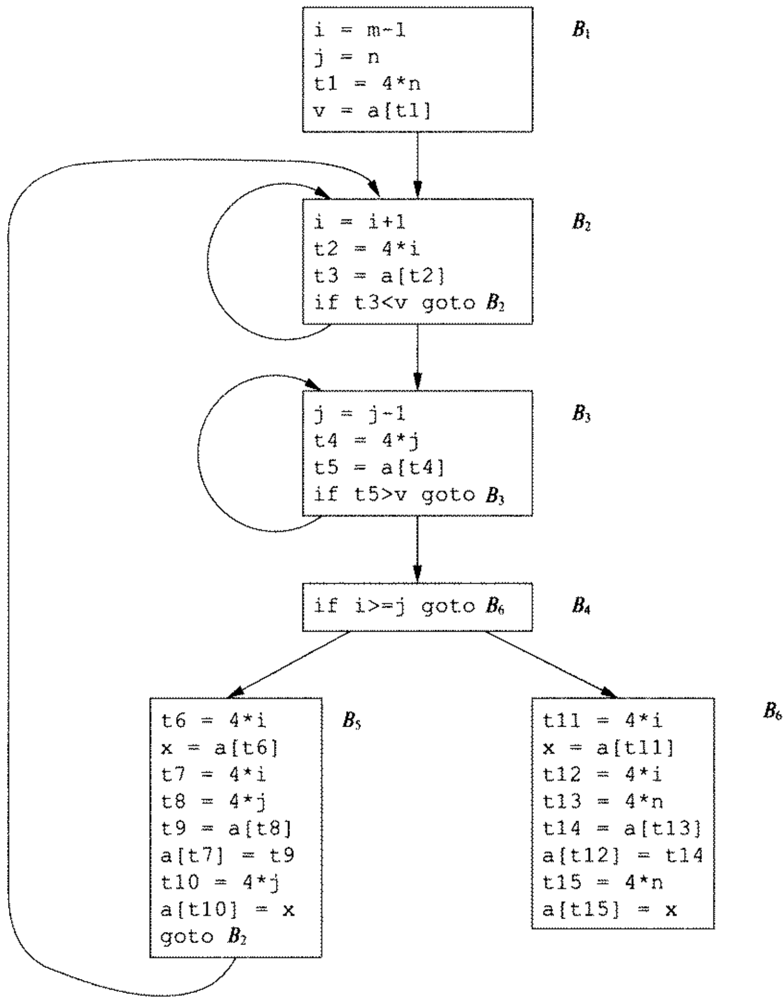
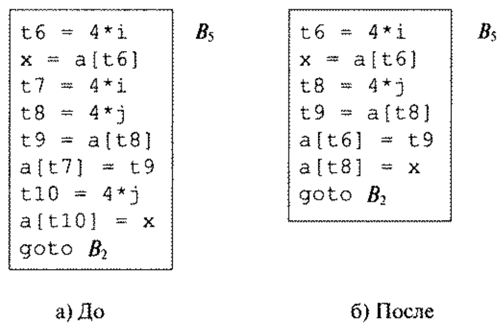


Рис. 9.3. Граф потока для фрагмента быстрой сортировки

9.1.3 Трансформации, сохраняющие семантику

Существует ряд способов, которыми компилятор может улучшить программу без изменения вычисляемой функции. Устранение общих подвыражений, размножение копий, удаление недоступного кода и дублирование констант — вот основные примеры таких преобразований, сохраняющих функции (или *сохраняющих семантику* (semantics-preserving)). Мы рассмотрим все их по очереди.

Зачастую программа включает несколько вычислений одного и того же значения, например смещения в массиве. Как упоминалось в разделе 9.1.2, некоторые из этих повторений не могут быть устранены программистом, поскольку они возникают на более низком уровне детализации, чем доступный в исходном языке. Например, блок B_5, показанный на рис. 9.4, *а*, дважды вычисляет значения $4 * i$ и $4 * j$, хотя ни одно из этих вычислений в исходном тексте явно не указано.



а) До б) После

Рис. 9.4. Локальное устранение общих подвыражений

9.1.4 Глобальные общие подвыражения

Выражение E называется *общим подвыражением* (common subexpression), если E было ранее вычислено и значения переменных в E с того времени не изменились. Повторного вычисления можно избежать, если использовать ранее вычисленное значение; т.е. если переменная x, которой присвоен результат предыдущего вычисления E, не изменялась между вычислениями.[2]

Пример 9.1. Присваивания `t7` и `t10` на рис. 9.4, *а* вычисляют общие подвыражения $4 * i$ и $4 * j$ соответственно. Эти шаги устранены на рис. 9.4, *б*, где вместо `t7` используется `t6`, а вместо `t10` — `t8`. □

[2]Если x изменялась, все равно можно повторно использовать предыдущее вычисление E, если сохранить полученное значение в новой переменной y наряду с сохранением в x и использовать значение y вместо повторного вычисления E.

Пример 9.2. На рис. 9.5 показан результат устранения как локальных, так и глобальных общих подвыражений из блоков B_5 и B_6 в графе потока на рис. 9.3. Рассмотрим сначала преобразование B_5, а затем упомянем о нескольких тонкостях использования массивов.

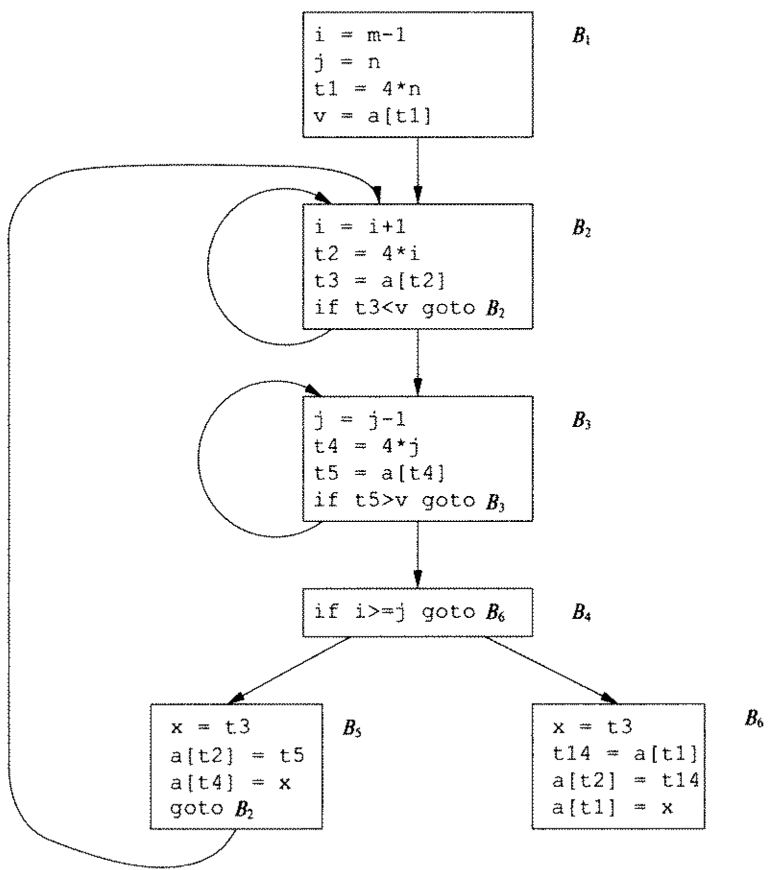


Рис. 9.5. B_5 и B_6 после устранения общих подвыражений

После устранения локальных общих подвыражений B_5 все равно вычисляет $4*i$ и $4*j$, как показано на рис. 9.4, *б*. Оба они являются общими подвыражениями; в частности, инструкции

```
t8 = 4*j
t9 = a[t8]
a[t8] = x
```

в B_5 могут быть заменены инструкциями

```
t9 = a[t4]
a[t4] = x
```

с использованием значения $t4$, вычисленного в блоке B_3. На рис. 9.5 видно, что при переходе управления от вычисления $4 * j$ в блоке B_3 к блоку B_5 значение j не изменяется, как не изменяется и $t4$, так что значение этой переменной можно использовать вместо $4 * j$.

Еще одно общее подвыражение появляется в B_5 после того, как $t4$ заменяет $t8$. Новое выражение $a\,[t4]$ соответствует значению $a\,[j]$ на уровне исходного текста. При передаче управления из блока B_3 в блок B_5 свое значение сохраняет не только j, но и $a\,[j]$, значение, вычисленное в переменную $t5$, поскольку в промежутке нет никаких присваиваний элементам массива a. Инструкции

```
t9 = a[t4]
a[t6] = t9
```

в B_5, таким образом, могут быть заменены инструкцией

```
a[t6] = t5
```

Аналогично значение, присвоенное x в блоке B_5 на рис. 9.4, *б*, как видно, то же, что и значение, присвоенное $t3$ в блоке B_2. Блок B_5 на рис. 9.5 представляет собой результат устранения общих подвыражений, соответствующих на уровне исходного текста значениям $a\,[i]$ и $a\,[j]$, из блока B_5 на рис. 9.4, *б*. Аналогичная серия преобразований выполнена и в блоке B_6 на рис. 9.5.

Выражения $a\,[t1]$ в блоках B_1 и B_6 на рис. 9.5 *не* являются общими подвыражениями, хотя в обоих случаях может использоваться $t1$. После того как управление покидает B_1 и перед тем как оно достигает B_6, оно может пройти через блок B_5, где имеются присваивания массиву a. Следовательно, $a\,[t1]$ может не иметь то же значение при достижении B_6, что и при покидании B_1, так что рассматривать $a\,[t1]$ как общее подвыражение небезопасно. □

9.1.5 Распространение копий

Блок B_5 на рис. 9.5 может быть улучшен удалением x путем применения двух новых преобразований. Одно из них связано с присваиванием вида `u = v`, именуемого *инструкцией копирования* или для краткости — просто *копированием*. Если пристально рассмотреть пример 9.2, мы поймем, что копирования становятся более частыми хотя бы потому, что алгоритм устранения общих подвыражений, как и ряд других алгоритмов, вносит в код новые инструкции копирования.

Пример 9.3. На рис. 9.6, *а* при удалении общего подвыражения в `c = d+e` используется новая переменная t для хранения значения $d+e$. Значение переменной t присваивается переменной c вместо выражения $d + e$ на рис. 9.6, *б*. Поскольку

управление может достичь выражения `c = d+e` как после присваивания переменной a, так и после присваивания b, было бы некорректно заменять выражение `c = d+e` присваиванием `c = a` или `c = b`. □

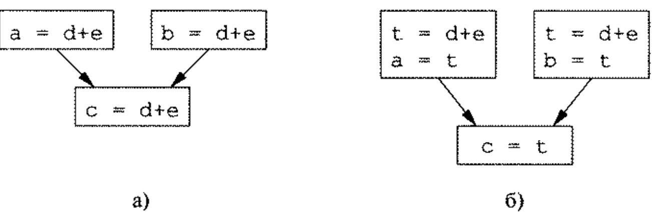


Рис. 9.6. Копирования, добавленные в процессе устранения общих подвыражений

Идея, лежащая в основе преобразования распространения копий, заключается в применении v вместо u везде, где это возможно, после инструкции копирования `u = v`. Например, присваивание `x = t3` в блоке B_5 на рис. 9.5 является копированием. Применение распространения копий к B_5 дает в результате код, показанный на рис. 9.7. Такое преобразование может показаться никаким не улучшением кода, но, как мы увидим в разделе 9.1.6, оно дает нам возможность удалить присваивание x.

```
x = t3
a[t2] = t5
a[t4] = t3
goto B2
```

Рис. 9.7. Базовый блок B_5 после распространения копий

9.1.6 Удаление бесполезного кода

Переменная в некоторой точке программы считается *живой* (live), или активной, если ее значение будет использовано в программе в последующем; в противном случае она считается *мертвой* (dead). Очередная идея оптимизации касается мертвого, или бесполезного кода, т.е. кода, вычисляющего значения, которые никогда не будут использованы. Хотя программист вряд ли будет создавать такой код умышленно, он может возникнуть в результате предшествующих преобразований.

Пример 9.4. Предположим, что переменная `debug` может принимать значения `TRUE` или `FALSE` в различных точках программы и используется в инструкциях типа

```
if (debug) print ...
```

Может оказаться, что компилятор сможет выяснить, что всякий раз по достижении этого кода `debug` принимает значение `FALSE` независимо от реальной последовательности ветвления программы. Если преобразование распространения копий заменяет `debug` значением `FALSE`, инструкция `print` оказывается мертвой, так как не может быть достигнута. В результате мы можем удалить и проверку, и инструкцию вывода из объектного кода. В общем случае выяснение в процессе компиляции того факта, что значение выражения – константа, и использование вместо выражения этой константы называется *дублированием констант* (constant folding). □

Одно из преимуществ распространения копий состоит именно в том, что зачастую оно превращает инструкцию копирования в мертвый код. Например, распространение копий с последующим удалением бесполезного кода приводит к устранению присвоения x и преобразованию кода на рис. 9.7 в код

```
a[t2] = t5
a[t4] = t3
goto B2
```

Он представляет собой дальнейшее улучшение блока B_5 на рис. 9.5.

9.1.7 Перемещение кода

Циклы являются очень важным объектом приложения усилий оптимизации, в особенности внутренние циклы, на выполнение которых тратится основное время работы программы. Это время может быть уменьшено, если уменьшить количество инструкций во внутреннем цикле, даже ценой увеличения кода вне его.

Важным изменением, уменьшающим количество кода в цикле, является *перемещение кода*. Это преобразование берет из цикла выражение, приводящее к одному и тому же результату независимо от того, сколько раз выполняется цикл (*вычисление, инвариантное относительно цикла* (loop-invariant computation)), и размещает его перед циклом. Заметим, что понятие "перед циклом" предполагает существование входа в цикл, т.е. базового блока, в который ведут все переходы извне цикла (см. раздел 8.4.5).

Пример 9.5. Например, вычисление *limit* – 2 является инвариантом цикла в следующей конструкции:

```
while(i <= limit - 2) /* Инструкции, не изменяющие limit */
```

Перемещение кода преобразует эту инструкцию в эквивалентный код

```
t = limit - 2;
while(i <= t) /* Инструкции, не изменяющие limit или t */
```

Теперь вычисление *limit* − 2 выполняется однократно, до входа в цикл. Ранее при выполнении n итераций цикла требовалось $n + 1$ раз вычислить значение *limit* − 2. □

9.1.8 Переменные индукции и снижение стоимости

Другая важная оптимизация состоит в поиске переменных индукции в циклах и оптимизации их вычислений. Переменная x называется "переменной индукции", если существует положительная или отрицательная константа c, такая, что всякий раз при присваивании x ее значение увеличивается на c. Например, в цикле, содержащем B_2 на рис. 9.5, переменными индукции являются i и $t2$. Переменные индукции могут быть вычислены при помощи единственного увеличения (сложения или вычитания) в каждой итерации цикла. Преобразование *снижения стоимости* (strength reduction) состоит в замене дорогой операции, такой как умножение, более дешевой, такой как сложение. Однако переменная индукции не только позволяет нам выполнить снижение стоимости; зачастую можно обойтись только одной переменной из группы переменных индукции, значения которых остаются неизменными в пределах итерации цикла.

При работе с циклами лучше работать "изнутри наружу", т.е. начинать со внутренних циклов и последовательно двигаться ко все большим охватывающим циклам. Посмотрим, как можно применить указанную оптимизацию к примеру быстрой сортировки, начиная с одного из вложенных циклов, а именно — состоящего из одного блока B_3. Заметим, что переменные j и $t4$ остаются неизменными в пределах итерации; всякий раз значение j уменьшается на 1, а значение $t4$ уменьшается на 4, поскольку этой переменной присваивается значение $4 * j$. Данные переменные j и $t4$ образуют хороший пример пары переменных индукции.

При наличии двух или более переменных индукции в цикле можно отбросить все, кроме одной. Во внутреннем цикле B_3 мы не можем полностью отказаться от j или $t4$: переменная $t4$ используется в блоке B_3, а переменная j — в блоке B_4. Однако можно проиллюстрировать снижение стоимости и часть процесса устранения переменных индукции. В конечном счете переменная j будет удалена при рассмотрении внешнего цикла, состоящего из базовых блоков B_2, B_3, B_4 и B_5.

Пример 9.6. Поскольку очевидно, что после присваивания переменной $t4$ на рис. 9.5 выполняется соотношение $t4 = 4 * j$ и переменная $t4$ не изменяется в цикле, состоящем из блока B_3, сразу после инструкции `j = j-1` должно выполняться соотношение $t4 = 4 * j + 4$. Следовательно, можно заменить присваивание `t4 = 4*j` на `t4 = t4-4`. Единственная проблема в том, что при первом входе в блок B_3 переменная $t4$ не содержит значения.

Поскольку при входе в базовый блок B_3 должно выполняться соотношение $t4 = 4*j$, поместим инициализацию переменной $t4$ в конце блока, где выполняется

инициализация j, что показано на рис. 9.8 дополнением к блоку B_1, выделенным пунктиром. Хотя мы и добавили одну команду, которая однократно выполняется в блоке B_1, замена умножения вычитанием ускорит объектный код, если умножение требует больше времени, чем сложение и вычитание, что справедливо для множества машин. □

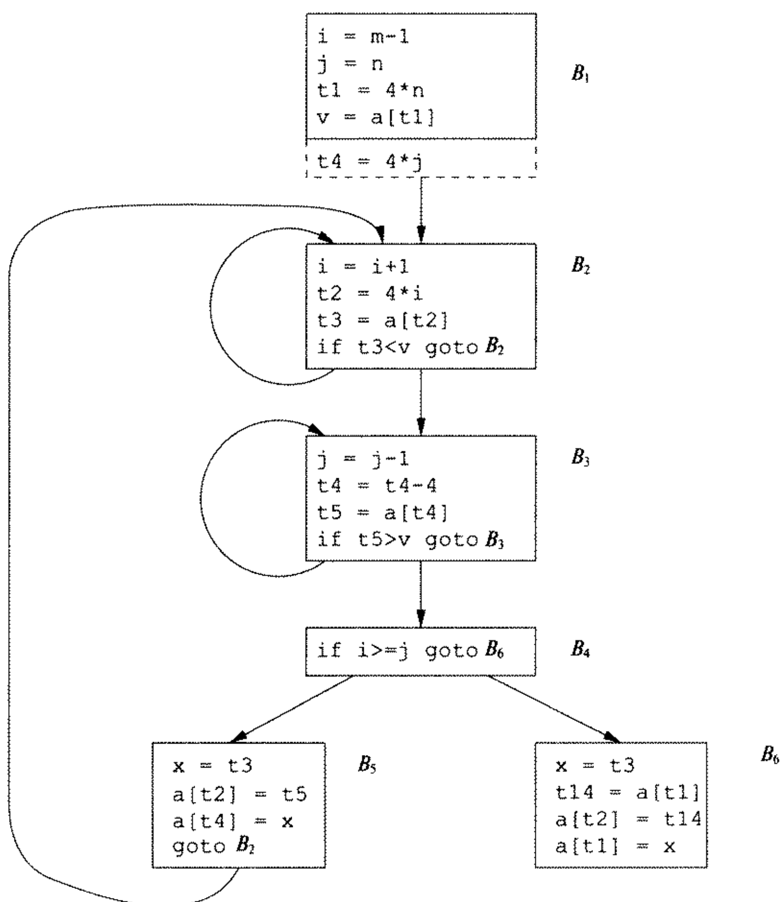


Рис. 9.8. Снижение стоимости, примененное к $4 * j$ в базовом блоке B_3

Завершим этот раздел еще одним примером — удаления переменной индукции. В этом примере i и j рассматриваются в контексте внешнего цикла, состоящего из блоков B_2, B_3, B_4 и B_5.

Пример 9.7. После снижения стоимости, примененного ко внутренним циклам B_2 и B_3, единственное использование i и j находится в блоке B_4, где сравниваются значения этих переменных. Мы знаем, что значения i и $t2$ удовлетворяют соот-

ношению $t2 = 4 * i$, а значения j и $t4$ — соотношению $t4 = 4 * j$. Таким образом, вместо проверки $i \geqslant j$ может использоваться проверка $t2 \geqslant t4$. После выполнения этой замены i в блоке B_2 и j в блоке B_3 становятся мертвыми переменными, а присваивание им в этих блоках — мертвым кодом, который может быть удален. Получающийся в результате граф потока показан на рис. 9.9. □

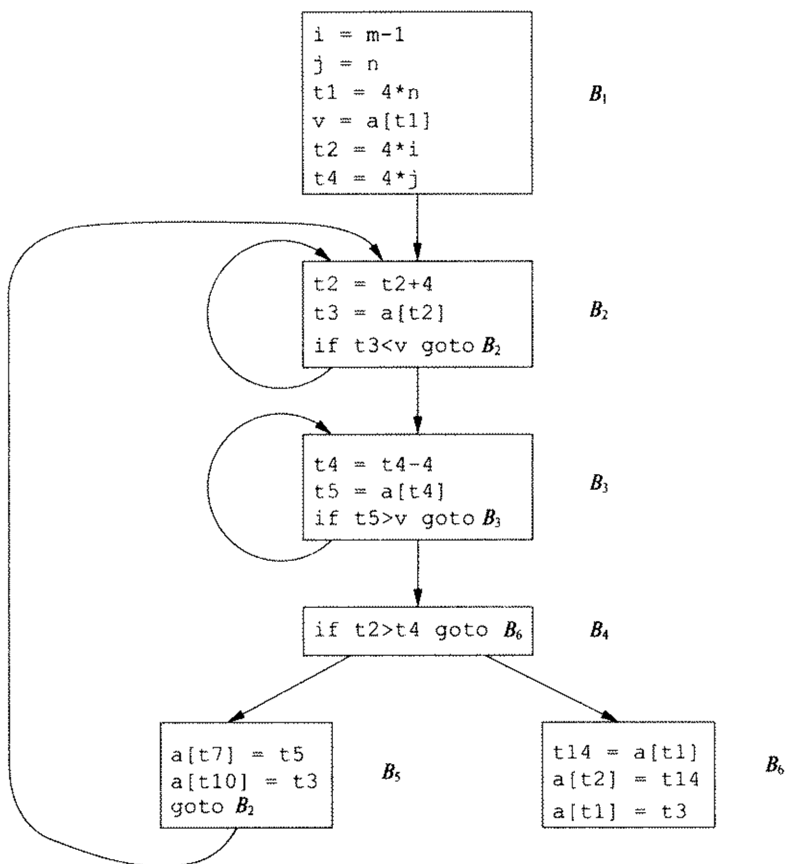


Рис. 9.9. Граф потока после устранения переменных индукции

Рассмотренные нами преобразования кода оказались весьма эффективными. На рис. 9.9 количество команд в базовых блоках B_2 и B_3 снизилось с 4 до 3 по сравнению с исходным графом потока на рис. 9.3. В базовом блоке B_5 количество команд сократилось с 9 до 3, а в базовом блоке B_6 — с 8 до 3. Правда, блок B_1 при этом вырос с 4 команд до 6, но он выполняется в данном фрагменте только один раз, так что на общее время выполнения фрагмента размер блока B_1 влияет очень мало.

9.1.9 Упражнения к разделу 9.1

Упражнение 9.1.1. Для графа потока, представленного на рис. 9.10, выполните следующее.

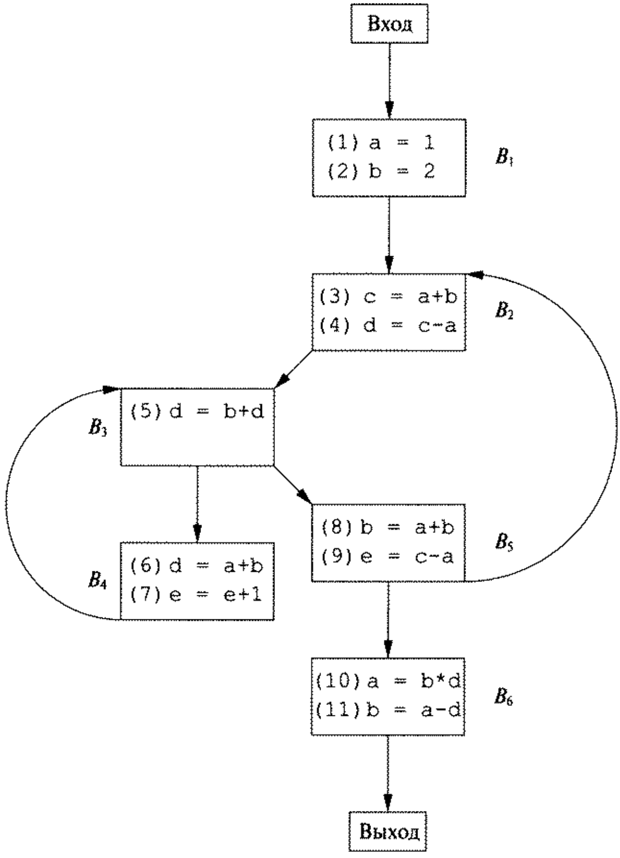


Рис. 9.10. Граф потока к упражнению 9.1.1

а) Найдите циклы в этом графе потока.

б) Инструкции 1 и 2 в базовом блоке B_1 представляют собой инструкции копирования, в которых a и b получают константные значения. Для каких использований a и b можно выполнить размножение копирований и заменить эти использования переменных использованиями констант? Выполните такую замену везде, где это возможно.

в) Найдите глобальные общие подвыражения в каждом цикле.

г) Найдите все переменные индукции для каждого цикла. Убедитесь, что вы учли все константы, внесенные в п. *б*.

д) Найдите для каждого цикла вычисления, инвариантные относительно этого цикла.

Упражнение 9.1.2. Примените преобразования из этого раздела к графу потока на рис. 8.9.

Упражнение 9.1.3. Примените преобразования из этого раздела к графу потока из *а*) упражнения 8.4.1; *б*) упражнения 8.4.2.

Упражнение 9.1.4. На рис. 9.11 приведен промежуточный код для вычисления скалярного произведения векторов A и B. Оптимизируйте насколько возможно этот код путем устранения общих подвыражений, снижения стоимости и устранения переменных индукции.

```
    dp = 0.
    i = 0
L:  t1 = i*8
    t2 = A[t1]
    t3 = i*8
    t4 = B[t3]
    t5 = t2*t4
    dp = dp+t5
    i = i+1
    if i<n goto L
```

Рис. 9.11. Промежуточный код вычисления скалярного произведения

9.2 Введение в анализ потоков данных

Все оптимизации, рассматривавшиеся в разделе 9.1, зависят от *анализа потоков данных*. Анализ потоков данных (data-flow analysis) означает методы, собирающие информацию о потоках данных вдоль путей выполнения программы. Например, один из способов реализации устранения глобальных общих подвыражений требует от нас определить, вычисляют ли два текстуально одинаковых выражения одно и то же значение при любом из возможных путей выполнения программы. В качестве другого примера, если результат присваивания не используется ни в одном из последующих путей выполнения программы, это присваивание можно удалить как мертвый код. Анализ потоков данных позволяет найти ответы на перечисленные и многие другие вопросы.

9.2.1 Абстракция потока данных

Как было сказано в разделе 1.6.2, выполнение программы может рассматриваться как ряд преобразований состояния программы, которое состоит из значений всех переменных программы, включая переменные в кадрах стека, находящихся ниже текущей вершины стека времени выполнения. Каждое выполнение инструкции промежуточного кода преобразует входное состояние программы в новое выходное состояние. Входное состояние связано с *точкой программы перед* инструкцией, а выходное — с *точкой программы после* инструкции.

При анализе поведения программы мы должны рассматривать все возможные последовательности точек программы ("пути") в графе потока, через которые может проходить программа. Затем из возможных состояний программы в каждой точке извлекается информация, необходимая для решения поставленных нами задач анализа потока данных. При более сложном анализе следует учитывать пути с переходами при вызовах и возвратах из процедур. Однако для начала мы ограничимся путями в одном графе потока для единственной процедуры.

Итак, что же нам может рассказать граф потока о возможных путях выполнения?

- В пределах одного базового блока точка программы после инструкции совпадает с точкой программы перед следующей инструкцией.
- Если имеется дуга от базового блока B_1 к базовому блоку B_2, то за точкой программы после последней инструкции B_1 может непосредственно следовать точка программы перед первой инструкцией B_2.

Таким образом, мы можем определить *путь выполнения* (execution path), или просто *путь*, от точки p_1 к точке p_n как последовательность точек $p_1, p_2, \ldots, p_n$, таких, что для каждого $i = 1, 2, \ldots, n-1$ либо

1. p_i является точкой, непосредственно предшествующей инструкции, а p_{i+1} — точкой, непосредственно следующей за этой инструкцией в том же блоке, либо
2. p_i представляет собой конец некоторого блока, а p_{i+1} — начало следующего блока.

В общем случае существует бесконечное число возможных путей выполнения программы и не существует верхней границы длины пути выполнения. Анализ программы подводит итог всем возможным состояниям, которые могут существовать в точке программы, на основании конечного множества фактов. Различные анализы могут выбирать разную информацию, и в общем случае ни один анализ не требует идеального представления состояния программы.

Пример 9.8. Даже простая программа на рис. 9.12 описывает неограниченное количество путей выполнения. Кратчайший путь не входит в цикл и состоит из точек программы $(1, 2, 3, 4, 9)$. Следующий по длине путь проходит одну итерацию цикла и состоит из точек $(1, 2, 3, 4, 5, 6, 7, 8, 3, 4, 9)$. Мы знаем, что, например, при первом прохождении точки (5) значение a равно 1 в соответствии с определением d_1. Мы говорим, что d_1 *достигает* точки (5) при первой итерации. В последующих итерациях точки (5) достигает d_3, и значение a становится равным 243.

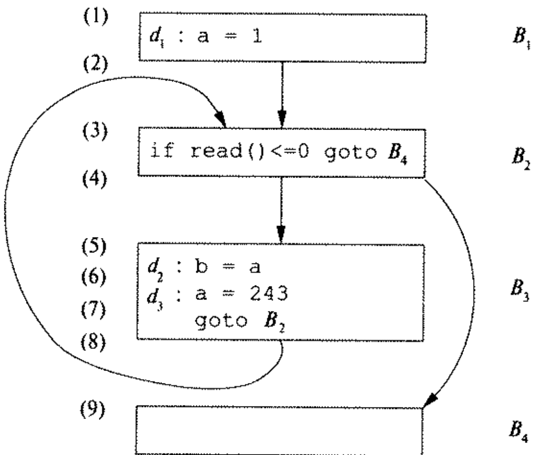


Рис. 9.12. Пример программы, иллюстрирующей абстракцию потока данных

В общем случае отследить все состояния программы для всех возможных путей невозможно. В анализе потока данных мы не выделяем пути, достигающие данной точки. Более того, мы не отслеживаем состояния полностью; вместо этого мы абстрагируемся от определенных деталей, отслеживая только данные, необходимые для проведения нашего анализа. Два примера иллюстрируют, как одни и те же состояния программы могут привести к разной информации, абстрагированной в точке программы.

1. Чтобы помочь пользователям в отладке их программ, мы можем решить отслеживать все возможные значения переменной в некоторой точке программы и в каких именно местах программы эти значения могут быть определены. Например, мы можем подвести итоги всех состояний программы в точке (5), сказав, что возможные значения переменной a в этой точке — $\{1, 243\}$ и что она может быть определена в $\{d_1, d_3\}$. Определения, которые могут *достичь* точки программы по некоторому пути, известны как *достигающие определения*.

2. Предположим, что теперь нас интересует реализация дублирования констант. Если использование переменной x достигается только одним определением и это определение присваивает x константу, то можно просто заменить x этой константой. Если же одной точки программы могут достичь несколько определений, то мы не можем прибегнуть к дублированию констант. Таким образом, при дублировании констант нас интересуют те определения, для которых данной точки программы достигает только одно-единственное определение, независимо от того, какой путь выполнения рассматривается. Для точки (5) на рис. 9.12 не существует определения, которое *должно* быть определением a в этой точке, так что для a в точке (5) это множество определений пустое. Даже если переменная имеет единственное определение в интересующей нас точке, это определение должно присваивать переменной константу. Таким образом, можно просто описать некоторые переменные как "не константы" вместо сбора информации об их возможных значениях или всех возможных определениях.

Как видите, одна и та же информация может быть подытожена различными способами в зависимости от цели анализа. □

9.2.2 Схема анализа потока данных

В каждом приложении анализа потока данных мы связываем с каждой точкой программы *значение потока данных* (data-flow value), которое представляет абстракцию множества всех возможных состояний программы, которые могут наблюдаться в данной точке. Множество возможных значений потока данных является *областью определения* (domain) этого приложения. Например, областью определения для достигающих определений является множество всех подмножеств определений в программе. Конкретное значение потока данных представляет собой множество определений, и мы хотим связать с каждой точкой программы точное множество определений, которые могут достигать данной точки. Как говорилось выше, выбор абстракции зависит от цели анализа; для эффективности мы отслеживаем только ту информацию, которая имеет отношение к решаемой нами задаче.

Обозначим значения потока данных до и после каждой инструкции s как $\text{IN}[s]$ и $\text{OUT}[s]$ соответственно. *Задача потока данных* (data-flow problem) состоит в поиске решения для множества ограничений, накладываемых на $\text{IN}[s]$ и $\text{OUT}[s]$ для всех инструкций s. Существует два вида ограничений: основанные на семантике инструкций (передаточные функции) и основанные на потоке управления.

Передаточные функции

Значения потока данных перед инструкцией и после нее ограничены семантикой этой инструкции. Предположим, например, что наш анализ потока данных

включает определение константного значения переменных в точках. Если переменная a имеет значение v перед выполнением инструкции b = a, то и a, и b после присваивания имеют значение v. Такое соотношение между значениями потока данных до и после инструкции присваивания известно как *передаточная функция* (transfer function).

Передаточные функции работают в двух направлениях: информация может распространяться как в прямом направлении вдоль пути выполнения, так и в обратном. В задаче прямого потока передаточная функция инструкции s, которая обычно обозначается как f_s, получает значение потока данных перед инструкцией и выдает новое значение потока данных после инструкции, т.е.

$$\text{OUT}[s] = f_s(\text{IN}[s])$$

И наоборот, в задаче обратного потока передаточная функция f_s для инструкции s преобразует значение потока данных после инструкции в новое значение потока данных до инструкции, т.е.

$$\text{IN}[s] = f_s(\text{OUT}[s])$$

Ограничения потока управления

Второе множество ограничений значений потоков данных порождается потоком управления. В базовом блоке поток управления очень простой. Если базовый блок B состоит из инструкций $s_1, s_2, \dots, s_n$ в указанном порядке, то значение потока управления на выходе из s_i то же, что и значение потока управления на входе в s_{i+1}, т.е.

$$\text{IN}[s_{i+1}] = \text{OUT}[s_i] \quad \text{для всех } i = 1, 2, \dots, n-1$$

Однако ребра потока управления между базовыми блоками приводят к более сложным ограничениям между последней инструкцией одного базового блока и первой инструкцией следующего. Например, если нас интересует сбор всех определений, которые могут достигать точки программы, то множество определений, достигающих лидера базового блока, представляет собой объединение определений после последних инструкций каждого из предшественников данного блока. В следующем разделе потоки данных между блоками рассматриваются более детально.

9.2.3 Схемы потоков данных в базовых блоках

При рассмотрении потоков данных в базовых блоках можно сэкономить время и память, поскольку поток управления проходит по базовому блоку без прерываний и ветвлений. Таким образом, можно переформулировать схему в терминах

значений потока данных при входе в блок и выходе из него. Обозначим значения потока данных непосредственно перед и непосредственно после каждого базового блока B как $\text{IN}[B]$ и $\text{OUT}[B]$ соответственно. Ограничения для $\text{IN}[B]$ и $\text{OUT}[B]$ можно вывести из ограничений на $\text{IN}[s]$ и $\text{OUT}[s]$ для различных инструкций s в базовом блоке B следующим образом.

Предположим, что блок B состоит из инструкций $s_1, \ldots, s_n$ в указанном порядке. Если s_1 — первая инструкция базового блока B, то $\text{IN}[B] = \text{IN}[s_1]$. Аналогично если s_n — последняя инструкция базового блока B, то $\text{OUT}[B] = \text{OUT}[s_n]$. Передаточная функция базового блока B, которую мы обозначим как f_B, может быть получена как композиция передаточных функций инструкций базового блока. Пусть f_{s_i} — передаточная функция для инструкции s_i. Тогда $f_B = f_{s_n} \circ \cdots \circ f_{s_2} \circ f_{s_1}$. Соотношение между началом блока и его концом имеет вид

$$\text{OUT}[B] = f_B(\text{IN}[B])$$

Ограничения, накладываемые потоком управления между базовыми блоками, можно легко переписать, заменяя $\text{IN}[B]$ и $\text{OUT}[B]$ на $\text{IN}[s_1]$ и $\text{OUT}[s_n]$ соответственно. Например, если значения потока данных представляют собой информацию о множествах констант, которые *могут* быть присвоены переменным, то мы получаем задачу прямого потока, в которой

$$\text{IN}[B] = \bigcup_{P-\text{предшественник } B} \text{OUT}[P]$$

При обратном потоке — например, как мы вскоре увидим, в случае анализа живых переменных — уравнения аналогичны, но IN и OUT меняются местами, т.е.

$$\text{IN}[B] = f_B(\text{OUT}[B])$$
$$\text{OUT}[B] = \bigcup_{S-\text{преемник } B} \text{IN}[S]$$

В отличие от линейных арифметических уравнений, уравнения потоков данных обычно не имеют единственного решения. Наша цель заключается в том, чтобы найти наиболее "точное" решение, которое удовлетворяет двум множествам ограничений: ограничениям потока управления и ограничениям передачи. Иначе говоря, нам нужно решение, которое приводит к корректному улучшению кода и не допускает небезопасных преобразований, которые могут изменить результат вычислений компьютера. Этот вопрос вкратце рассматривается во врезке "Консерватизм анализа потоков данных" и более подробно — в разделе 9.3.4. В следующих подразделах мы рассмотрим некоторые из наиболее важных примеров задач, которые могут быть решены с помощью анализа потоков данных.

Выявление возможных использований до определения

Вот как мы используем решение задачи достигающих определений для обнаружения использований переменных до их определения. Фокус заключается во введении фиктивного определения для каждой переменной x на входе в граф потока. Если фиктивное определение x достигает точки p, где x может быть использовано, то возможно использование этой переменной до ее определения. Заметим, что никогда нельзя быть абсолютно уверенным, что в программе имеется ошибка, поскольку могут быть причины (возможно, со сложным логическим доказательством), по которым путь p, на котором нет реального определения x, никогда не будет пройден.

9.2.4 Достигающие определения

Достигающие определения (reaching definitions) — одна из наиболее распространенных и полезных схем потока данных. Зная, где именно в программе может быть определена каждая переменная x при достижении потоком управления каждой точки p, можно получить много информации об этой переменной. В частности, компилятор может выяснить, является ли x константой в точке p, а отладчик может сообщить о возможном использовании в точке p не инициализированной переменной x.

Мы говорим, что определение d *достигает* точки p, если существует путь от точки, непосредственно следующей за d, к точке p, такой, что d не уничтожается вдоль этого пути. Мы *уничтожаем* (kill) определение переменной x, если существует иное определение x где-то вдоль пути[3]. Интуитивно понятно, что, если определение d некоторой переменной x достигает точки p, то d может быть местом, где последний раз определяется значение x, используемое в p.

Определением переменной x является инструкция, которая присваивает или может присваивать значение переменной x. Параметры процедур, обращения к массивам и косвенные обращения могут использовать псевдонимы, так что не так легко сказать, обращается ли некоторая инструкция к конкретной переменной x. Анализ программы должен быть консервативным: если мы не знаем, присваивает ли инструкция s значение переменной x, то мы должны считать, что она *может* сделать это, т.е. что переменная x после инструкции s может иметь либо исходное значение, бывшее у нее до инструкции s, либо новое значение, созданное s. Для простоты в оставшейся части главы предполагается, что мы работаем только с переменными, не имеющими псевдонимов. Этот класс пере-

[3]Заметим, что путь может иметь циклы, так что можно прийти к другому определению d вдоль пути, который не уничтожает d.

менных включает все локальные скалярные переменные в большинстве языков программирования; в случае C и C++ исключаются локальные переменные, адреса которых были вычислены в некоторой точке программы.

Пример 9.9. На рис. 9.13 показан граф потока с семью определениями. Нас интересуют определения, достигающие блока B_2. Все определения в блоке B_1 достигают начала блока B_2. Определение $d_5: \ j = j - 1$ в блоке B_2 также достигает начала блока B_2, поскольку других определений j в цикле, приводящем к началу блока B_2, нет. Однако это определение уничтожает определение $d_2: \ j = n$, не позволяя ему достичь блоков B_3 или B_4. Инструкция $d_4: \ i = i + 1$ в B_2 не достигает начала B_2, поскольку переменная i всегда переопределяется в $d_7: \ i = u3$. Наконец, определение $d_6: \ a = u2$ также достигает начала блока B_2. □

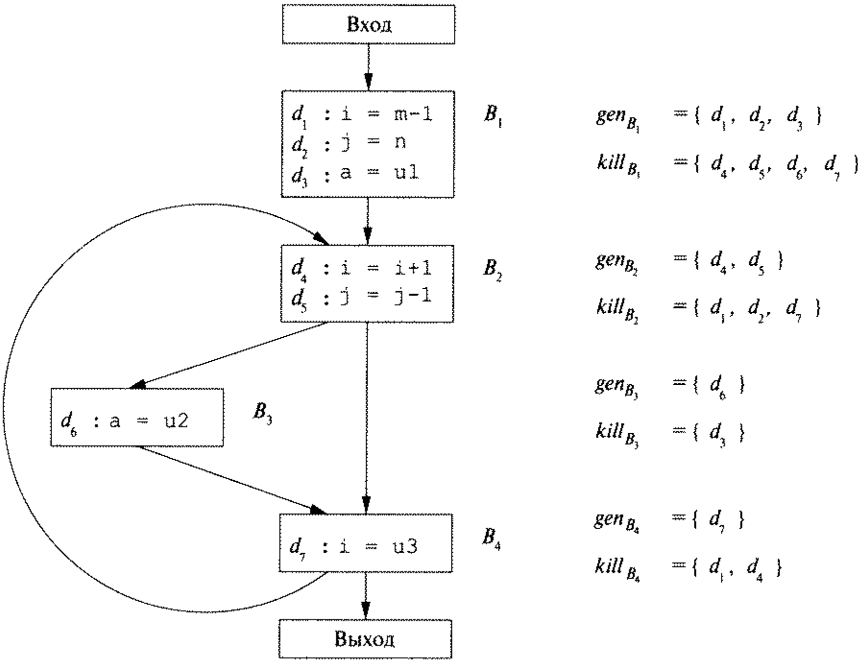


Рис. 9.13. Граф потока для иллюстрации достигающих определений

При используемом нами определении достигающих определений иногда допускаются неточности. Однако все они — в безопасном направлении. Например, вспомним о нашем предположении, что могут быть обойдены все ребра графа потока. На практике это предположение может оказаться неверным. Например, ни для каких значений a и b поток управления не сможет достичь инструкции 2 в приведенном далее фрагменте:

```
if (a == b) инструкция1; else if (a == b) инструкция2;
```

Консерватизм анализа потоков данных

Поскольку все схемы потоков данных вычисляют всего лишь приближения (определяемые всеми возможными путями выполнения программы), следует гарантировать, что любые ошибки будут допущены только в "безопасном направлении". Стратегия принятия решений *безопасна* (или *консервативна*), если она не позволяет нам изменить результаты вычисления программы. Такая безопасная стратегия может, к сожалению, не допустить некоторых улучшений кода программы, которые оставили бы неизменным ее смысл, но в конечном счете безопасной стратегии, которая разрешала бы все допустимые оптимизации кода, не запрещая ни одной из них, не существует. Применение небезопасных стратегий, ускоряющих работу программы ценой ее ошибочности, совершенно неприемлемо.

Таким образом, при разработке схемы потока данных следует продумывать, как именно будет использована полученная информация, и гарантировать, что все приближения будут делаться в "консервативном", или "безопасном", направлении. Каждая схема и ее применение должны рассматриваться отдельно. Например, при использовании достигающих определений для дублирования констант безопасно считать, что определение является достигающим, в то время как оно не является таковым (мы можем считать, что x не является константой, в то время как на самом деле можно выполнить дублирование константы), но считать, что определение не является достигающим, в то время как на самом деле это не так, — опасно (мы можем таким образом заменить x константой, в то время как на самом деле значение x при выполнении программы может принимать значения, отличные от этой константы).

В общем виде задача определения возможности прохода для каждого пути графа потока неразрешима. Таким образом, мы просто предполагаем, что каждый путь графа потока при каком-то из выполнений программы может быть пройден. В применении к определению достижимости консервативное решение состоит в предположении, что определение может достичь некоторой точки, даже если оно ее не достигает. Таким образом, мы допускаем наличие путей, которые не могут быть пройдены при выполнении программы, и разрешаем определениям проходить через неоднозначные определения той же переменной.

Уравнения передачи для достигающих определений

Теперь займемся ограничениями для задачи достигающих определений. Начнем с детального изучения единственной инструкции. Рассмотрим определение

$$d:\ \texttt{u = v+w}$$

Здесь, как и в большинстве случаев далее, + обозначает обобщенный бинарный оператор.

Эта инструкция "генерирует" определение d переменной u и "уничтожает" все другие определения этой переменной в программе. Передаточная функция определения d может быть записана как

$$f_d(x) = gen_d \cup (x - kill_d) \tag{9.1}$$

Здесь $gen_d = \{d\}$ — множество определений, генерируемых инструкцией, а $kill_d$ — множество всех прочих определений u в программе.

Как говорилось в разделе 9.2.2, передаточная функция базового блока может быть получена путем композиции передаточных функций содержащихся в нем инструкций. Композиция функций вида (9.1), о котором мы будем говорить как о "виде (или форме) *gen-kill*", имеет тот же вид, что легко увидеть. Предположим, что имеются функции $f_1(x) = gen_1 \cup (x - kill_1)$ и $f_2(x) = gen_2 \cup (x - kill_2)$. Тогда

$$\begin{aligned} f_2(f_1(x)) &= gen_2 \cup (gen_1 \cup (x - kill_1) - kill_2) = \\ &= (gen_2 \cup (gen_1 - kill_2)) \cup (x - (kill_1 \cup kill_2)) \end{aligned}$$

Это правило распространяется на базовый блок, состоящий из произвольного количества инструкций. Предположим, что блок B имеет n инструкций, передаточные функции которых $f_i = gen_i \cup (x - kill_i)$ для $i = 1, 2, \ldots, n$. Тогда передаточная функция для базового блока B может быть записана как

$$f_B(x) = gen_B \cup (x - kill_B),$$

где

$$kill_B = kill_1 \cup kill_2 \cup \cdots \cup kill_n$$

и

$$\begin{aligned} gen_B = gen_n &\cup (gen_{n-1} - kill_n) \cup (gen_{n-2} - kill_{n-1} - kill_n) \cup \\ &\cup \cdots \cup (gen_1 - kill_2 - kill_3 - \cdots - kill_n) \end{aligned}$$

Таким образом, базовый блок, как и инструкция, с одной стороны, генерирует множество определений, с другой — уничтожает множество определений. Множество *gen* содержит все определения внутри блока, которые "видимы" непосредственно после блока; будем называть их *нижнепредставленными* (downwards

exposed). Определение нижнепредставлено в базовом блоке только в том случае, если оно не уничтожено последующим определением той же переменной в этом же базовом блоке. Множество *kill* базового блока представляет собой простое объединение всех определений, уничтоженных отдельными инструкциями. Обратите внимание, что определение может присутствовать как в множестве *gen* базового блока, так и в его множестве *kill*. Если это так, то больший приоритет имеет множество *gen*, поскольку в форме *gen-kill* множество *kill* применяется до множества *gen*.

Пример 9.10. Множество *gen* для базового блока

$$d_1 : \texttt{a} = 3$$
$$d_2 : \texttt{a} = 4$$

представляет собой $\{d_2\}$, поскольку d_1 не является нижнепредставленным определением. Множество *kill* содержит как d_1, так и d_2, поскольку d_1 уничтожает d_2 и наоборот. Тем не менее, поскольку вычитание множества *kill* предшествует операции объединения с множеством *gen*, конечная передаточная функция для этого блока будет включать определение d_2. □

Уравнения потока управления

Теперь рассмотрим множество ограничений, порождаемых потоком управления между базовыми блоками. Поскольку определение достигает точки программы, если существует по крайней мере один путь, вдоль которого эта точка может быть достигнута, $\text{OUT}[P] \subseteq \text{IN}[B]$ везде, где имеется ребро потока управления от P к B. Однако, поскольку определение не может достичь точки, если нет пути для ее достижения, $\text{IN}[B]$ должно быть не больше, чем объединение достигающих определений всех предшествующих базовых блоков. Иными словами, можно безопасно считать, что

$$\text{IN}[B] = \bigcup_{P-\text{предшественник } B} \text{OUT}[P]$$

Будем называть объединение *оператором сбора* (meet operator) для достигающих определений. Этот оператор используется для суммирования вкладов различных путей при их слиянии в любой схеме потока данных.

Итеративный алгоритм для достигающих определений

Мы считаем, что любой граф потока содержит два пустых блока — входной блок, являющийся стартовой точкой графа, и выходной блок, через который проходят все выходы из графа. Поскольку начала графа не достигают никакие определения, передаточная функция для входного блока представляет собой простую константную функцию, возвращающую $\varnothing$, т.е. $\text{OUT}[\text{Вход}] = \varnothing$.

Задача достигающих определений формулируется при помощи следующих уравнений:

$$\text{OUT}[\text{Вход}] = \varnothing$$

и для всех базовых блоков B, отличных от входного,

$$\text{OUT}[B] = gen_B \cup (\text{IN}[B] - kill_B)$$

$$\text{IN}[B] = \bigcup_{P-\text{предшественник } B} \text{OUT}[P]$$

Данные уравнения могут быть решены при помощи приведенного ниже алгоритма. В результате работы этого алгоритма получается *наименьшая фиксированная точка* (least fixedpoint) уравнений, т.е. решение, значения множеств IN и OUT которого содержатся в соответствующих значениях любого другого решения этих уравнений. Этот результат приемлем, поскольку любое определение в одном из множеств IN и OUT, конечно же, должно достигать указанной точки. Это решение корректно, поскольку оно не включает никакие определения, о которых мы точно знаем, что они не являются достигающими.

Алгоритм 9.11. Достигающие определения

ВХОД: граф потока, в котором для каждого блока B вычислены множества $kill_B$ и gen_B.

ВЫХОД: $\text{IN}[B]$ и $\text{OUT}[B]$, множества достигающих определений для входа и выхода каждого блока B графа потока.

МЕТОД: мы используем итеративный подход с начальной оценкой $\text{OUT}[B] = \varnothing$ для всех B, сходящийся к требуемым значениям IN и OUT. Поскольку итерации должны продолжаться до тех пор, пока не сойдутся множества IN (а следовательно, и OUT), мы можем использовать логическую переменную *change* в качестве индикатора, были ли внесены при данном проходе изменения в какое-либо из множеств OUT. Однако как в этом, так и в аналогичных алгоритмах, которые будут рассмотрены ниже, мы полагаем, что механизм отслеживания изменений очевиден, и не рассматриваем его.

Набросок алгоритма приведен на рис. 9.14. Первые две строки инициализируют некоторые значения потока данных[4]. В строке 3 начинается цикл, итерации которого выполняются до схождения, а внутренний цикл в строках 4–6 применяет уравнения потока данных к каждому блоку, отличному от входного. □

Интуитивно алгоритм 9.11 распространяет определения до тех пор, пока это возможно без их уничтожения, моделируя все возможные пути выполнения программы. В конечном счете этот алгоритм завершит свою работу, поскольку для

[4]Внимательный читатель заметит, что можно объединить строки 1 и 2. Однако в аналогичных алгоритмах потоков данных может оказаться необходимым инициализировать входной и выходной узлы не так, как все остальные. Поэтому мы следуем общему шаблону итеративных алгоритмов, в которых "граничные условия" наподобие строки 1 отделены от инициализации (строка 2).

```
1)  OUT [Вход] = ∅;
2)  for (каждый базовый блок B, отличный от входного) OUT [B] = ∅;
3)  while (внесены изменения в OUT)
4)        for (каждый базовый блок B, отличный от входного) {
5)              IN [B] = ∪_{P-предшественник B} OUT [P];
6)              OUT [B] = gen_B ∪ (IN [B] - kill_B);
          }
```

Рис. 9.14. Итеративный алгоритм для вычисления достигающих определений

любого B множество OUT $[B]$ никогда не уменьшается в размере; если определение попало в это множество, оно останется в нем навсегда (см. упражнение 9.2.6). Поскольку множество всех определений конечно, рано или поздно должен произойти проход цикла, при котором ни в одно из множеств OUT не будет добавлено ни одно определение, и алгоритм завершит свою работу. Поскольку, если множество OUT в данном проходе неизменно, множество IN будет неизменно при следующем проходе, работу алгоритма можно безопасно завершить по условию неизменности множества OUT. Если не изменяется ни одно множество IN, то не может измениться и ни одно множество OUT, так что все последующие итерации цикла оставят все множества IN и OUT неизменными.

Верхняя граница количества итераций в цикле равна количеству узлов в графе потока. Причина этого в том, что если определение достигает некоторой точки, то оно может сделать это по пути без циклов, а количество узлов в графе потока и есть верхняя граница количества узлов в пути без циклов. При каждой итерации цикла while каждое определение перемещается как минимум на один узел вдоль рассматриваемого пути, а зачастую и более чем на один узел — в зависимости от порядка, в котором посещаются узлы.

В действительности при правильном упорядочении блоков в цикле в строке 4, как показывают эмпирические данные, среднее количество итераций для реальных программ не превышает 5 (см. раздел 9.6.7). Поскольку множества определений могут быть представлены битовыми векторами, а операции над этими множествами могут быть реализованы при помощи логических операций над битовыми векторами, на практике алгоритм 9.11 на удивление эффективен.

Пример 9.12. Представим семь определений $d_1, d_2, \ldots, d_7$ в графе потока на рис. 9.13 битовыми векторами, в которых i-й бит слева представляет определение d_i. Объединение множеств вычисляется как логическая операция ИЛИ над соответствующими битовыми векторами. Разность двух множеств $S - T$ вычисляется как побитовая логическая операция И над вектором S и дополнением к битовому вектору T.

На рис. 9.15 показаны значения, принимаемые множествами IN и OUT в алгоритме 9.11. Начальные значения, указанные при помощи верхнего индекса 0, как, например, $\text{IN}[B]^0$, присваиваются в цикле в строке 2 на рис. 9.14. Все множества пустые, что представлено векторами 0000000. Значения на последующих итерациях алгоритма указываются с использованием верхних индексов: $\text{IN}[B]^1$ и $\text{OUT}[B]^1$ — на первой итерации, $\text{IN}[B]^2$ и $\text{OUT}[B]^2$ — на второй.

Блок B	$\text{OUT}[B]^0$	$\text{IN}[B]^1$	$\text{OUT}[B]^1$	$\text{IN}[B]^2$	$\text{OUT}[B]^2$
B_1	0000000	0000000	1110000	0000000	1110000
B_2	0000000	1110000	0011100	1110111	0011110
B_3	0000000	0011100	0001110	0011110	0001110
B_4	0000000	0011110	0010111	0011110	0010111
Выход	0000000	0010111	0010111	0010111	0010111

Рис. 9.15. Вычисление IN и OUT

Предположим, что цикл for в строках 4–6 выполняется с блоком B, принимающим значения

$$B_1, B_2, B_3, B_4, \text{Выход}$$

в указанном порядке. Когда $B = B_1$, поскольку $\text{OUT}[\text{Вход}] = \varnothing$, $\text{IN}[B_1]^1$ представляет собой пустое множество, а $\text{OUT}[B_1]^1$ равен gen_{B_1}. Это значение отличается от предыдущего значения $\text{OUT}[B_1]^0$, так что мы знаем, что на первой итерации внесены изменения (а потому будет выполнена и вторая итерация).

Затем рассмотрим $B = B_2$ и вычислим

$$\begin{aligned} \text{IN}[B_2]^1 &= \text{OUT}[B_1]^1 \cup \text{OUT}[B_4]^0 = \\ &= 1110000 + 0000000 = 1110000 \\ \text{OUT}[B_2]^1 &= gen_{B_2} \cup \left(\text{IN}[B_2]^1 - kill_{B_2}\right) = \\ &= 0001100 + (1110000 - 1100001) = 0011100 \end{aligned}$$

Результаты вычислений показаны на рис. 9.15. Например, в конце первого прохода $\text{OUT}[B_2]^1 = 0011100$, что отражает тот факт, что d_4 и d_5 генерируются в B_2, в то время как d_3 достигает начала базового блока B_2 и не уничтожается в нем.

Заметим, что после второй итерации $\text{OUT}[B_2]$ изменяется; это отражает тот факт, что d_6 также достигает начала базового блока B_2 и не уничтожается в нем. Мы не увидели этого на первой итерации, поскольку путь от d_6 к концу B_2, $B_3 \to B_4 \to B_2$, в данном порядке за одну итерацию не проходится. К тому моменту, когда на первой итерации мы выяснили, что d_6 достигает конца B_4, значения $\text{IN}[B_2]$ и $\text{OUT}[B_2]$ были уже вычислены.

После второй итерации никаких изменений в OUT не вносится, так что после третьего прохода алгоритм завершает свою работу, оставляя IN и OUT такими, какими они показаны в последних двух столбцах на рис. 9.15. □

9.2.5 Анализ активных переменных

Некоторые улучшающие код преобразования зависят от информации, вычисляемой в направлении, противоположном потоку управления программы; один такой пример мы сейчас и рассмотрим. В *анализе активных переменных* (live-variable analysis) для переменной x и точки p мы хотим выяснить, может ли значение x из точки p использоваться вдоль некоторого пути в графе потока, начинающемся в точке p. Если может, то мы говорим, что переменная x *активна* (жива) в точке p, если нет — *неактивна* (мертва).

Важное применение анализа активных переменных — при распределении регистров для базовых блоков. Этот вопрос уже поднимался в разделах 8.6 и 8.8. После того как значение вычислено в регистр и будет использоваться в блоке, его не обязательно сохранять в памяти, если это значение будет мертвым на выходе из блока. Если все регистры заняты, а нам нужен свободный регистр, то, в первую очередь, используются регистры с мертвыми значениями, поскольку эти значения не надо сохранять.

Мы определим уравнения потока данных непосредственно в терминах IN$[B]$ и OUT$[B]$, которые представляют собой множества активных переменных в точках непосредственно перед блоком B и после него соответственно. Эти уравнения могут быть также выведены путем определения передаточных функций для отдельных инструкций с последующей их композицией для получения передаточной функции блока в целом. Определим

1. def_B — множество переменных, *определенных* (т.е. получающих значения) в блоке B до любых их использований в этом блоке;

2. use_B — множество переменных, значения которых могут использоваться в блоке B до любых определений этих переменных.

Пример 9.13. Например, блок B_2 на рис. 9.13 использует i. Он также использует j до переопределения этой переменной, если только i и j не являются псевдонимами друг друга. В предположении, что среди переменных на рис. 9.13 псевдонимов нет, $use_{B_2} = \{i, j\}$. Кроме того, блок B_2 определяет i и j. В том же предположении отсутствия псевдонимов $def_{B_2} = \{i, j\}$. □

Как следствие этих определений любая переменная в use_B должна рассматриваться как активная на входе в блок B, в то время как переменные из def_B в начале блока B мертвы.

Значит, уравнения, связывающие *def* и *use* с неизвестными IN и OUT, определяются следующим образом:

$$\text{IN}[\text{ВХОД}] = \varnothing$$

И для всех базовых блоков B, отличных от выходного,

$$\text{IN}[B] = use_B \cup (\text{OUT}[B] - def_B)$$
$$\text{OUT}[B] = \bigcup_{S - \text{преемник } B} \text{IN}[S]$$

Первое уравнение определяет граничное условие, состоящее в том, что активных переменных при выходе из программы нет. Второе уравнение гласит, что переменная является активной при входе в блок, если она используется в блоке до переопределения или если она активна на выходе из блока и не переопределена в нем. Третье уравнение говорит о том, что переменная активна при выходе из блока тогда и только тогда, когда она активна при входе по крайней мере в один из блоков-преемников.

Сравнивая соотношения между уравнениями для активных переменных и для достигающих определений, можно заметить следующее.

- Оба множества используют объединение в качестве оператора сбора. Причина этого в том, что в каждой схеме потока данных мы распространяем информацию вдоль путей и нас интересует ситуация, когда существует хотя бы один путь с требуемым свойством, а не когда таким свойством обладают все пути.
- Однако поток информации в случае активных переменных идет "назад", обратно направлению потока управления, поскольку в этой задаче мы хотим убедиться, что использование переменной x в точке p передается всем точкам, предшествующим p вдоль путей выполнения, так что об использовании x становится известно в предшествующих точках, до ее использования.

Для решения задачи в обратном направлении вместо OUT[ВХОД] мы инициализируем IN[ВЫХОД]. Множества IN и OUT меняются ролями, а *use* и *def* заменяют соответственно *gen* и *kill*. Как и в случае достигающих определений решение уравнений для активных переменных не обязательно единственное, и мы хотим найти решение с наименьшим множеством активных переменных. Использующийся для этого алгоритм, по сути, представляет развернутую в обратном направлении версию алгоритма 9.11.

Алгоритм 9.14. Анализ активных переменных

ВХОД: граф потока с множествами *def* и *use*, вычисленными для каждого базового блока.

Выход: множества переменных, активных на входе ($\text{IN}[B]$) и выходе ($\text{OUT}[B]$) каждого блока B графа потока.

Метод: выполнить программу, приведенную на рис. 9.16. □

$\text{IN}[\text{Выход}] = \varnothing$;
for (каждый базовый блок B, отличный от выходного) $\text{IN}[B] = \varnothing$;
while (Внесены изменения в IN)
  for (каждый базовый блок B, отличный от выходного) {
    $\text{OUT}[B] = \bigcup_{S-\text{преемник } B} \text{IN}[S]$;
    $\text{IN}[B] = use_B \cup (\text{OUT}[B] - def_B)$;
  }

Рис. 9.16. Итеративный алгоритм для вычисления активных переменных

9.2.6 Доступные выражения

Выражение $x + y$ *доступно* (available) в точке p, если любой путь от входного узла к p вычисляет $x + y$ и после последнего такого вычисления до достижения p нет последующих присваиваний переменным x и y[5]. Когда речь идет о доступных выражениях, мы говорим, что блок *уничтожает* выражение $x + y$, если он присваивает (или может присваивать) x и y и после этого не вычисляет $x + y$ заново. Блок *генерирует* выражение $x + y$, если он вычисляет $x + y$ и не выполняет последующих переопределений x и y.

Заметим, что понятия "уничтожения" и "генерации" доступного выражения не те же, что в случае достигающих определений. Несмотря на это понятия "уничтожение" и "генерация" ведут себя, по сути, так же, как и уничтожение, и генерация для достигающих определений.

Основное применение информации о доступных выражениях — поиск глобальных общих подвыражений. Например, на рис. 9.17, *а* выражение $4 * i$ в блоке B_3 будет общим подвыражением, если $4 * i$ доступно во входной точке блока B_3. Это выражение будет доступно, если i не будет присвоено новое значение в блоке B_2 или если $4 * i$ будет заново вычислено после такого присвоения в блоке B_2, как показано на рис. 9.17, *б*.

Можно вычислить множество генерируемых выражений для каждой точки блока, проходя от начала до конца блока. В точке, предшествующей блоку, сгенерированных выражений нет. Если в точке p доступно множество выражений S, а q представляет собой точку после p с инструкцией `x = y+z` между ними, то мы образуем множество доступных в q выражений следующим образом.

---

[5] Как обычно в этой главе, оператор + означает обобщенный оператор, не обязательно оператор сложения.

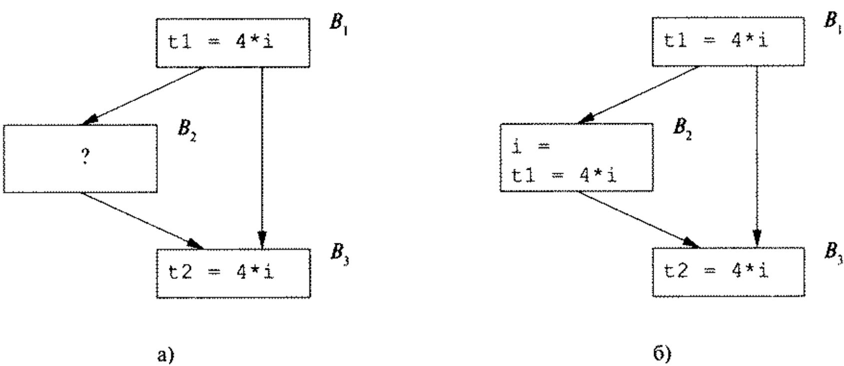


Рис. 9.17. Потенциальные общие подвыражения, пересекающие границы блоков

1. Добавляем к S выражение $y + z$.

2. Удаляем из S все выражения, включающие переменную x.

Заметим, что описанные действия должны выполняться в указанном порядке, так как x может совпадать с y или z. После того как мы достигнем конца блока, S будет представлять собой множество сгенерированных выражений блока. Множество уничтоженных выражений представляет собой множество всех выражений, скажем, $y + z$, таких, что y или zопределяется в блоке, и при этом $y + z$ блоком не генерируется.

Пример 9.15. Рассмотрим четыре инструкции, показанные на рис. 9.18. После первой инструкции доступно выражение $b + c$, после второй становится доступным выражение $a - d$, но $b + c$ более недоступно, поскольку при этом переопределяется b. Третья инструкция не делает $b + c$ доступным, поскольку в ней переопределяется c. После последней инструкции в связи с тем, что она изменяет d, становится недоступным и $a - d$. Итак, здесь не генерируется ни одно доступное выражение и при этом уничтожаются все выражения, включающие a, b, c или d.□

Мы можем найти доступные выражения методом, напоминающим метод вычисления достигающих определений. Предположим, что U — "универсальное" множество всех выражений, появляющихся в правой части одной или нескольких инструкций программы. Пусть для каждого блока B множество IN $[B]$ содержит выражения из U, доступные в точке непосредственно перед началом блока B, а OUT $[B]$ — такое же множество для точки, следующей за концом блока B. Определим e_gen_B как множество выражений, генерируемых B, а e_kill_B — как множество выражений из U, уничтожаемых в B. Заметим, что множества IN, OUT, *e_gen* и *e_kill* могут быть представлены в виде битовых векторов. Неизвестные множе-

Инструкция	Доступные выражения
	$\varnothing$
a = b + c	
	$\{b+c\}$
b = a - d	
	$\{a-d\}$
c = b + c	
	$\{a-d\}$
d = a - d	
	$\varnothing$

Рис. 9.18. Вычисление доступных выражений

ства IN и OUT связаны друг с другом и с известными *e_gen* и *e_kill* следующими соотношениями:

$$\text{OUT}[\text{Вход}] = \varnothing$$

и для всех базовых блоков B, отличных от входного,

$$\text{OUT}[B] = e_gen_B \cup (\text{IN}[B] - \text{e_kill}_B)$$

$$\text{IN}[B] = \bigcap_{P - \text{предшественник } B} \text{OUT}[P]$$

Приведенные выше уравнения выглядят почти идентичными уравнениям для достигающих определений. Как и в случае достигающих определений, граничное условие — $\text{OUT}[\text{Вход}] = \varnothing$, поскольку при выходе из входного узла доступных выражений нет. Наиболее важное отличие состоит в том, что оператор сбора в данном случае — не объединение, а пересечение. Пересечение множеств используется потому, что выражение доступно в начале блока только тогда, когда оно доступно в конце *всех* его предшественников (в отличие от определения, которое достигает начала блока, если достигает конца хотя бы одного из его предшественников).

Использование оператора $\cap$ вместо $\cup$ делает поведение уравнений для доступных выражений отличающимся от поведения уравнений для достигающих определений. В то время как ни одно из множеств не представляет собой единственное решение, в случае достигающих определений мы получаем наименьшее решение, соответствующее определению достижимости, и получаем его, начиная с предположения о недостижимости чего бы то ни было в какой угодно точке, а затем "наращиваем" его. При этом подходе мы никогда не предполагаем, что определение d может достичь точки p, до тех пор, пока не будет найден реальный путь распространения d до p. В отличие от этого для уравнений доступных выражений мы хотим получить решение с наибольшими множествами доступных выражений, а потому начинаем со слишком большого приближения и идем по пути его уменьшения.

Может показаться неочевидным, что, начав с предположения "доступно все (т.е. множество U) и везде, за исключением конца входного блока" и удаляя только те выражения, для которых мы находим пути, по которым они становятся недоступны, мы получим множество истинно доступных выражений. В случае доступных выражений консервативным является получение подмножества точного множества доступных выражений, и это именно то, что мы делаем. Аргументом в пользу консервативности подмножеств является наше предполагаемое использование информации для замены вычисления доступного выражения предварительно вычисленным значением. Отсутствие информации о доступности выражения только предотвратит возможное изменение кода, в то время как предположение о доступности выражения, когда на самом деле это не так, может привести к некорректному изменению программы, которое изменит вычисляемые ею результаты.

Пример 9.16. Рассмотрим единственный блок B_2 на рис. 9.19, чтобы проиллюстрировать влияние начального приближения $\text{OUT}[B_2]$ на $\text{IN}[B_2]$. Обозначим через G и K соответственно $e_gen_{B_2}$ и $e_kill_{B_2}$. Уравнениями потока данных для блока B_2 являются

$$\begin{aligned} \text{IN}[B_2] &= \text{OUT}[B_1] \cap \text{OUT}[B_2] \\ \text{OUT}[B_2] &= G \cup (\text{IN}[B_2] - K) \end{aligned}$$

Эти уравнения могут быть переписаны в виде рекуррентных соотношений, в которых I^j и O^j означают j-е приближения $\text{IN}[B_2]$ и $\text{OUT}[B_2]$ соответственно:

$$\begin{aligned} I^{j+1} &= \text{OUT}[B_1] \cap O^j \\ O^{j+1} &= G \cup \left(I^{j+1} - K\right) \end{aligned}$$

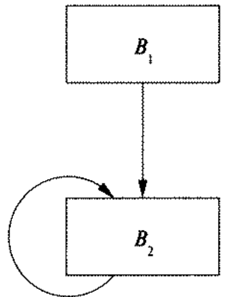


Рис. 9.19. Инициализация множеств OUT значениями ∅ слишком ограничивающая

Почему работает алгоритм для доступных выражений

Мы должны пояснить, почему выбор начальных значений OUT (за исключением входного блока), равных U, множеству всех выражений, приводит к консервативному решению уравнений потока данных, т.е. что все найденные доступные выражения *действительно* доступны. Во-первых, поскольку в данной схеме потока данных операцией сбора является пересечение, любая причина, по которой выражение $x + y$ не является доступным в некоторой точке, будет распространяться по графу потока по всем доступным путям, пока выражение $x + y$ не будет вычислено заново и не станет вновь доступным. Во-вторых, есть только две причины, по которым $x + y$ может быть недоступно.

1. $x + y$ уничтожается в блоке B, поскольку x или y определено без последующего вычисления $x + y$. В этом случае при первом применении передаточной функции f_B выражение $x + y$ будет удалено из OUT $[B]$.

2. $x + y$ никогда не вычисляется вдоль некоторого пути. Поскольку $x + y$ никогда не появляется в OUT [ВХОД] и никогда не генерируется вдоль рассматриваемого пути, можно показать по индукции по длине пути, что $x + y$ в конечном счете будет удалено из множеств IN и OUT вдоль этого пути.

Таким образом, по окончании внесения изменений решение, полученное итеративным алгоритмом на рис. 9.20, будет включать только истинно доступные выражения.

Начав с $O^0 = \varnothing$, мы получим $I^1 = \text{OUT}[B_1] \cap O^0 = \varnothing$. Однако, если начать с $O^0 = U$, то получим $I^1 = \text{OUT}[B_1] \cap O^0 = \text{OUT}[B_1]$, как и должно было быть. Интуитивно решение, полученное при $O^0 = U$, более желательно, поскольку корректно отражает тот факт, что выражения в OUT $[B_1]$ и не уничтоженные в B_2, остаются доступны в конце B_2. □

Алгоритм 9.17. Доступные выражения

ВХОД: граф потока, у которого для каждого блока B вычислены e_kill_B и e_gen_B. Начальный блок — B_1.

ВЫХОД: IN $[B]$ и OUT $[B]$, множества выражений, доступных на входе и выходе из каждого блока B графа потока.

МЕТОД: выполняется алгоритм, приведенный на рис. 9.20. Пояснение шагов этого алгоритма аналогично пояснению к алгоритму на рис. 9.14. □

OUT [Вход] = $\emptyset$;
for (каждый базовый блок B, отличный от входного) OUT $[B] = U$;
while (внесены изменения в OUT)
    for (каждый базовый блок B, отличный от входного) {
        IN $[B] = \bigcap_{P-\text{предшественник } B}$ OUT $[P]$;
        OUT $[B] = e_gen_B \cup (\text{IN}[B] - e_kill_B)$;
    }

Рис. 9.20. Итеративный алгоритм для вычисления доступных выражений

9.2.7 Резюме

В этом разделе мы рассмотрели три примера задач потоков данных: достигающие определения, активные переменные и доступные выражения. Как подытожено на рис. 9.21, каждая задача задается областью определения значений потока данных, его направлением, семейством передаточных функций, граничными условиями и оператором сбора. Обобщенно оператор сбора обозначен как $\wedge$.

В последней строке таблицы показаны инициализирующие значения, использующиеся в итеративных алгоритмах. Эти значения выбираются таким образом, чтобы алгоритм находил наиболее точное решение уравнений. Говоря строго, данный выбор не является частью определения задачи потока данных; это артефакт, необходимый для итеративного алгоритма ее решения. Например, мы видели, как передаточная функция базового блока может быть получена путем композиции функций отдельных инструкций блока; аналогичный композиционный подход может использоваться и для вычисления передаточной функции для всей процедуры или для вычисления передаточной функции от входа процедуры до любой точки программы. Этот подход будет рассмотрен в разделе 9.7.

9.2.8 Упражнения к разделу 9.2

Упражнение 9.2.1. Для графа потока на рис. 9.10 (см. упражнения к разделу 9.1) вычислите

а) множества *gen* и *kill* для каждого блока;

б) множества IN и OUT для каждого блока.

Упражнение 9.2.2. Для графа потока на рис. 9.10 вычислите множества *e_gen*, *e_kill*, IN и OUT для доступных выражений.

Упражнение 9.2.3. Для графа потока на рис. 9.10 вычислите множества *def*, *use*, IN и OUT для анализа активных переменных.

	Достигающие определения	Активные переменные	Доступные выражения
Область определения	Множества определений	Множества переменных	Множества выражений
Направление	Прямое	Обратное	Прямое
Передаточная функция	$gen_B \cup (x - kill_B)$	$use_B \cup (x - def_B)$	$e_gen_B \cup (x - e_kill_B)$
Граничное условие	$\text{OUT}[\text{ВХОД}] = \varnothing$	$\text{IN}[\text{ВЫХОД}] = \varnothing$	$\text{OUT}[\text{ВХОД}] = \varnothing$
Оператор сбора $\wedge$	$\cup$	$\cup$	$\cap$
Уравнения	$\text{OUT}[B] = f_B(\text{IN}[B])$ $\text{IN}[B] = \wedge_{P \in pred(B)} \text{OUT}[P]$	$\text{IN}[B] = f_B(\text{OUT}[B])$ $\text{OUT}[B] = \wedge_{S \in succ(B)} \text{IN}[S]$	$\text{OUT}[B] = f_B(\text{IN}[B])$ $\text{IN}[B] = \wedge_{P \in pred(B)} \text{OUT}[P]$
Инициализация	$\text{OUT}[B] = \varnothing$	$\text{IN}[B] = \varnothing$	$\text{OUT}[B] = U$

Функции $pred(B)$ и $succ(B)$ у оператора сбора означают множества соответственно предшественников и преемников блока B в графе потока.

Рис. 9.21. Задачи потоков данных

! Упражнение 9.2.4. Предположим, что V — множество комплексных чисел. Какие из приведенных операций могут служить в качестве оператора сбора для полурешетки на V?

а) Сложение $(a + ib) \wedge (c + id) = (a + c) + i\,(b + d)$.

б) Умножение $(a + ib) \wedge (c + id) = (ac - bd) + i\,(ad + bc)$.

в) Покомпонентный минимум $(a + ib) \wedge (c + id) = \min(a, c) + i \min(b, d)$.

г) Покомпонентный максимум $(a + ib) \wedge (c + id) = \max(a, c) + i \max(b, d)$.

! Упражнение 9.2.5. Утверждается, что если блок B состоит из n инструкций и i-я инструкция имеет множества gen_i и $kill_i$, то передаточная функция для блока B имеет множества gen_B и $kill_B$, определяемые как

$$\begin{aligned} kill_B &= kill_1 \cup kill_2 \cup \cdots \cup kill_n \\ gen_B &= gen_n \cup (gen_{n-1} - kill_n) \cup (gen_{n-2} - kill_{n-1} - kill_n) \cup \\ &\quad \cup \cdots \cup (gen_1 - kill_2 - kill_3 - \cdots - kill_n) \end{aligned}$$

Докажите это утверждение по индукции по n.

! Упражнение 9.2.6. Докажите по индукции по числу итераций цикла for в строках 4–6 алгоритма 9.11, что ни одно из множеств IN или OUT никогда не уменьшается. Иначе говоря, если в какой-то момент определение помещено в одно из этих множеств, то в последующем оно никогда не будет из него удалено.

! Упражнение 9.2.7. Покажите корректность алгоритма 9.11, т.е. покажите, что

а) если определение d помещается в IN $[N]$ или OUT $[B]$, то существует путь от d к началу или концу блока B соответственно, вдоль которого переменная, определенная в d, не может быть переопределена;

б) если определение d не помещается в IN $[N]$ или OUT $[B]$, то не существует пути от d к началу или концу блока B соответственно, вдоль которого переменная, определенная в d, не может быть переопределена.

! Упражнение 9.2.8. Докажите следующие утверждения об алгоритме 9.14.

а) Множества IN и OUT никогда не уменьшаются.

б) Если переменная x помещена в IN $[B]$ или OUT $[B]$, то существует путь от начала или конца блока B соответственно, вдоль которого x может быть использована.

в) Если переменная x не помещена в IN $[B]$ или OUT $[B]$, то не существует пути от начала или конца блока B соответственно, вдоль которого x может быть использована.

! **Упражнение 9.2.9.** Докажите следующие утверждения об алгоритме 9.17.

а) Множества IN и OUT никогда не растут; т.е. последовательные значения этих множеств являются подмножествами (не обязательно истинными) их предыдущих значений.

б) Если выражение e удалено из IN $[B]$ или OUT $[B]$, то существует путь от входа графа потока к началу или концу блока B соответственно, вдоль которого e либо никогда не вычисляется, либо после его последнего вычисления один из его аргументов может быть переопределен.

в) Если выражение e остается в IN $[B]$ или OUT $[B]$, то вдоль каждого пути от входа графа потока к началу или концу блока B соответственно вычисляется e, и после его последнего вычисления ни один из его аргументов не может быть переопределен.

! **Упражнение 9.2.10.** Проницательный читатель заметит, что в алгоритме 9.11 можно сэкономить некоторое время путем инициализации OUT $[B]$ значением gen_B для всех блоков B. Аналогично в алгоритме 9.14 можно инициализировать IN $[B]$ значением gen_B. Мы не поступаем так из соображений единообразия в изложении темы, как будет видно при рассмотрении алгоритма 9.21. Однако можно ли в алгоритме 9.17 инициализировать множества OUT $[B]$ значениями e_gen_B? Поясните ваш ответ.

! **Упражнение 9.2.11.** Наши анализы потоков данных не используют преимущества семантики условных выражений. Предположим, в конце базового блока обнаружена проверка наподобие

```
if (x < 10) goto ...
```

Как можно воспользоваться тем, что проверка $x < 10$ означает углубление наших знаний о достигающих определениях? Под углублением знаний здесь предполагается, что мы устраняем некоторые достигающие определения, которые в действительности не в состоянии достичь определенной точки программы.

9.3 Основы анализа потока данных

Рассмотрев несколько практических примеров абстракций потока данных, перейдем к изучению схем потоков данных в общем виде. Мы формально ответим на ряд фундаментальных вопросов об алгоритмах потоков данных.

1. При каких условиях корректен итеративный алгоритм, использующийся в анализе потока данных?

2. Насколько точно решение, полученное итеративным алгоритмом?

3. Будет ли сходиться итеративный алгоритм?

4. Каков смысл решения уравнений?

В разделе 9.2 мы неформально ответили на каждый из этих вопросов при изучении задачи о достигающих определениях. Вместо ответов на те же вопросы для каждой из последующих задач с нуля мы опирались на аналогию с уже рассмотренной задачей. Здесь же мы представим общий подход, который раз и навсегда отвечает на все поставленные вопросы для большого семейства задач потоков данных. Сперва мы определим желательные свойства схем потоков данных и докажем в качестве следствия этих свойств корректность, точность и сходимость алгоритма потока данных, а также выясним смысл его решения. Таким образом, чтобы понять старые алгоритмы или сформулировать новые, мы просто показываем, что предложенные определения задач потоков данных обладают рядом свойств, и тут же получаем ответы на все поставленные выше сложные вопросы.

Концепция общей теоретической структуры для класса схем имеет и практические следствия. Эта структура помогает нам идентифицировать повторно используемые компоненты алгоритмов при проектировании нашего программного обеспечения. Тем самым не только снижается количество работы, которую надо сделать программисту, но и уменьшается количество возможных ошибок, которые могли бы появиться при многократном программировании схожих деталей.

Структура анализа потока данных (data-flow analysis framework) $(D, V, \wedge, F)$ состоит из

1. направления потока данных D, который может принимать значение "прямой" или "обратный";

2. полурешетки (см. соответствующее определение в разделе 9.3.1), включающей *область определения* значений V и оператор сбора $\wedge$;

3. семейства F передаточных функций, областями определения и значений которых является V; это семейство должно включать функции, подходящие для граничных условий и представляющие собой константные передаточные функции для входного и выходного узлов любого графа потока.

9.3.1 Полурешетки

Полурешетка (semilattice) представляет собой множество V и бинарный оператор сбора $\wedge$, такой, что для всех x, y и z из V выполняются следующие соотношения.

1. $x \wedge x = x$ (идемпотентность)
2. $x \wedge y = y \wedge x$ (коммутативность)
3. $x \wedge (y \wedge z) = (x \wedge y) \wedge z$ (ассоциативность)

Полурешетка имеет *верхний* элемент, обозначаемый как $\top$, такой, что

$$\text{для всех } x \in V \text{ выполняется } \top \wedge x = x$$

Полурешетка может иметь (необязательно) *нижний* элемент, обозначаемый как $\bot$, такой, что

$$\text{для всех } x \in V \text{ выполняется } \bot \wedge x = \bot$$

Частичные порядки

Как мы увидим, оператор сбора полурешетки определяет частичный порядок значений из области определения. Отношение $\leqslant$ представляет собой *частичный порядок* (partial order) на множестве V, если для всех x, y и z из V выполняются следующие соотношения.

1. $x \leqslant x$ (рефлексивность)
2. Если $x \leqslant y$ и $y \leqslant x$, то $x = y$ (*антисимметрия*)
3. Если $x \leqslant y$ и $y \leqslant z$, то $x \leqslant z$ (*транзитивность*)

Пара $(V, \leqslant)$ называется *частично упорядоченным множеством* (partially ordered set; сокращенно – poset[6]). Для частично упорядоченного множества удобно ввести также отношение $<$, определяемое как

$$x < y \text{ тогда и только тогда, когда } (x \leqslant y) \text{ и } (x \neq y)$$

Частичный порядок в полурешетке

Определим частичный порядок для полурешетки $(V, \wedge)$. Для x и y из V определим

$$x \leqslant y \text{ тогда и только тогда, когда } x \wedge y = x$$

Поскольку оператор сбора $\wedge$ идемпотентен, коммутативен и ассоциативен, определенный таким образом порядок $\leqslant$ является рефлексивным, антисимметричным и транзитивным. Чтобы понять, почему это так, заметим следующее.

- Рефлексивность: для всех x $x \leqslant x$. Доказательство этого факта основано на том, что $x \wedge x = x$ в силу идемпотентности оператора сбора.

[6]Мы не рискнули перевести это сокращение на русский язык, получив в результате "чум". – *Прим. пер.*

- Антисимметрия. Если $x \leqslant y$ и $y \leqslant x$, то $x = y$. По определению $x \leqslant y$ означает $x \wedge y = x$, а $y \leqslant x$ означает $y \wedge x = y$. В силу коммутативности оператора $\wedge$ получаем $x = (x \wedge y) = (y \wedge x) = y$.

- Транзитивность. Если $x \leqslant y$ и $y \leqslant z$, то $x \leqslant z$. По определению $x \leqslant y$ и $y \leqslant z$ означают $x \wedge y = x$ и $y \wedge z = y$. Использование ассоциативности оператора сбора дает $(x \wedge z) = ((x \wedge y) \wedge z) = (x \wedge (y \wedge z)) = (x \wedge y) = x$. Поскольку показано, что $x \wedge z = x$, получаем $x \leqslant z$, что и доказывает транзитивность.

Пример 9.18. Операторы сбора, использованные в разделе 9.2, — это операторы объединения и пересечения множеств. Оба они идемпотентны, коммутативны и ассоциативны. Для объединения множеств верхним элементом является $\varnothing$, а нижним — универсальное множество U, поскольку для любого подмножества $x \subseteq U$ выполняются соотношения $\varnothing \cup x = x$ и $U \cup x = U$. В случае пересечения множеств $\top$ представляет собой U, а $\perp$ — $\varnothing$. Область определения значения полурешетки V в этом случае представляет собой множество всех подмножеств U, которое иногда называется *показательным множеством* (power set) U и обозначается как 2^U.

Для всех x и y из V из соотношения $x \cup y = x$ вытекает $x \supseteq y$; таким образом, частичный порядок, продиктованный объединением множеств, представляет собой $\supseteq$, включение множества. Соответственно, частичный порядок, продиктованный пересечением множеств, представляет собой $\subseteq$, содержание в множестве. Иначе говоря, для пересечения множеств множества с меньшим количеством элементов рассматриваются данным частичным порядком как меньшие. Однако в случае объединения множеств меньшими считаются множества с *бóльшим* количеством элементов. То, что бóльшие по размеру множества оказываются меньшими в частичном порядке, противоречит интуитивному взгляду, но эта ситуация — неизбежное следствие принятых определений[7].

Как говорилось в разделе 9.2, обычно имеется много решений уравнений потоков данных, среди которых наибольшее решение (в смысле частичного порядка $\leqslant$) является наиболее точным. Например, для достигающих определений наиболее точным среди всех решений уравнений потоков данных является уравнение с наименьшим количеством определений, которое соответствует наибольшему элементу в частичном порядке, определенном оператором сбора — объединением. В случае доступных выражений наиболее точным решением является решение с наибольшим количеством выражений. И вновь это решение является наибольшим в частичном порядке, определенном пересечением в качестве оператора сбора. □

[7]Если определить частичный порядок не как $\leqslant$, а как $\geqslant$, то проблема останется, просто вместо пересечений проблемными окажутся объединения множеств.

Наибольшие нижние границы

Существует еще одно полезное соотношение между оператором сбора и определяемым им частичным порядком. Предположим, что $(V, \wedge)$ — полурешетка. *Наибольшей нижней границей* (greatest lower bound — glb) области определения элементов x и y является элемент g, такой, что

1. $g \leqslant x$;
2. $g \leqslant y$;
3. если z — любой элемент, такой, что $z \leqslant x$ и $z \leqslant y$, то $z \leqslant g$.

Отсюда следует, что $x \wedge y$ является их наибольшей нижней границей. Чтобы увидеть, почему это так, положим $g = x \wedge y$ и рассмотрим следующую аргументацию.

- $g \leqslant x$, поскольку $(x \wedge y) \wedge x = x \wedge y$. Доказательство включает простое использование ассоциативности, коммутативности и идемпотентности:

$$
\begin{aligned}
g \wedge x = ((x \wedge y) \wedge x) &= (x \wedge (y \wedge x)) = \\
= (x \wedge (x \wedge y)) &= ((x \wedge x) \wedge y) = \\
= (x \wedge y) &= g
\end{aligned}
$$

- $g \leqslant y$ доказывается аналогично.

- Предположим, что z представляет собой любой элемент, такой, что $z \leqslant x$ и $\leqslant y$. Мы утверждаем, что $z \leqslant g$, а следовательно, z не может быть наибольшей нижней границей x и y, если только не равно g. Вот доказательство этого утверждения: $(z \wedge g) = (z \wedge (x \wedge y)) = ((z \wedge x) \wedge y)$. Поскольку $z \leqslant x$, мы знаем, что $(z \wedge x) = z$, так что $(z \wedge g) = (z \wedge y)$. Так как $z \leqslant y$, нам известно, что $z \wedge y = z$, а следовательно, $z \wedge g = z$. Итак, мы доказали, что $z \leqslant g$, и делаем вывод, что $g = x \wedge y$ является единственной наибольшей нижней границей x и y.

Диаграммы решеток

Зачастую полезно изображать область определения V в виде диаграммы решетки, представляющей собой граф, узлами которого являются элементы V, а ребра направлены вниз от x к y, если $y \leqslant x$. Например, на рис. 9.22 показано множество V для схемы потоков данных для достигающих определений, в которой имеются три определения: d_1, d_2 и d_3. Поскольку $\leqslant$ в данном случае — $\supseteq$, ребра направлены вниз от любого подмножества из этих трех определений к каждому из их надмножеств. В силу транзитивности $\leqslant$ для удобства на диаграмме будут

Объединения, наименьшие верхние границы и решетки

Над элементами частично упорядоченного множества по аналогии с операцией наибольшей нижней границы можно определить *наименьшую верхнюю границу* (least upper bound — lub) элементов x и y как элемент b, такой, что $x \leqslant b$, $y \leqslant b$, и если z — любой элемент, такой, что $x \leqslant z$ и $y \leqslant z$, то $b \leqslant z$. Можно показать, что если такой элемент b существует, то он единственный.

В истинной *решетке* над элементами области определения существует две операции — уже знакомая нам операция сбора $\wedge$ и оператор *объединения*, обозначаемый как $\vee$, который дает наименьшую верхнюю границу двух элементов (которая, следовательно, должна всегда существовать в решетке). Мы рассматриваем только "половинные" решетки, в которых имеется только один из операторов сбора или объединения. Таким образом, рассматриваемые нами полурешетки являются *полурешетками сбора* (meet semilattices). Можно также говорить о *полурешетках объединения* (join semilattices), у которых существует только один оператор объединения; имеется ряд книг, в которых при анализе программ используется понятие полурешеток объединения. Но поскольку традиционно литература, посвященная потокам данных, говорит о решетках сбора, в данной книге мы поступим так же.

опущены ребра от x к y, если на диаграмме имеется иной путь от x к y. Таким образом, хотя $\{d_1, d_2, d_3\} \leqslant \{d_1\}$, мы не изображаем это ребро из-за наличия, например, пути, идущего через $\{d_1, d_2\}$.

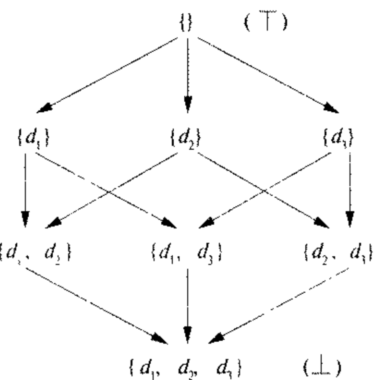


Рис. 9.22. Решетка подмножеств определений

Следует заметить, что на такой диаграмме ясно видны результаты операции сбора. Поскольку $x \wedge y$ представляет собой наибольшую нижнюю границу, всегда имеется большее z, к которому ведут пути от x и y. Например, если x — $\{d_1\}$, а y — $\{d_2\}$, то z на рис. 9.22 представляет собой $\{d_1, d_2\}$, что логично, так как оператор сбора представляет собой объединение. Верхний элемент множества всегда находится наверху в диаграмме, так что от элемента $\top$ существуют пути к каждому из остальных элементов. Аналогично нижний элемент находится внизу в решетке, и к этому элементу $\perp$ имеется путь от каждого из остальных элементов.

Решетки произведений

Хотя на рис. 9.22 имеется только три определения, диаграмма решетки типичной программы может оказаться существенно большей. Множество значений потока данных является показательным множеством определений, и при наличии в программе n определений содержит 2^n элементов. Однако определение достигает точки программы независимо от достижимости других определений. Таким образом, решетку[8] определений можно рассматривать как полученную в результате "произведения" простых решеток для каждого из определений. Если бы в программе имелось только одно определение d, то решетка имела бы два элемента — пустое множество $\{\}$, являющееся верхним элементом, и нижний элемент $\{d\}$.

Формально можно построить решетки произведений следующим образом. Предположим, что $(A, \wedge_A)$ и $(B, \wedge_B)$ — (полу)решетки. *Решетка произведения* (product lattice) этих двух решеток определяется следующим образом.

1. Областью определения решетки произведения является $A \times B$.

2. Оператор сбора $\wedge$ решетки произведения определяется следующим образом. Если (a, b) и (a', b') являются элементами области определения решетки произведения, то

$$(a, b) \wedge (a', b') = (a \wedge_A a', b \wedge_B b') \tag{9.2}$$

Очень просто выразить частичный порядок $\leqslant$ для решетки произведения в терминах частичных порядков $\leqslant_A$ и $\leqslant_B$ для A и B:

$$(a, b) \leqslant (a', b') \text{ тогда и только тогда, когда } a \leqslant_A a' \text{ и } b \leqslant_B b' \tag{9.3}$$

Чтобы понять, почему (9.3) следует из (9.2), заметим, что

$$(a, b) \wedge (a', b') = (a \wedge_A a', b \wedge_B b')$$

---

[8]Здесь и далее мы часто опускаем приставку "полу", поскольку решетки наподобие рассматриваемой имеют оба оператора, даже если одним мы и не пользуемся.

Можно задаться вопросом “При каких условиях $(a \wedge_A a', b \wedge_B b') = (a, b)$?” Это соотношение выполняется в точности тогда, когда $a \wedge_A a' = a$ и $b \wedge_B b' = b$. Но эти два условия те же, что и $a \leqslant_A a'$ и $b \leqslant_B b'$.

Произведение решеток — операция ассоциативная, так что можно показать, что правила (9.2) и (9.3) распространяются на любое количество решеток, т.е. если даны решетки $(A_i, \wedge_i)$, $i = 1, 2, \ldots, k$, то произведение всех k решеток в указанном порядке имеет область определения $A_1 \times A_2 \times \cdots \times A_k$, оператор сбора, определяемый как

$$(a_1, a_2, \ldots, a_k) \wedge (b_1, b_2, \ldots, b_k) = (a_1 \wedge b_1, a_2 \wedge b_2, \ldots, a_k \wedge b_k),$$

и частичный порядок

$$(a_1, a_2, \ldots, a_k) \leqslant (b_1, b_2, \ldots, b_k) \text{ тогда и только тогда, когда } a_i \leqslant b_i \text{ для всех } i$$

Высота полурешетки

Получить информацию о скорости сходимости алгоритма анализа потока данных можно путем изучения связанной с ним полурешетки. *Восходящей цепочкой* в частично упорядоченном множестве $(V, \leqslant)$ является последовательность, в которой $x_1 < x_2 < \cdots < x_n$. *Высота* полурешетки представляет собой наибольшее количество отношений < в восходящей цепочке, т.е. высота на единицу меньше количества элементов в такой цепочке. Например, высота полурешетки достигающих определений программы с n определениями равна n.

Показать сходимость итеративного алгоритма потока данных существенно проще, если полурешетка имеет конечную высоту. Ясно, что конечную высоту имеет полурешетка, состоящая из конечного множества значений; возможна также ситуация, когда решетка с бесконечным количеством значений также имеет конечную высоту. Соответствующим примером является решетка, используемая в алгоритме распространения констант, который будет подробно рассмотрен в разделе 9.4.

9.3.2 Передаточные функции

Семейство передаточных функций $F : V \to V$ в структуре потока данных обладает следующими свойствами.

1. F содержит тождественную функцию I, такую, что $I(x) = x$ для всех x из V.

2. F замкнуто относительно композиции, т.е. для любых двух функций f и g из F функция h, определяемая как $h(x) = g(f(x))$, также принадлежит F.

Пример 9.19. В случае достигающих определений тождественная функция F представляет собой частный случай, когда *gen* и *kill* являются пустыми множествами. Замкнутость относительно композиции фактически была показана в разделе 9.2.4; вкратце повторим здесь это доказательство. Предположим, что у нас есть функции

$$f_1(x) = G_1 \cup (x - K_1) \text{ и } f_2(x) = G_2 \cup (x - K_2)$$

Тогда

$$f_2(f_1(x)) = G_2 \cup ((G_1 \cup (x - K_1)) - K_2)$$

Правая часть приведенной формулы алгебраически эквивалентна

$$(G_2 \cup (G_1 - K_2)) \cup (x - (K_1 \cup K_2))$$

Если положить $K = K_1 \cup K_2$ и $G = G_2 \cup (G_1 - K_2)$, то композиция функций f_1 и f_2 принимает вид $f(x) = G \cup (x - K)$, что делает ее принадлежащей семейству F. Если рассмотреть доступные выражения, то можно применить те же аргументы, которые использованы для достигающих определений, и показать как наличие тождественной функции в семействе F, так и его замкнутость относительно композиции. □

Монотонные структуры

Чтобы итеративный алгоритм потока данных работал, структура потока данных должна удовлетворять еще одному условию. Мы говорим, что структура *монотонна*, если при применении любой передаточной функции f из F к двум членам V, первый из которых не превышает второй, полученный результат для первого члена не превышает результат вычислений для второго члена.

Формально структура потока данных $(D, F, V, \wedge)$ *монотонна*, если

$$\text{из } x \leqslant y \text{ следует } f(x) \leqslant f(y) \text{ для всех } x \text{ и } y \text{ из } V \text{ и } f \text{ из } F \qquad (9.4)$$

Монотонность может быть определена и следующим эквивалентным образом:

$$f(x \wedge y) \leqslant f(x) \wedge f(y) \text{ для всех } x \text{ и } y \text{ из } V \text{ и } f \text{ из } F \qquad (9.5)$$

Уравнение (9.5) гласит, что если собрать два значения и применить передаточную функцию f, то результат не превзойдет результат, полученный путем применения f к отдельным значениям с последующим их сбором. Эти два определения монотонности кажутся слишком различными, так что они оба оказываются весьма полезны — в разных ситуациях мы будем пользоваться разными определениями. Ниже приведен набросок доказательства их эквивалентности.

Сначала выведем (9.5) из (9.4). Поскольку $x \wedge y$ представляет собой наибольшую нижнюю границу x и y, мы знаем, что

$$x \wedge y \leqslant x \text{ и } x \wedge y \leqslant y$$

Таким образом, согласно (9.4)

$$f(x \wedge y) \leqslant f(x) \text{ и } f(x \wedge y) \leqslant f(y)$$

Поскольку $f(x) \wedge f(y)$ представляет собой наибольшую нижнюю границу $f(x)$ и $f(y)$, мы получаем (9.5).

Теперь предположим, что выполняется (9.5), и докажем, что при этом выполняется (9.4). Мы принимаем, что $x \leqslant y$, и используем (9.5) для того, чтобы заключить, что $f(x) \leqslant f(y)$, что докажет (9.4). Уравнение (9.5) гласит, что

$$f(x \wedge y) \leqslant f(x) \wedge f(y)$$

Но поскольку мы предположили, что $x \leqslant y$, по определению $x \wedge y = x$. Таким образом, (9.5) приходит к виду

$$f(x) \leqslant f(x) \wedge f(y)$$

Поскольку $f(x) \wedge f(y)$ представляет собой наибольшую нижнюю границу $f(x)$ и $f(y)$, мы знаем, что $f(x) \wedge f(y) \leqslant f(y)$. Таким образом,

$$f(x) \leqslant f(x) \wedge f(y) \leqslant f(y)$$

и из (9.5) следует (9.4).

Дистрибутивные структуры

Зачастую структура удовлетворяет более строгому, чем (9.5), условию, которое мы называем *условием дистрибутивности*:

$$f(x \wedge y) = f(x) \wedge f(y) \text{ для всех } x \text{ и } y \text{ из } V \text{ и } f \text{ из } F$$

Безусловно, если $a = b$, то в соответствии со свойством идемпотентности $a \wedge b = a$, так что $a \leqslant b$. Таким образом, дистрибутивность влечет за собой монотонность (однако обратное неверно).

Пример 9.20. Пусть y и z — множества определений в структуре достигающих определений. Пусть f — функция, определяемая уравнением $f(x) = G \cup (x - K)$ для некоторых множеств определений G и K. Можно убедиться, что структура достигающих определений удовлетворяет условию дистрибутивности, проверив, что

$$G \cup ((y \cup z) - K) = (G \cup (y - K)) \cup (G \cup (z - K))$$

Это уравнение только кажется страшным. Начнем с определений, входящих в G. Очевидно, что эти определения входят в множества, определяемые как левой, так и правой частями уравнения. Таким образом, необходимо рассмотреть только определения, не входящие в G. В этом случае можно убрать G из уравнения и проверить выполнение уравнения

$$(y \cup z) - K = (y - K) \cup (z - K)$$

Это тождество элементарно проверяется при помощи диаграммы Венна. □

9.3.3 Итеративный алгоритм в обобщенной структуре

Можно обобщить алгоритм 9.11 для работы с большим количеством задач потоков данных.

Алгоритм 9.21. Итеративное решение обобщенной задачи потока данных

ВХОД: структура потока данных со следующими компонентами:

1. граф потока данных с отдельно помеченными входным и выходным узлами;
2. направление потока данных D;
3. множество значений V;
4. оператор сбора $\wedge$;
5. множество функций F, где f_B из F представляет собой передаточную функцию для блока B;
6. константное значение $v_{\text{ВХОД}}$ или $v_{\text{ВЫХОД}}$ из V, представляющее собой граничное условие для прямой и обратной структуры соответственно.

ВЫХОД: значения из V для IN$[B]$ и OUT$[B]$ для каждого блока B в графе потока данных.

МЕТОД: алгоритмы для решения прямой и обратной задач потока данных показаны на рис. 9.23, *а* и *б* соответственно. Как и в случае хорошо знакомых итеративных алгоритмов для потоков данных из раздела 9.2, мы вычисляем IN и OUT для каждого блока как ряд последовательных приближений. □

Можно записать прямую и обратную версии алгоритма 9.21 с функцией, реализующей оператор сбора, в качестве параметра. В качестве параметра выступает и функция, реализующая передаточную функцию для каждого блока. Сам граф потока и граничное значение также являются параметрами. Таким образом, разработчик компилятора может избежать записи базового итеративного алгоритма для каждой структуры потока данных, используемой на стадии оптимизации.

Можно воспользоваться рассматривавшейся абстрактной схемой для доказательства ряда полезных свойств итеративного алгоритма.

1) $\text{OUT}[\text{Вход}] = v_{\text{Вход}}$;
2) **for** (каждый базовый блок B, отличный от входного) $\text{OUT}[B] = \top$;
3) **while** (внесены изменения в OUT)
4) **for** (каждый базовый блок B, отличный от входного) {
5) $\text{IN}[B] = \wedge_{P-\text{предшественник } B}\text{OUT}[P]$;
6) $\text{OUT}[B] = f_B(\text{IN}[B])$;
}

а) Итеративный алгоритм для прямой задачи потока данных

1) $\text{IN}[\text{Выход}] = v_{\text{Выход}}$;
2) **for** (каждый базовый блок B, отличный от выходного) $\text{IN}[B] = \top$;
3) **while** (внесены изменения в IN)
4) **for** (каждый базовый блок B, отличный от выходного) {
5) $\text{OUT}[B] = \wedge_{S-\text{преемник } B}\text{IN}[S]$;
6) $\text{IN}[B] = f_B(\text{OUT}[B])$;
}

б) Итеративный алгоритм для обратной задачи потока данных

Рис. 9.23. Прямая и обратная версии итеративного алгоритма

1. Если алгоритм 9.21 сходится, то получающийся результат является решением уравнений потоков данных.

2. Если структура монотонна, то найденное решение оказывается максимальной фиксированной точкой (maximum fixedpoint) уравнений потоков данных. *Максимальная фиксированная точка* представляет собой решение, обладающее тем свойством, что в любом другом решении значения $\text{IN}[B]$ и $\text{OUT}[B]$ не превышают (в смысле оператора $\leqslant$) соответствующих значений максимальной фиксированной точки.

3. Если полурешетка структуры монотонна и имеет конечную высоту, то алгоритм гарантированно сходится.

Докажем указанные свойства для прямой структуры. Доказательство для обратной структуры, по сути, то же самое. Показать первое свойство очень просто. Если при выходе из некоторой итерации цикла while уравнения не удовлетворяются, то в OUT (в прямом случае) или в IN (в обратном случае) будет внесено по крайней мере одно изменение, так что потребуется вновь вернуться в цикл и выполнить очередную его итерацию.

Чтобы доказать второе свойство, сначала покажем, что значения $\text{IN}[B]$ и $\text{OUT}[B]$ для любого B в процессе выполнения итераций алгоритма могут толь-

ко уменьшаться (в смысле отношения $\leqslant$ для решетки). Это утверждение можно доказать по индукции.

Базис: в качестве базиса индукции надо показать, что значения IN $[B]$ и OUT $[B]$ после первой итерации не больше, чем инициализирующие значения. Это утверждение тривиально, поскольку IN $[B]$ и OUT $[B]$ для всех блоков B, отличающихся от входного, инициализируются значением $\top$.

Индукция: предположим, что после k-й итерации все значения не больше значений после $(k-1)$-й итерации, и покажем, что то же самое выполняется и на $(k+1)$-й итерации по отношению к k-й. Строка 5 на рис. 9.23, *а* содержит

$$\text{IN}[B] = \wedge_{P-\text{предшественник } B}\text{OUT}[P]$$

Воспользуемся обозначениями $\text{IN}[B]^i$ и $\text{OUT}[B]^i$ для значений IN $[B]$ и OUT $[B]$ после итерации i. Если $\text{OUT}[P]^k \leqslant \text{OUT}[P]^{k-1}$, то в силу свойств оператора сбора $\text{IN}[B]^{k+1} \leqslant \text{IN}[B]^k$. Далее, строка 6 гласит

$$\text{OUT}[B] = f_B(\text{IN}[B])$$

В силу монотонности из $\text{IN}[B]^{k+1} \leqslant \text{IN}[B]^k$ вытекает $\text{OUT}[B]^{k+1} \leqslant \text{OUT}[B]^k$.

Заметим, что любые изменения значений IN $[B]$ и OUT $[B]$ обязаны удовлетворять уравнению. Операторы сбора возвращают наибольшую нижнюю границу своих операндов, а передаточные функции возвращают только решения, согласующиеся с самим блоком и заданными входными данными. Таким образом, по завершении работы итеративного алгоритма в результате должны получиться значения, которые как минимум не меньше соответствующих значений любого другого решения, т.е. результатом работы алгоритма 9.21 является максимальная фиксированная точка уравнений.

Наконец, рассмотрим третье утверждение, в котором структура потока данных имеет конечную высоту. Поскольку значения каждого IN $[B]$ и OUT $[B]$ при каждом изменении уменьшаются, а алгоритм завершает свою работу при отсутствии изменений, он гарантированно сходится после количества итераций, не превышающего произведения высоты структуры и количества узлов графа потока.

9.3.4 Смысл решения потока данных

Мы знаем, что решение, найденное при помощи итеративного алгоритма, представляет собой максимальную фиксированную точку, но что представляет собой данное решение с точки зрения семантики программы? Чтобы понять решение схемы потока данных $(D, F, V, \wedge)$, опишем вначале, каким должно быть идеальное решение структуры. Мы покажем, что в общем случае это идеальное решение не может быть получено, но алгоритм 9.21 дает нам консервативное приближение к этому идеалу.

Идеальное решение

Без потери общности будем считать, что интересующая нас структура потока данных является задачей прямого потока. Рассмотрим входную точку базового блока B. Идеальное решение начинается с поиска всех *возможных* путей выполнения, ведущих от входной точки программы к началу B. Путь "возможен", только если некоторое вычисление программы следует в точности по этому пути. Идеальное решение должно вычислить значение потока данных в конце каждого возможного пути и применить к найденным значениям оператор сбора для получения их наибольшей нижней границы. Тогда никакое выполнение программы не в состоянии привести к меньшему значению для данной ее точки. Кроме того, эта граница плотная: не существует большего значения потока данных, которое являлось бы наибольшей нижней границей значения, вычисленного по всем возможным путям графа потока к базовому блоку B.

Попытаемся определить идеальное решение более формально. Пусть f_B — передаточная функция для каждого базового блока B в графе потока. Рассмотрим путь

$$P = \text{Вход} \to B_1 \to B_2 \to \cdots \to B_{k-1} \to B_k$$

от входного узла к некоторому блоку B_k. Путь программы может содержать циклы, так что в пути P один и тот же базовый блок может появляться неоднократно. Определим *передаточную функцию* f_P для P как композицию $f_{B_1}, f_{B_2}, \ldots, f_{B_{k-1}}$. Обратите внимание, что f_{B_k} не является частью данной композиции, отражая тот факт, что путь достигает начала базового блока B_k, но не его конца. Значение потока данных, создаваемое этим путем, представляет собой $f_P(v_{\text{Вход}})$, где $v_{\text{Вход}}$ — результат константной передаточной функции, представляющей начальный входной узел. Идеальным результатом для блока B, таким образом, является

$$\text{IDEAL}[B] = \bigwedge_{P\text{—возможный путь от входа к } B} f_P(v_{\text{Вход}})$$

Мы заключаем, что в терминах теоретического частичного порядка решетки $\leqslant$ для рассматриваемой структуры

- любой ответ, больший, чем IDEAL, неверен;
- любое значение, меньшее, чем идеальное, или равное ему, консервативно, т.е. безопасно.

Интуитивно значение, более близкое к идеальному, является более точным.[9] Чтобы увидеть, почему любые решения должны не превосходить идеальное, заметим, что для любого блока любое решение, большее, чем IDEAL, может быть

---

[9]Заметим, что в прямой задаче значение IDEAL $[B]$ соответствует значению IN $[B]$. В обратной задаче, которую мы здесь не рассматриваем, мы бы определили IDEAL $[B]$ как идеальное значение OUT $[B]$.

получено путем игнорирования некоторого пути выполнения, по которому может пойти программа; мы не можем гарантировать, что вдоль этого пути не произойдет что-то, что сделает недействительным все улучшения программы, которые могли бы быть сделаны на основе большего решения. И наоборот, любое решение, меньшее, чем IDEAL, может рассматриваться как включающее некоторые пути, либо не существующие в графе потока, либо существующие, но такие, по которым программа никогда не проследует. Это меньшее решение допускает только те преобразования, которые корректны для всех возможных выполнений программы, но может запрещать некоторые из преобразований, разрешенных решением IDEAL.

Решение сбором по путям

Однако, как говорилось в разделе 9.1, поиск всех возможных путей выполнения — задача неразрешимая. Следовательно, требуется поиск приближенного решения. В абстракции потока данных мы полагаем, что может быть пройден каждый путь в графе потока. Таким образом, мы можем определить решение сбором по путям (meet-over-paths) для B как

$$\text{MOP}[B] = \bigwedge_{P\text{ — путь от входа к } B} f_P(v_{\text{вход}})$$

Заметим, что, как и в случае идеального решения, MOP$[B]$ в прямой структуре дает значения для IN$[B]$; в случае обратного потока MOP$[B]$ представляют собой значения OUT$[B]$.

Рассматриваемые в решении MOP пути представляют собой надмножество всех путей, которые могут быть выполнены. Таким образом, решение MOP собирает вместе не только значения потоков данных всех выполнимых путей, но и дополнительные значения, связанные с путями, которые не могут быть выполнены. Сбор идеального решения и дополнительных членов не может дать решение, большее идеального. Таким образом, для всех B выполняется соотношение MOP$[B] \leqslant$ IDEAL$[B]$.

Максимальная фиксированная точка и решение сбором по путям

Заметим, что в MOP-решении в случае наличия циклов количество рассматриваемых путей остается неограниченным. Таким образом, MOP-определение само по себе не приводит к алгоритму. Итеративный алгоритм не должен начинаться с поиска всех путей, ведущих к базовому блоку, перед тем как применять оператор сбора. Вместо этого

1. итеративный алгоритм посещает базовые блоки не обязательно в порядке их выполнения;

2. в каждой точке слияния алгоритм применяет оператор сбора к значениям потока данных, полученным к этому моменту; некоторые из этих значе-

ний были искусственно введены при инициализации, а не представляют результат выполнения программы от начала к рассматриваемой точке.

Так как же соотносятся решение сбором по путям и решение, получаемое алгоритмом 9.21?

Начнем рассмотрение с порядка обхода узлов. В пределах итерации мы можем посетить базовый блок до посещения его предшественников. Если таковым предшественником является входной узел, OUT [Вход] инициализирован корректным константным значением. Остальные узлы инициализированы значением $\top$, не меньшим, чем окончательный ответ. Монотонность обеспечивает при использовании $\top$ в качестве входных данных получение результата, не меньшего, чем искомое решение. В этом смысле можно говорить о $\top$ как о не представляющем никакой информации.

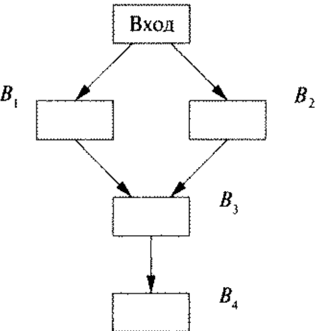


Рис. 9.24. Граф потока, иллюстрирующий влияние раннего сбора по путям

В чем выражается ранее применение оператора сбора? Рассмотрим простой пример, показанный на рис. 9.24, и предположим, что интересующее нас значение — IN $[B_4]$. В соответствии с определением MOP

$$\text{MOP}[B_4] = ((f_{B_3} \circ f_{B_1}) \wedge (f_{B_3} \circ f_{B_2}))(v_{\text{Вход}})$$

В итеративном алгоритме при посещении узлов в порядке B_1, B_2, B_3, B_4

$$\text{IN}[B_4] = f_{B_3}((f_{B_1}(v_{\text{Вход}}) \wedge f_{B_2}(v_{\text{Вход}})))$$

В то время как оператор сбора применяется в конце MOP-определения, итеративный алгоритм применяет его раньше. Ответ при этом получается одинаковым, только если структура потока данных дистрибутивна. Если структура потока данных монотонна, но не дистрибутивна, то IN $[B_4] \leqslant$ MOP $[B_4]$. Вспомним, что в общем случае решение IN $[B]$ безопасно (консервативно), если для всех базовых блоков B IN $[B] \leqslant$ IDEAL $[B]$.

Представим набросок доказательства того, что в общем случае решение максимальной фиксированной точки, получаемое итеративным алгоритмом, всегда безопасно. Простая индукция по i показывает, что значения, полученные после i итераций, не превосходят результата сбора по всем путям длиной i и менее. Однако итеративный алгоритм завершается, только если он достигает того же ответа, что и при неограниченном количестве итераций. Таким образом, результат оказывается не большим, чем MOP-решение. Поскольку MOP $\leqslant$ IDEAL, а MFP $\leqslant$ MOP, мы получаем MFP $\leqslant$ IDEAL, а следовательно, решение максимальной фиксированной точки, получаемое при использовании итеративного алгоритма, является безопасным.

9.3.5 Упражнения к разделу 9.3

Упражнение 9.3.1. Постройте диаграмму решетки для произведения трех решеток, каждая из которых основана на одном определении d_i, $i = 1, 2, 3$. Как ваша диаграмма соотносится с диаграммой на рис. 9.22?

! **Упражнение 9.3.2.** В разделе 9.3.3 мы доказали, что если структура имеет конечную высоту, то итеративный алгоритм сходится. Далее приведен пример, в котором структура не имеет конечной высоты и итеративный алгоритм не является сходящимся. Множество значений V представляет собой неотрицательные целые числа, а оператор сбора — минимум. Имеется три передаточные функции, а именно

1. тождественность $f_I(x) = x$;
2. "половина" $f_H(x) = x/2$;
3. "единица" $f_O(x) = 1$.

Множество передаточных функций F состоит из трех указанных функций и всевозможных их композиций.

а) Опишите множество F.

б) Что собой представляет отношение $\leqslant$ в данной схеме?

в) Приведите пример графа потока с присоединенными передаточными функциями, для которого алгоритм 9.21 не сходится.

г) Является ли данная структура монотонной, дистрибутивной?

! **Упражнение 9.3.3.** Мы доказали, что алгоритм 9.21 сходится, если структура монотонна и имеет конечную высоту. Вот пример структуры, иллюстрирующей важность монотонности (конечности высоты оказывается недостаточно). Область

определения $V = \{1, 2\}$, оператор сбора — минимум, а множество функций F состоит из тождественной функции (f_I) и функции "переключения" ($f_S(x) = 3 - x$), которая меняет 1 на 2 и наоборот.

а) Покажите, что эта структура имеет конечную высоту, но не монотонна.

б) Приведите пример графа потока и назначения передаточных функций, такой, что алгоритм 9.21 не сходится.

! **Упражнение 9.3.4.** Пусть $\text{MOP}_i[B]$ — сбор по всем путям длиной i или менее от входа до базового блока B. Докажите, что после i итераций алгоритма 9.21 $\text{IN}[B] \leqslant \text{MOP}_i[B]$. Покажите также как следствие, что если алгоритм 9.21 сходится, то он сходится к результату, который не превышает (в смысле отношения $\leqslant$) MOP-решение.

! **Упражнение 9.3.5.** Предположим, что множество функций F структуры содержит только функции вида *gen-kill*, т.е. область определения V является показательным множеством некоторого множества, а $f(x) = G \cup (x - K)$ для некоторых множеств G и K. Докажите, что если оператор сбора представляет собой *а*) объединение или *б*) пересечение, то описанная структура дистрибутивна.

9.4 Распространение констант

Все рассматривавшиеся в разделе 9.2 схемы в действительности представляют собой простые примеры дистрибутивных структур с конечной высотой. Таким образом, применение к ним итеративного алгоритма 9.21 как в прямом, так и в обратном направлениях, приводит в каждом случае к MOP-решению. В этом разделе мы поближе познакомимся с полезной структурой потока данных, обладающей более интересными свойствами.

Вспомним, что распространение констант (constant propagation), или "дублирование констант" (constant folding), заменяет выражения, которые при выполнении всякий раз вычисляют одну и ту же константу, самой этой константой. Описанная ниже структура распространения констант отличается от уже рассматривавшихся задач потоков данных тем, что

а) имеет неограниченное множество возможных значений потока данных даже для фиксированного графа потока и

б) не является дистрибутивной.

Распространение констант является прямой задачей потока данных. Полурешетка, представляющая значения потока данных, и семейство передаточных функций будут представлены далее.

9.4.1 Значения потока данных для структуры распространения констант

Множество значений потока данных представляет собой решетку произведения, в которой имеется по одному компоненту для каждой переменной программы. В решетку для единственной переменной входит следующее.

1. Все константы подходящего для данной переменной типа.

2. Значение NAC, означающее "не константу" (not-a-constant). Переменная отображается на это значение, если выясняется, что она не имеет константного значения. Это может происходить потому, что переменной может быть присвоено входное значение либо она может быть вычислена из переменной, не являющейся константой, а также на разных путях, приводящих к одной и той же точке программы, ей могут присваиваться разные константы.

3. Значение UNDEF, которое означает "не определена" (undefined). Переменная получает это значение, если пока что о ней ничего нельзя утверждать наверняка; вероятно, пока что не найдено определение, достигающее рассматриваемой точки.

Заметим, что NAC и UNDEF — это не одно и то же, это существенно разные вещи. NAC говорит о том, что имеется много путей определения переменной, так что мы знаем, что она не может быть константой; UNDEF же говорит о том, что мы знаем о переменной слишком мало, чтобы считать ее чем бы то ни было.

Полурешетка для типичной целочисленной переменной показана на рис. 9.25. Ее верхним элементом является UNDEF, а нижним — NAC, т.е. наибольшим значением в частичном порядке является UNDEF, а наименьшим — NAC. Значения констант неупорядочены, но все они меньше UNDEF и больше NAC. Как говорилось в разделе 9.3.1, сбор двух значений представляет собой их наибольшую нижнюю границу. Таким образом, для всех значений v

$$\text{UNDEF} \wedge v = v \text{ и } \text{NAC} \wedge v = \text{NAC}$$

Для любой константы c

$$c \wedge c = c$$

Для двух различных констант c_1 и c_2

$$c_1 \wedge c_2 = \text{NAC}$$

Значение потока данных в этой структуре представляет собой отображение каждой переменной программы на одно из значений в полурешетке констант. Значение переменной v в отображении m обозначается как $m(v)$.

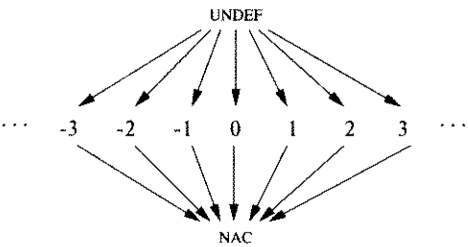


Рис. 9.25. Полурешетка, представляющая возможные "значения" одной целочисленной переменной

9.4.2 Сбор в структуре распространения констант

Полурешетка значений потока данных представляет собой произведение полурешеток наподобие показанной на рис. 9.25, по одной для каждой переменной. Таким образом, $m \leqslant m'$ тогда и только тогда, когда для всех переменных v $m(v) \leqslant m'(v)$. Или $m \wedge m' = m''$, если для всех переменных v выполняется соотношение $m''(v) = m(v) \wedge m'(v)$.

9.4.3 Передаточные функции для структуры распространения констант

Далее мы полагаем, что базовый блок содержит только одну инструкцию. Передаточные функции для базовых блоков, состоящих из нескольких инструкций, могут быть построены путем композиции функций, соответствующих отдельным инструкциям. Множество F состоит из некоторых передаточных функций, которые принимают отображения переменных на значения в решетке констант и возвращают другие отображения данного вида.

F содержит тождественную функцию, которая получает в качестве аргумента и возвращает одно и то же отображение. Кроме того, F содержит константную передаточную функцию для входного узла. Эта передаточная функция возвращает, независимо от полученного аргумента, отображение m_0, где $m_0(v) =$ UNDEF для всех переменных v. Такое граничное условие не лишено смысла, поскольку перед началом выполнения программы нет определения ни одной переменной.

В общем случае пусть f_s — передаточная функция для инструкции s и пусть m и m' представляют значения потока данных, такие, что $m' = f_s(m)$. Мы опишем f_s в терминах отношений между m и m'.

1. Если s не является инструкцией присваивания, то f_s представляет собой тождественную функцию.

2. Если s — присваивание переменной x, то для всех переменных $v \neq x$ выполняется соотношение $m'(v) = m(v)$, где $m'(x)$ определяется следующим образом.

   а) Если в правой части инструкции s находится константа c, то $m'(x) = c$.

   б) Если в правой части инструкции s находится выражение типа $y + z$[10], то

$$m'(x) = \begin{cases} m(y) + m(z), & \text{если } m(y) \text{ и } m(z) \text{ — константные значения,} \\ \text{NAC}, & \text{если либо } m(x)\text{, либо } m(y) \text{ является NAC,} \\ \text{UNDEF} & \text{в противном случае.} \end{cases}$$

   в) Если в правой части инструкции s находится любое иное выражение (например, вызов функции или присваивание посредством указателя), то $m'(x) = \text{NAC}$.

9.4.4 Монотонность структуры распространения констант

Покажем, что структура распространения констант монотонна. Во-первых, можно рассмотреть влияние функции f_s на единственную переменную. Во всех случаях, кроме 2, *б*, f_s либо не изменяет значение $m(x)$, либо изменяет отображение на возвращающее константу или NAC. Понятно, что в этих случаях f_s является монотонной.

В случае 2, *б* влияние f_s протабулировано на рис. 9.26. В первом и втором столбцах представлены возможные входные значения y и z; в последнем столбце показаны выходные значения x. Значения в каждом столбце или подстолбце упорядочены от наибольшего к наименьшему. Чтобы показать, что функция монотонна, убедимся, что для каждого из возможных значений y значение x не возрастает с уменьшением значения z. Например, если y представляет собой константное значение c_1, а значение z изменяется от UNDEF через c_2 к NAC, то значение x соответственно изменяется от UNDEF через $c_1 + c_2$ к NAC. Эту проверку можно выполнить для всех значений y. Впрочем, в силу симметрии нам даже не требуется повторять процедуру для второго операнда, чтобы сделать заключение о том, что выходное значение не может возрастать при уменьшении входного.

---

[10]Как обычно, + представляет обобщенный оператор, а не обязательно сложение.

$m(y)$	$m(z)$	$m'(x)$
UNDEF	UNDEF	UNDEF
	c_2	UNDEF
	NAC	NAC
c_1	UNDEF	UNDEF
	c_2	$c_1 + c_2$
	NAC	NAC
NAC	UNDEF	NAC
	c_2	NAC
	NAC	NAC

Рис. 9.26. Передаточная функция распространения констант для x = y+z

9.4.5 Недистрибутивность структуры распространения констант

Структура распространения констант монотонна, но не дистрибутивна, т.е. итеративное MFP-решение безопасно, но может быть меньше MOP-решения. Приведенный далее пример доказывает недистрибутивность рассматриваемой структуры.

Пример 9.22. В программе на рис. 9.27 x и y устанавливаются равными 2 и 3 в базовом блоке B_1, и 3 и 2 соответственно в базовом блоке B_2. Мы знаем, что независимо от выбранного пути значение z в конце блока B_3 равно 5. Однако итеративный алгоритм не сможет обнаружить этот факт. Он применит оператор сбора на входе B_3 и получит значения x и y, равные NAC. Поскольку сложение двух NAC дает NAC, алгоритм 9.21 на выходе из программы даст $z =$ NAC. Этот результат безопасен, но не точен. Алгоритм 9.21 не в состоянии отследить корреляцию, заключающуюся в том, что когда x равно 2, то y равно 3, и наоборот. Можно, но гораздо более высокой ценой, использовать более сложную структуру, которая отслеживает все возможные равенства, которые могут выполняться между парами выражений, включающими переменные программы. Такой подход рассматривается в упражнении 9.4.2.

Можно объяснить эту потерю точности недистрибутивностью структуры распространения констант теоретически. Пусть f_1, f_2 и f_3 — передаточные функции, представляющие блоки B_1, B_2 и B_3 соответственно. Как показано на рис. 9.28,

$$f_3(f_1(m_0) \wedge f_2(m_0)) < f_3(f_1(m_0)) \wedge f_3(f_2(m_0))$$

А это показывает недистрибутивность рассматриваемой структуры. □

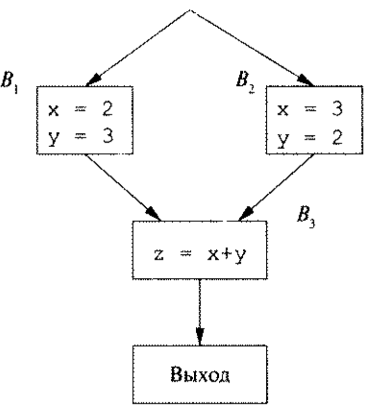


Рис. 9.27. Пример, демонстрирующий, что структура распространения констант недистрибутивна

m	$m(x)$	$m(y)$	$m(z)$
m_0	UNDEF	UNDEF	UNDEF
$f_1(m_0)$	2	3	UNDEF
$f_2(m_0)$	3	2	UNDEF
$f_1(m_0) \wedge f_2(m_0)$	NAC	NAC	UNDEF
$f_3(f_1(m_0) \wedge f_2(m_0))$	NAC	NAC	NAC
$f_3(f_1(m_0))$	2	3	5
$f_3(f_2(m_0))$	3	2	5
$f_3(f_1(m_0)) \wedge f_3(f_2(m_0))$	NAC	NAC	5

Рис. 9.28. Пример недистрибутивности передаточных функций

9.4.6 Интерпретация результатов

Значение UNDEF используется в итеративном алгоритме с двумя целями: для инициализации входного узла и инициализации внутренних точек программы перед итерациями. Смысл этих двух случаев немного различен. В первом случае утверждается, что в начале выполнения программы значения переменных не определены; во втором — что из-за недостатка информации в начале итеративного процесса мы аппроксимируем решение верхним элементом UNDEF. В конце итеративного процесса переменные на выходе из входного блока сохраняют значение UNDEF, поскольку OUT [ВХОД] никогда не изменяется.

Значения UNDEF могут оказаться у переменной и в других точках программы. Если это происходит, значит, для данной переменной не нашлось определений на

путях, ведущих к данной точке программы. Заметим, что в силу определения оператора сбора при наличии хотя бы одного пути, на котором имеется определение переменной, достигающее рассматриваемой точки программы, эта переменная не будет иметь значение UNDEF. Если все определения, достигающие точки программы, имеют одно и то же константное значение, то переменная рассматривается как константа, даже если она может оказаться не определена на некотором пути.

Полагая, что программа корректна, алгоритм может найти большее количество констант, т.е. для повышения эффективности программы алгоритм выбирает некоторые значения для переменных, которые могут быть не определенными. Такое изменение правомерно в большинстве языков программирования, поскольку не определенная переменная может принимать любое значение. Если семантика языка требует, чтобы все не определенные переменные принимали некоторое конкретное значение, то мы должны соответствующим образом изменить формулировку задачи. Если же нас интересует поиск в программе переменных, которые могут быть не определены, то для получения такого результата можно переформулировать задачу анализа потока данных (см. упражнение 9.4.1).

Пример 9.23. На рис. 9.29 значения x на выходе из базовых блоков B_2 и B_3 — соответственно 10 и UNDEF. Поскольку UNDEF $\wedge$ $10 = 10$, значение x на входе в блок B_4 равно 10. Таким образом, в блоке B_5, где используется x, его можно заменить на 10. Если выполнение пойдет по пути $B_1 \to B_3 \to B_4 \to B_5$, то значение x, достигающее базового блока B_5, будет не определенным. Таким образом, представляется некорректным замена использования x на 10.

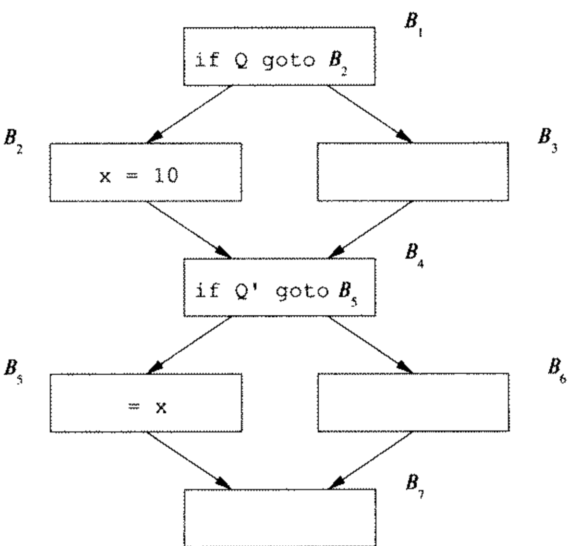


Рис. 9.29. Сбор UNDEF и константы

Однако если предикат Q не может быть ложен, если истинен предикат Q', то такой путь никогда не будет пройден. Программист может знать об этом, но такой вывод может оказаться за пределами возможностей анализа потоков данных. Таким образом, если считать, что программа корректна и что все переменные определены до их использования, то вполне корректно и то, что значение x при входе в базовый блок B_5 может принимать только значение 10. Если программа некорректна, то выбрать 10 в качестве значения x — в любом случае не хуже, чем позволить этой переменной принимать некоторое случайное значение. □

9.4.7 Упражнения к разделу 9.4

Упражнение 9.4.1. Предположим, что мы хотим обнаружить все переменные, которые могут быть не инициализированными вдоль некоторого пути к рассматриваемой точке. Каким образом следует модифицировать структуру из этого раздела для поиска таких ситуаций?

Упражнение 9.4.2. Интересная и мощная структура анализа потока данных получается, если представить, что область определения V включает все возможные разбиения выражений, так что два выражения принадлежат одному и тому же классу тогда и только тогда, когда они имеют одинаковые значения вдоль любого пути к рассматриваемой точке. Чтобы избежать перечисления бесконечного количества выражений, можно представить V при помощи перечисления только минимальных пар эквивалентных выражений. Например, если мы выполняем инструкции

```
a = b
c = a + d
```

то минимальное множество эквивалентностей представляет собой $\{a \equiv b, c \equiv a+d\}$. Отсюда следуют прочие эквивалентности, такие как $c \equiv b+d$ и $a+e \equiv b+e$, но их не надо перечислять явно.

а) Каким должен быть оператор сбора для такой структуры?

б) Разработайте структуру данных для представления значений области определения и алгоритм реализации оператора сбора.

в) Какие функции связаны с инструкциями? Какое влияние должна оказывать инструкция наподобие `a = b+c` на разбиение выражений (т.е. на значение в V)?

г) Является ли данная структура монотонной, дистрибутивной?

9.5 Устранение частичной избыточности

В этом разделе мы подробно рассмотрим минимизацию количества вычислений выражений, т.е. рассмотрим все возможные последовательности выполнения в графе потока и узнаем, сколько раз вычисляется выражение, подобное $x + y$. Выполняя перемещения кода и при необходимости сохраняя результат во временной переменной, зачастую можно снизить количество вычислений такого выражения вдоль многих путей выполнения, при этом ни по одному из возможных путей количество вычислений не увеличится. Заметим, что количество различных мест, где выполняется вычисление $x + y$, может и увеличиться, но это относительно неважно, поскольку снизится количество *вычислений* выражения $x + y$.

Применение разработанного в этом разделе преобразования кода повышает производительность получающегося кода, поскольку, как вы увидите, операция никогда не применяется, если только она не абсолютно необходима. Каждый оптимизирующий компилятор реализует нечто наподобие рассматриваемого здесь преобразования, даже если при этом использует "менее агрессивный" алгоритм по сравнению с рассматривающимся здесь. Однако имеется и иная причина рассмотреть эту задачу. Поиск места или мест в графе потока для вычисления каждого выражения требует четырех различных видов анализа потока данных. Таким образом, изучение "устранения частичной избыточности", как называется минимизация количества вычислений выражений, существенно способствует пониманию той роли, которую анализ потоков данных играет в компиляторе.

Избыточность в программе существует в различных формах. Как говорилось в разделе 9.1.4, она может быть в виде общих подвыражений, когда несколько вычислений выражения дают одно и то же значение. Она может быть и в виде выражения, инвариантного относительно цикла, которое вычисляет одно и то же значение на каждой итерации цикла. Избыточность может быть частичной, если она обнаруживается вдоль некоторых, но *не всех*, путей. Общие подвыражения и выражения, инвариантные относительно цикла, можно рассматривать как частные случаи частичной избыточности; таким образом, для устранения различных видов избыточности можно разработать единый алгоритм устранения частичной избыточности.

Далее мы рассмотрим различные типы избыточности, а затем поставим обобщенную задачу устранения частичной избыточности и представим алгоритм ее решения. Этот алгоритм представляет особый интерес, поскольку включает решение нескольких задач потоков данных как в прямом, так и в обратном направлениях.

9.5.1 Источники избыточности

На рис. 9.30 проиллюстрированы три вида избыточности: общие подвыражения, выражения, инвариантные по отношению к циклу, и частично избыточные выражения. На рисунке показан код как до, так и после выполнения каждого вида оптимизации.

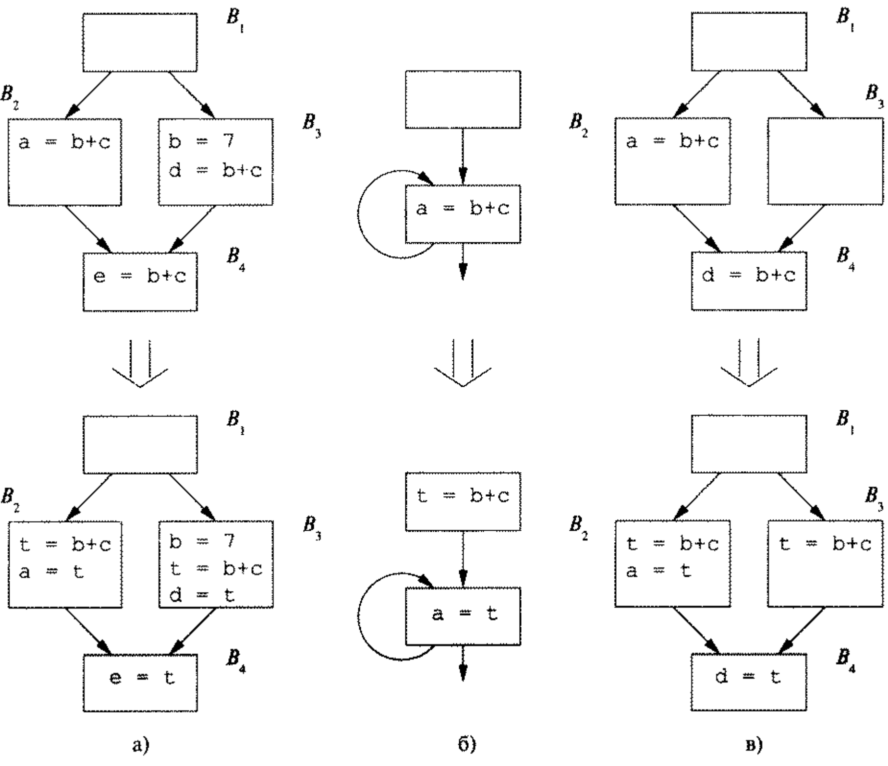


Рис. 9.30. Примеры а) глобального общего подвыражения; б) перемещения кода, инвариантного по отношению к циклу; в) устранения частичной избыточности

Глобальные общие подвыражения

На рис. 9.30, *а* выражение $b + c$, вычисляемое в блоке B_4, избыточно: оно уже вычислено при достижении блока B_4, независимо от того, каким именно путем поток управления достигает блока B_4. Как видно из приведенного примера, значение выражения может отличаться на разных путях. Можно оптимизировать приведенный код, сохраняя результат вычисления $b + c$ в блоках B_2 и B_3 в одной и той же временной переменной, скажем, t, и вместо повторного вычисления выражения $b + c$ присваивая значение t переменной e в блоке B_4. При наличии

Поиск "глубоких" общих подвыражений

Анализ доступных выражений для определения избыточных выражений работает только с текстуально идентичными выражениями. Например, применение устранения общих подвыражений распознает, что `t1` в фрагменте кода

```
t1 = b + c; a = t1 + d;
```

имеет то же значение, что и `t2` в

```
t2 = b + c; e = t2 + d;
```

при условии, что переменные b и c не изменялись между этими фрагментами. Однако данный анализ не в состоянии определить идентичность a и e. Найти такие "глубокие" общие выражения можно путем повторного применения устранения общих подвыражений до тех пор, пока на очередной итерации не будет найдено ни одного нового общего подвыражения. Можно также воспользоваться структурой из упражнения 9.4.2 для поиска "глубоких" подвыражений.

присваивания переменным b или c после последнего вычисления $b + c$, но до блока B_4 выражение в блоке B_4 избыточным не является.

Формально мы говорим, что выражение $b + c$ является (полностью) *избыточным* в точке p, если в этой точке имеется доступное в смысле раздела 9.2.6 выражение. Иначе говоря, выражение $b + c$ вычисляется вдоль всех путей, достигающих p, и переменные b и c не переопределяются после того, как вычислено последнее из выражений. Условие неизменности b и c является необходимым, потому что, хотя выражение $b + c$ выполняется перед достижением точки p текстуально, значение $b + c$, вычисленное в точке p, может оказаться иным из-за возможного изменения операндов.

Выражения, инвариантные относительно циклов

На рис. 9.30, *б* показан пример выражения, инвариантного относительно цикла. Выражение $b + c$ является инвариантом в предположении, что в цикле не изменяются ни переменная b, ни переменная c. Эту программу можно оптимизировать путем замены таких повторных вычислений единственным вычислением за пределами цикла. Результаты вычисления присваиваются временной переменной, скажем, t, а затем выражение в цикле заменяется переменной t. Здесь есть один момент, на который следует обратить особое внимание при таком "перемещении кода". Мы не должны выполнять ни одну команду, которая не выполняется без оп-

тимизации. Например, если можно выйти из цикла без выполнения инструкции, являющейся инвариантом относительно цикла, то такая инструкция не должна выноситься за пределы цикла. Тому есть две причины.

1. Если инструкция генерирует исключение, то в результате мы получим генерацию исключения там, где его могло бы и не быть.

2. При выходе из цикла без вычисления "оптимизированная" программа оказывается менее эффективной, чем исходная.

Чтобы гарантировать возможность оптимизации выражений, являющихся инвариантами относительно циклов while, компилятор обычно представляет инструкцию

```
while c { S; }
```

так же, как и инструкцию

```
if c { repeat S; until not c; }
```

При этом выражение, являющееся инвариантом относительно цикла, может быть вынесено и располагаться непосредственно перед конструкцией repeat-until.

В отличие от устранения общих подвыражений, когда вычисление избыточного выражения просто опускается, устранение выражений, инвариантных относительно циклов, выполняет их перенос изнутри цикла наружу. Поэтому такая оптимизация известна как перемещение кода, инвариантного относительно цикла (loop-invariant code motion). Эту оптимизацию может потребоваться применить повторно, поскольку когда выясняется, что переменная содержит значение, инвариантное относительно цикла, то таковыми же могут оказаться и выражения, использующие эту переменную.

Частично избыточные выражения

Пример частично избыточного выражения приведен на рис. 9.30, *в*. Выражение $b+c$ в блоке B_4 избыточно на пути $B_1 \to B_2 \to B_4$, но не на пути $B_1 \to B_3 \to B_4$. Можно устранить избыточность в первом пути, разместив вычисление $b+c$ в блоке B_3. Все результаты вычисления $b + c$ записываются во временную переменную t, а вычисление в блоке B_4 заменяется переменной t. Таким образом, подобно перемещению кода, инвариантного относительно цикла, устранение частичной избыточности требует размещения новых вычислений выражения.

9.5.2 Все ли избыточные вычисления могут быть устранены?

Можно ли устранить все избыточные вычисления вдоль каждого пути? Ответ на этот вопрос отрицателен, если только мы не допускаем изменения графа потока путем создания новых блоков.

Пример 9.24. В примере, показанном на рис. 9.31, *а*, вычисление выражения $b+c$ в блоке B_4 избыточно, если программа следует по пути $B_1 \to B_2 \to B_4$. Однако мы не можем просто переместить вычисление $b + c$ в блок B_3, поскольку это действие приведет к излишнему вычислению $b + c$ на пути $B_1 \to B_3 \to B_5$.

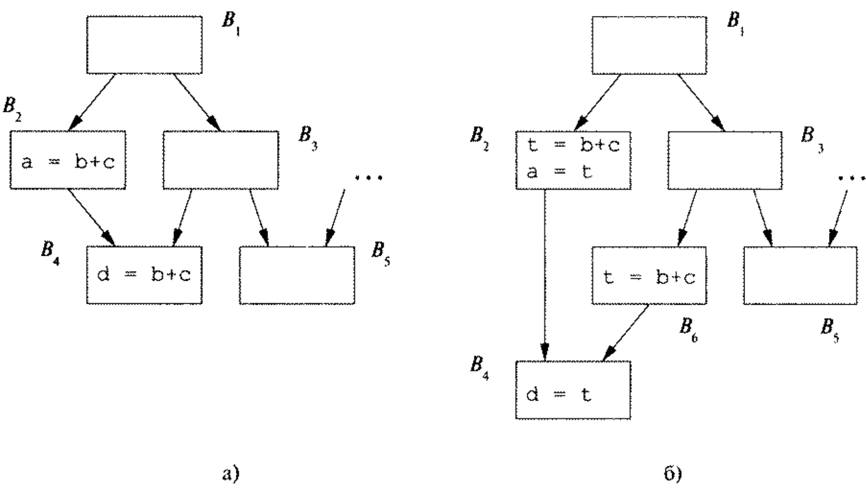


Рис. 9.31. $B_3 \to B_4$ является критическим ребром

Было бы хорошо, если бы можно было вставить вычисление $b+c$ только вдоль ребра от блока B_3 к блоку B_4. Это можно сделать, поместив инструкцию в новый блок, скажем, B_6, и передавая управление от блока B_3 блоку B_6 при переходе к блоку B_4. Это преобразование показано на рис. 9.31, *б*. □

Определим *критическое ребро* графа потока как ребро, идущее от узла с более чем одним преемником к узлу с более чем одним предшественником. Введение новых блоков вдоль критических ребер позволяет нам всегда найти блок для размещения интересующего нас выражения. Например, ребро от B_3 к B_4 на рис. 9.31, *а* является критическим, поскольку у блока B_3 два преемника, а у блока B_4 — два предшественника.

Добавления блоков может оказаться недостаточным для того, чтобы устранить все избыточные выражения. Как показано в примере 9.25, может потребоваться дублирование кода для того, чтобы выделить путь, в котором обнаружена избыточность.

Пример 9.25. В примере, показанном на рис. 9.32, *а*, выражение $b + c$ избыточно вычисляется на пути $B_1 \to B_2 \to B_4 \to B_6$. Мы хотели бы устранить излишнее вычисление $b + c$ в блоке B_6 на этом пути и вычислять это выражение только на пути $B_1 \to B_3 \to B_4 \to B_6$. Однако не существует единственной точки (или ребра) в исходной программе, единственным образом соответствующей данному

пути. Для создания такой точки программы мы можем дублировать пару блоков B_4 и B_6 так, что одна пара достижима по пути, проходящему через B_2, а вторая пара — по пути, проходящему через B_3, как показано на рис. 9.32, *б*. Результат $b+c$ сохраняется в переменной t в блоке B_2 и перемещается в переменную d в блоке B_6', копии B_6, достигаемой из B_2. □

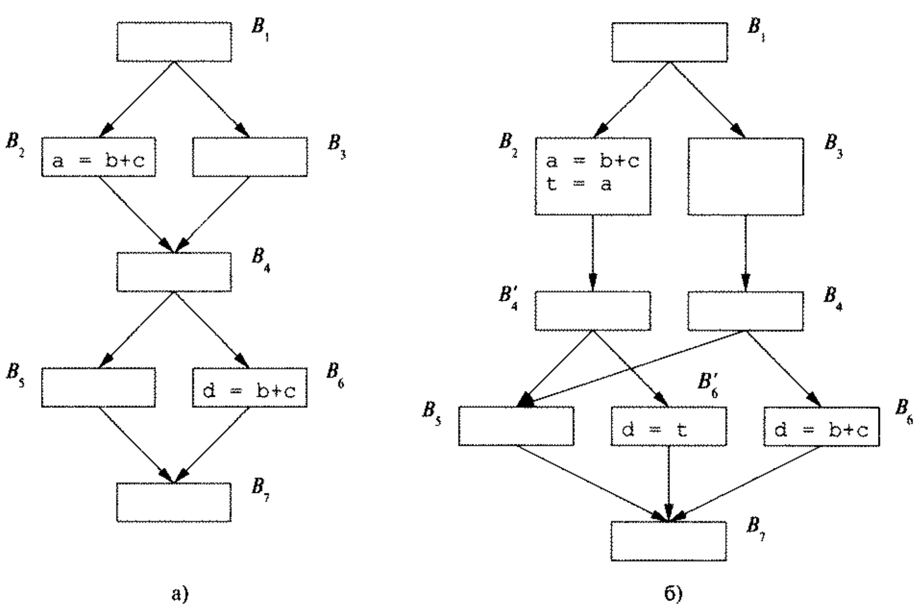


Рис. 9.32. Дублирование кода

Поскольку количество путей экспоненциально зависит от количества условных ветвлений в программе, устранение всех избыточных вычислений может существенно увеличить размер оптимизированного код. Поэтому ограничимся в нашем рассмотрении методов устранения избыточности теми, которые могут вносить дополнительные блоки, но не дублируют части графа потока управления.

9.5.3 Отложенное перемещение кода

Желательно, чтобы оптимизированная при помощи алгоритма устранения частичной избыточности программа обладала следующими свойствами.

1. Все избыточные вычисления выражений, которые можно удалить без дублирования кода, должны быть удалены.

2. Оптимизированная программа не выполняет никаких вычислений, которые не выполняются при выполнении исходной программы.

3. Все выражения вычисляются в последний возможный момент.

Последнее свойство является важным в силу того, что значения выражений, являющихся избыточными, обычно хранятся до их использования в регистрах. Максимально позднее вычисление значения минимизирует его время жизни – промежуток времени между его определением и временем его последнего использования, что минимизирует использование им регистра. Будем говорить об оптимизации устранения частичной избыточности с максимально возможной задержкой вычислений как об *отложенном перемещении кода* (lazy code motion).

Для улучшения интуитивного понимания задачи рассмотрим сначала частичную избыточность одного выражения вдоль одного пути. Для удобства при рассмотрении будем считать, что каждая инструкция представляет собой базовый блок, состоящий из одной этой инструкции.

Полная избыточность

Выражение e в блоке B является избыточным, если e вычисляется вдоль всех путей, достигающих B, и впоследствии его операнды не переопределяются. Пусть S – множество блоков, каждый из которых содержит выражение e и которые делают e в B избыточным. Множество ребер, покидающих блоки из S, с необходимостью образует *разрез* (cutset), который, будучи удален, отсоединит блок B от входа в программу. Более того, вдоль путей, ведущих из блоков из S к B, не переопределяются никакие операнды e.

Частичная избыточность

Если выражение e в блоке B только частично избыточно, алгоритм отложенного перемещения кода пытается сделать e в B полностью избыточным, помещая дополнительные копии выражения в графе потока. Если попытка успешна, оптимизированный граф потока также имеет множество базовых блоков S, каждый из которых содержит выражение e и выходящие ребра которых являются разрезом между входом и блоком B. Аналогично случаю полной избыточности никакие операнды e не переопределяются вдоль путей, которые идут из блоков из S к B.

9.5.4 Ожидаемость выражений

Существует дополнительное ограничение, накладываемое на вносимые выражения, заключающееся в том, что никакие дополнительные операции не должны выполняться. Копии выражений должны помещаться в точках программы, в которых выражение является *ожидаемым* (anticipated). Мы говорим, что выражение $b + c$ является *ожидаемым* в точке p, если все пути, ведущие из p, в конечном счете вычисляют значение выражения $b + c$ на основе значений b и c, доступных в этой точке.

Рассмотрим устранение частичной избыточности вдоль ациклического пути $B_1 \to B_2 \to \cdots \to B_n$. Предположим, что выражение e вычисляется только в блоках B_1 и B_n и что операнды e не переопределяются в блоках вдоль указанного пути. Существуют входящие ребра, которые присоединяются к этому пути, и выходящие ребра, покидающие этот путь. Выражение e *не* является ожидаемым при входе в блок B_i тогда и только тогда, когда существует выходящее ребро, покидающее блок B_j, $i \leqslant j < n$, приводящее к пути выполнения, не использующему значение e. Таким образом, ожидаемость ограничивает местоположения, в которые может быть вставлено выражение.

Можно создать разрез, который включает ребро $B_{i-1} \to B_i$ и делает e избыточным в B_n, если e либо доступно, либо ожидаемо на входе в B_i. Если e ожидаемо, но не доступно на входе в B_i, необходимо разместить копию выражения e на входящем ребре.

У нас есть выбор места размещения копий выражения, поскольку обычно существует несколько разрезов в графе потока, удовлетворяющих всем требованиям. В изложенном выше материале вычисления добавляются в ребро, входящее в интересующий нас путь, и, таким образом, выражение вычисляется как можно ближе к использованию без внесения избыточности. Заметим, что эти добавленные операции сами по себе могут быть частично избыточными по отношению к другим экземплярам того же выражения в программе. Такая частичная избыточность может быть устранена путем дальнейшего перемещения вычислений.

Итак, ожидаемость выражений ограничивает местоположения, в которые может быть помещено выражение; выражение не может быть размещено ранее, чем оно становится ожидаемым. Чем раньше размещается выражение, тем бóльшая избыточность может быть устранена, а среди всех решений, устраняющих одни и те же избыточности, то, которое вычисляет выражение как можно позже, минимизирует время жизни регистров, хранящих значения этих выражений.

9.5.5 Алгоритм отложенного перемещения кода

Приведенный материал приводит нас к четырехшаговому алгоритму. На первом шаге используется ожидаемость для определения местоположений, в которые могут быть помещены выражения. На втором шаге выполняется поиск *наиболее раннего* разреза среди всех, которые устраняют как можно большее количество избыточных вычислений без дублирования кода и без внесения нежелательных вычислений. На этом шаге вычисления размещаются в точках программы, где значения их результатов впервые становятся ожидаемыми. На третьем шаге разрез перемещается вниз к точке, где любые задержки будут приводить либо к изменениям семантики программы, либо к появлению избыточности. Четвертый, последний шаг представляет собой простой проход для очистки кода путем удаления присваиваний временным переменным, используемым только один раз. Каждый

шаг завершается проходом по потоку данных: второй и третий шаги представляют собой задачи в прямом направлении потока данных, а первый и четвертый шаги — в направлении, обратном направлению потока данных.

Обзор алгоритма

1. Найти все выражения, ожидаемые в каждой точке программы, с использованием прохода в направлении, обратном потоку данных.

2. Второй шаг размещает вычисление там, где значения выражений впервые становятся ожидаемыми вдоль некоторого пути. После размещения копий выражения в местах, где выражение впервые становится ожидаемым, это выражение будет *доступным* в точке p программы, если оно было ожидаемо вдоль всех путей, достигающих p. Доступность может быть разрешена с использованием прохода в прямом направлении потока данных. Чтобы разместить выражения в как можно более ранних позициях, следует просто найти те точки программы, в которых выражения ожидаемы, но не доступны.

3. Выполнение выражения, как только оно становится ожидаемым, может дать значение задолго до его использования. Выражение *откладываемо* (postponable) в точке программы, если оно ожидаемо, но еще не было использовано ни на каком пути, ведущем к этой точке. Найти откладываемые выражения можно с использованием прохода в прямом направлении потока данных. Выражения размещаются в тех точках программы, где они перестают быть откладываемыми.

4. Простой заключительный проход в направлении, обратном потоку данных, используется для устранения присваиваний временным переменным, которые используются в программе только один раз.

Предварительные шаги

Теперь представим алгоритм отложенного перемещения кода полностью. Чтобы упростить задачу, полагаем, что изначально каждая инструкция содержится в базовом блоке, состоящем из одной этой инструкции, а новые вычисления выражений вносятся только в начало блока. Чтобы гарантировать, что такое упрощение не снижает эффективность алгоритма, мы вставляем новый блок между источником и приемником ребра, если приемник имеет более одного предшественника. Очевидно, что таким образом обрабатываются все критические ребра программы.

Семантика каждого блока B абстрагируется с использованием двух множеств: множество e_use_B представляет собой множество выражений, вычисляемых в B, а e_kill_B — множество уничтоженных выражений, т.е. выражений, операнды которых определяются в B. Пример 9.26 будет использоваться в процессе рассмот-

рения четырех анализов потоков данных, определения которых подытожены на рис. 9.34.

Пример 9.26. В графе потока на рис. 9.33, *а* выражение $b + c$ появляется три раза. Поскольку блок B_9 является частью цикла, выражение может вычисляться многократно. Вычисление в блоке B_9 не только инвариантно относительно цикла; оно также является избыточным, поскольку его значение уже использовалось в блоке B_7. В данном примере мы должны вычислять $b + c$ только дважды: один раз — в блоке B_5 и второй — вдоль пути после B_2, но до B_7. Алгоритм отложенного перемещения кода помещает вычисления выражения в начале блоков B_4 и B_5. □

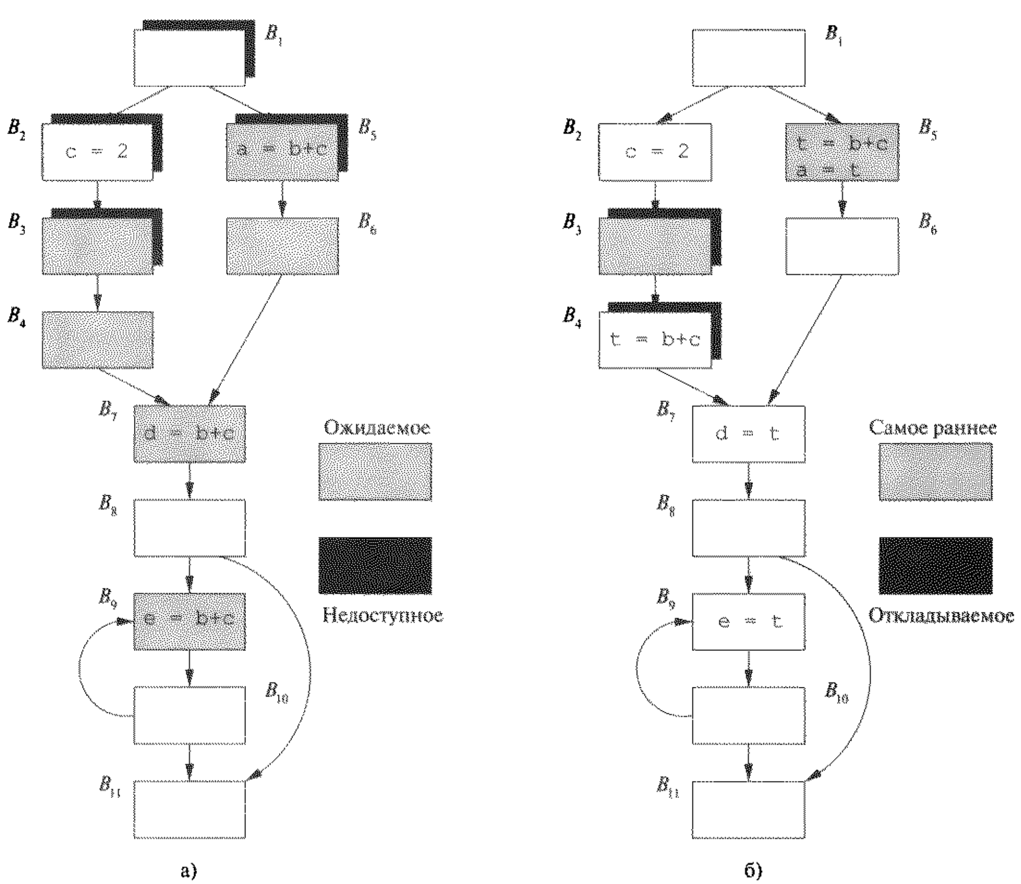


Рис. 9.33. Граф потока к примеру 9.26

	а) Ожидаемые выражения	б) Доступные выражения
Область определения	Множества выражений	Множества выражений
Направление	Обратное	Прямое
Передаточная функция	$f_B(x) = e_use_B \cup (x - e_kill_B)$	$f_B(x) = (anticipated[B].in \cup x) - e_kill_B$
Граничное условие	$\text{IN}[\text{Выход}] = \varnothing$	$\text{OUT}[\text{Вход}] = \varnothing$
Оператор сбора ($\wedge$)	$\cap$	$\cap$
Уравнения	$\text{IN}[B] = f_B(\text{OUT}[B])$ $\text{OUT}[B] = \wedge_{S,succ(B)} \text{IN}[S]$	$\text{OUT}[B] = f_B(\text{IN}[B])$ $\text{IN}[B] = \wedge_{P,pred(B)} \text{OUT}[P]$
Инициализация	$\text{IN}[B] = U$	$\text{OUT}[B] = U$

	в) Откладываемые выражения	г) Используемые выражения
Область определения	Множества выражений	Множества выражений
Направление	Прямое	Обратное
Передаточная функция	$f_B(x) = (earliest[B] \cup x) - e_use_B$	$f_B(x) = (e_use_B \cup x) - latest[B]$
Граничное условие	$\text{OUT}[\text{Вход}] = \varnothing$	$\text{IN}[\text{Выход}] = \varnothing$
Оператор сбора ($\wedge$)	$\cap$	$\cup$
Уравнения	$\text{OUT}[B] = f_B(\text{IN}[B])$ $\text{IN}[B] = \wedge_{P,pred(B)} \text{OUT}[P]$	$\text{IN}[B] = f_B(\text{OUT}[B])$ $\text{OUT}[B] = \wedge_{S,succ(B)} \text{IN}[S]$
Инициализация	$\text{OUT}[B] = U$	$\text{IN}[B] = \varnothing$

$$earliest[B] = anticipated[B].in - available[B].in$$

$$latest[B] = (earliest[B] \cup postponable[B].in) \cap$$

$$\cap \left(e_use_B \cup \neg \left(\bigcap_{S,succ(B)} (earliest[S] \cup postponable[S].in)\right)\right)$$

Рис. 9.34. Четыре прохода для устранения частичной избыточности

Заполнение квадрата

Ожидаемые выражения (иногда называемые в литературе очень занятыми выражениями (very busy expressions)) представляют собой тип анализа потока данных, с которым мы ранее не сталкивались. Мы уже встречались с обратными структурами, такими как анализ живых переменных (раздел 9.2.5), и со структурами, оператором сбора в которых являлось пересечение, такими как доступные выражения (раздел 9.2.6), но сейчас мы впервые столкнулись с анализом, обладающим обоими этими свойствами. Почти все анализы, с которыми мы сталкивались, можно поместить в квадратную таблицу, разбив их на четыре группы — в зависимости от направления (прямое или обратное) и оператора сбора (объединение или пересечение). Заметим также, что анализы с объединением сводятся к вопросу о существовании пути, вдоль которого истинно некоторое условие, в то время как анализы с пересечением выясняют, истинно ли некоторое условие вдоль всех возможных путей.

Ожидаемые выражения

Вспомним, что выражение $b + c$ является ожидаемым в точке p программы, если все пути, ведущие из точки p, в конечном счете вычисляют значение $b + c$ с использованием значений b и c, доступных в этой точке.

На рис. 9.33, *а* все блоки, на входе в которые ожидается $b + c$, выделены серым цветом. Выражение $b + c$ ожидаемо в блоках B_3, B_4, B_5, B_6, B_7 и B_9. Оно не ожидаемо на входе в блок B_2, поскольку значение c переопределяется в этом блоке, а следовательно, значение $b + c$, которое могло бы быть вычислено в начале блока B_2, не используется вдоль любого пути. Выражение $b + c$ не является ожидаемым на входе в блок B_1, поскольку оно не нужно вдоль пути от B_1 к B_2 (хотя и используется на пути $B_1 \to B_5 \to B_6$). Аналогично это выражение не является ожидаемым в начале блока B_8 из-за ветвления от B_8 к B_{11}. Ожидаемость выражения может осциллировать вдоль пути, что на рисунке иллюстрируется путем $B_7 \to B_8 \to B_9$.

Уравнения потока данных для задачи ожидаемых выражений показаны на рис. 9.34, *а*. Анализ использует проход в обратном направлении. Ожидаемое выражение на выходе из блока B является ожидаемым на входе в этот блок, только если оно не входит в множество e_kill_B. Кроме того, блок B генерирует в качестве вновь используемых множество выражений e_use_B. На выходе из программы ни одно выражение не является ожидаемым. Поскольку нас интересует поиск выражений, ожидаемых вдоль каждого последующего пути, оператор сбора представляет собой пересечение. Следовательно, внутренние точки должны быть

инициализированы универсальным множеством U, как в случае задачи о доступных выражениях из раздела 9.2.6.

Доступные выражения

В конце второго шага копии выражения будут размещены в точках программы, где они впервые становятся ожидаемыми. В данном случае выражение является *доступным* (available) в точке p программы, если оно ожидаемо вдоль всех путей, достигающих p. Задача аналогична задаче о доступных выражениях из раздела 9.2.6. Передаточная функция, используемая здесь, немного отличается от передаточной функции для доступных выражений из раздела 9.2.6. Выражение доступно на выходе из блока, если оно

1) либо

   а) доступно на входе, либо

   б) находится в множестве ожидаемых выражений на входе (т.е. *может* стать доступным, если мы будем вычислять его здесь)

   и

2) не уничтожается в блоке.

Уравнения потока данных для доступных выражений показаны на рис. 9.34, *б*. Чтобы не запутаться в смысле обозначений IN, к имени результата анализа добавлено $[B]\,.in$.

При использовании стратегии наиболее раннего возможного размещения множество выражений, размещаемых в блоке B, т.е. $earliest\,[B]$, определяется как множество ожидаемых, но еще не доступных выражений:

$$earliest\,[B] = anticipated\,[B]\,.in - available\,[B]\,.in$$

Пример 9.27. Выражение $b+c$ в графе потока на рис. 9.35 не является ожидаемым на входе в блок B_3, но ожидаемо на входе в блок B_4. Однако нет необходимости вычислять выражение $b+c$ в блоке B_4, так как оно уже доступно благодаря блоку B_2. □

Пример 9.28. На рис. 9.33, *а* затененными показаны блоки, для которых выражение $b+c$ не является доступным; это блоки B_1, B_2, B_3 и B_5. Позиции раннего размещения представлены серыми прямоугольниками с черными тенями — это блоки B_3 и B_5. Заметим, например, что выражение $b+c$ рассматривается как доступное на входе в блок B_4, поскольку существует путь $B_1 \to B_2 \to B_3 \to B_4$, вдоль которого $b+c$ ожидаемо как минимум один раз (в данном случае — в B_3) и с начала блока B_3 ни b, ни c не переопределяются. □

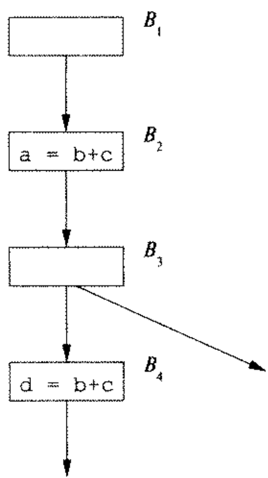


Рис. 9.35. Граф потока для примера 9.27, иллюстрирующий использование доступности

Откладываемые выражения

Третий шаг откладывает вычисления выражений на как можно более поздний срок, сохраняя при этом семантику программы и минимизируя избыточность. В примере 9.29 демонстрируется важность этого шага.

Пример 9.29. В графе потока на рис. 9.36 выражение $b + c$ на пути $B_1 \to B_5 \to \to B_6 \to B_7$ вычисляется дважды. Выражение $b + c$ является ожидаемым даже в начале блока B_1. Если мы вычислим это выражение, как только оно станет ожидаемым, то мы должны вычислять его в базовом блоке B_1. Затем этот результат должен храниться до его использования в блоке B_7, т.е. в процессе вычислений, включающих цикл из блоков B_2 и B_3. Однако можно отложить вычисления выражения $b + c$ до начала блока B_5 и до того момента, когда поток управления будет переходить от блока B_4 к блоку B_7. □

Пример 9.30. Вновь обратимся к выражению $b + c$ на рис. 9.33. Двумя наиболее ранними точками для этого выражения являются базовые блоки B_3 и B_5. Обратите внимание на то, что на рис. 9.33, *а* они оба выделены как серым, так и черным цветом, что указывает на то, что в этих блоках выражение $b + c$ является одновременно ожидаемым и недоступным. Нельзя отложить $b + c$ из блока B_5 в блок B_6, поскольку $b + c$ используется в B_5. Однако можно отложить это выражение из блока B_3 в блок B_4.

Отложить $b + c$ из блока B_4 в блок B_7 нельзя. Причина этого в том, что, хотя $b + c$ и не используется в B_4, размещение вычисления в B_7 может привести

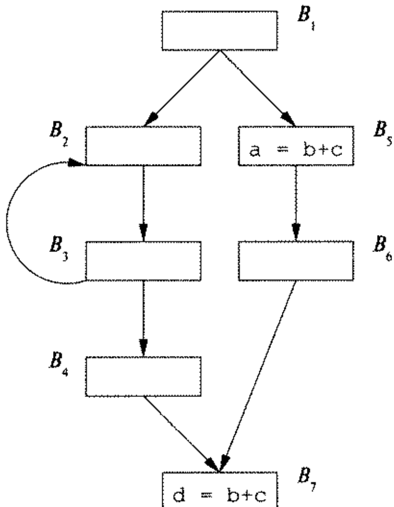


Рис. 9.36. Граф потока к примеру 9.29, иллюстрирующий необходимость откладывания выражения

к избыточному вычислению $b + c$ на пути $B_5 \to B_6 \to B_7$. Как вы увидите, B_4 — одно из наиболее поздних мест, где можно вычислять $b + c$. □

Уравнения потока данных для задачи откладываемых выражений показаны на рис. 9.34, *в*. Анализ проводится в прямом направлении. "Отложить" выражение до входа в программу невозможно, так что OUT [ВХОД] $= \varnothing$. Выражение откладываемо до выхода из блока B, если оно не используется в блоке, и либо оно откладываемо до входа в B, либо принадлежит множеству *earliest* $[B]$. Выражение не откладываемо до входа в блок, если только все его предшественники не включают данное выражение в свои множества *postponable* на выходе из этих блоков. Таким образом, оператор сбора — пересечение множеств, а внутренние точки должны быть инициализированы верхним элементом полурешетки — универсальным множеством.

Грубо говоря, выражение помещается на *границе*, где оно переходит из откладываемого в не откладываемое. Говоря более конкретно, выражение e может быть помещено в начале блока B, только если выражение на входе в B принадлежит его множеству *earliest* или *postponable*. Кроме того, B находится на откладываемой границе e, если выполняется одно из следующих условий.

1. $e \notin postponable\,[B]\,.out$. Другими словами, $e \in e_use_B$.

2. e не может быть отложено до одного из преемников B. Другими словами, существует преемник B, такой, что e не входит в множество *earliest* или *postponable* на входе в этот преемник.

Выражение e может быть помещено в начале блока B в любом из приведенных выше случаев благодаря введению новых блоков на этапе предварительных шагов алгоритма.

Пример 9.31. На рис. 9.33, *б* показан результат анализа. Серые прямоугольники представляют блоки, множества *earliest* которых включают выражение $b + c$. Черные тени указывают блоки, которые включают $b + c$ в множествах *postponable*. Таким образом, последние размещения выражений — на входах в блоки B_4 и B_5, поскольку

1. $b + c$ входит в множество *postponable* блока B_4, но не блока B_7 и
2. множество *earliest* блока B_5 включает $b + c$, а сам блок его использует.

Выражение сохраняется во временной переменной t в блоках B_4 и B_5 и во всех остальных местах, как показано на рисунке, вместо выражения $b + c$ используется значение, сохраненное в t. □

Используемые выражения

Наконец, обратный проход используется для выяснения, используются ли за пределами блока созданные в нем временные переменные. Мы говорим, что выражение *используется* (used) в точке p, если существует путь, ведущий из p, который использует выражение до того, как его значение будет вычислено заново. Это, по сути, анализ живых переменных, но выполненный не для переменных, а для выражений.

Уравнения потока данных для используемых выражений представлены на рис. 9.34, *г*. Анализ выполняется в обратном направлении. Используемое выражение на выходе из блока B является используемым на входе в этот блок, только если оно не входит в множество *latest*. В качестве новых использований блок генерирует множество e_use_B. На выходе из программы никакие выражения не используются. Поскольку нас интересует поиск выражений, которые могут быть использованы любым из путей, оператором сбора является объединение. Соответственно, внутренние точки должны быть инициализированы верхним элементом полурешетки — пустым множеством.

Алгоритм отложенного перемещения кода

Все описанные шаги алгоритма подытожены в формальном описании алгоритма 9.32.

Алгоритм 9.32. Отложенное перемещение кода

ВХОД: граф потока, для каждого базового блока B которого вычислены множества e_use_B и e_kill_B.

ВЫХОД: модифицированный граф потока, удовлетворяющий четырем условиям отложенного перемещения кода из раздела 9.5.3.

МЕТОД: выполняем следующие действия.

1. Вставляем пустые блоки во все ребра, входящие в блоки с более чем одним предшественником.

2. Для всех блоков B находим $anticipated\,[B]\,.in$, как определено на рис. 9.34, *а*.

3. Для всех блоков B находим $available\,[B]\,.in$, как определено на рис. 9.34, *б*.

4. Вычисляем наиболее раннее размещение для всех блоков B:

$$earliest\,[B] = anticipated\,[B]\,.in - available\,[B]\,.in$$

5. Для всех блоков B находим $postponable[B].in$, как определено на рис. 9.34, *в*.

6. Для всех блоков B вычисляем самые поздние размещения:

$$latest\,[B] = (earliest\,[B] \cup postponable\,[B]\,.in) \cap$$
$$\cap \left(e_use_B \cup \neg \left(\bigcap_{S \in succ(B)} (earliest\,[S] \cup postponable\,[S]\,.in)\right)\right)$$

   Здесь $\neg$ означает дополнение по отношению к множеству всех выражений, вычисляемых программой.

7. Для всех блоков B находим $used\,[B]\,.out$, как определено на рис. 9.34, *г*.

8. Для каждого выражения, скажем, $x + y$, вычисляемого программой, выполняем следующее.

   а) Создаем для $x + y$ новую переменную, скажем, t.

   б) Для всех блоков B, таких, что $x + y \in latest\,[B] \cap used\,[B]\,.out$, в начало блока B добавляем `t = x+y`.

   в) Для всех блоков B, таких, что $x+y \in e_use_B \cap (\neg latest[B] \cup used.out[B])$, заменяем каждый оригинал $x + y$ на t. □

Резюме

При устранении частичной избыточности при помощи единого алгоритма находится множество типов избыточных операций. Этот алгоритм иллюстрирует, как много задач потоков данных могут использоваться при поиске оптимального размещения выражений.

1. Ограничения на размещения накладываются анализом ожидаемости выражений, который представляет собой *обратный* анализ потока данных с пересечением в качестве оператора сбора и определяет, используется ли выражение *впоследствии* каждой точкой программы по *всем* путям.

2. Наиболее ранее размещение выражения определяется точкой программы, в которой выражение ожидаемо, но не доступно. Поиск доступных выражений выполняется при помощи *прямого* анализа потока данных с оператором сбора, представляющим собой пересечение множеств. Этот анализ выясняет, является ли выражение ожидаемым *перед* каждой точкой программы вдоль *всех* путей.

3. Наиболее позднее размещение выражения определяется точкой программы, в которой выражение более не может быть откладываемым. Выражения в некоторой точке программы являются откладываемыми, если вдоль *всех* путей, *достигающих* данной точки программы, не встречается использование этого выражения. Поиск откладываемых выражений выполняется при помощи *прямого* анализа потока данных с оператором сбора, представляющим собой пересечение множеств.

4. Присваивание временным переменным устраняется, если только они не используются *впоследствии* вдоль *некоторого* пути. Поиск используемых выражений выполняется при помощи *обратного* анализа потока данных с оператором сбора, представляющим собой объединение множеств.

9.5.6 Упражнения к разделу 9.5

Упражнение 9.5.1. Для графа потока на рис. 9.37 выполните следующее.

а) Вычислите множество *anticipated* для начала и конца каждого блока.

б) Вычислите множество *available* для начала и конца каждого блока.

в) Вычислите множество *earliest* для каждого блока.

г) Вычислите множество *postponable* для начала и конца каждого блока.

д) Вычислите множество *used* для начала и конца каждого блока.

е) Вычислите множество *latest* для каждого блока.

ж) Введите временную переменную t; покажите, где она вычисляется и где используется.

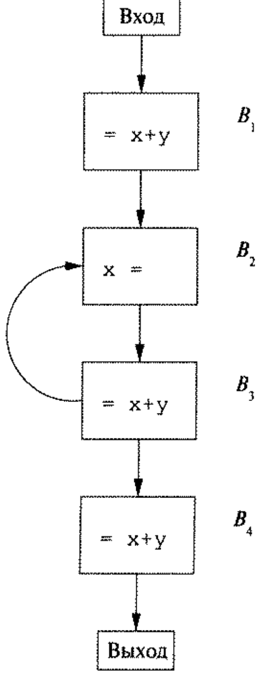


Рис. 9.37. Граф потока к упражнению 9.5.1

Упражнение 9.5.2. Повторите упражнение 9.5.1 для графа потока на рис. 9.10 (см. упражнения к разделу 9.1). В своем анализе вы можете ограничиться выражениями $a + b$, $c - a$ и $b * d$.

!! **Упражнение 9.5.3.** Рассмотренные в данном разделе концепции применимы также для устранения частично бесполезного кода. Переменная является *частично бесполезной* (partially dead), если она активна только на некоторых путях, но не на всех. Можно оптимизировать выполнение программы путем вычисления определения только на тех путях, на которых переменная является активной. В отличие от устранения частичной избыточности, когда выражения размещаются до исходного их положения, новые определения переменных размещаются после исходных. Разработайте алгоритм для перемещения частично бесполезного кода, чтобы выражения вычислялись только тогда, когда они в конце концов будут использованы.

9.6 Циклы в графах потоков

Выше мы никак не выделяли циклы — они рассматривались так же, как и любые другие виды потоков управления. Однако важность циклов не подлежит сомнению, по крайней мере потому, что программы затрачивают основное время выполнения именно на работу циклов, так что оптимизация, повышающая производительность циклов, может оказать существенное влияние на производительность программы в целом. Таким образом, желательно четко идентифицировать циклы и рассматривать их отдельно от других видов потоков управления.

Циклы также влияют на время работы анализа программы. Если в программе нет ни одного цикла, то ответ на задачи потоков данных можно получить путем одного прохода по программе. Например, задача потока данных в прямом направлении может быть решена путем однократного посещения всех узлов в топологическом порядке.

В этом разделе мы рассмотрим новые концепции — доминаторы, упорядочение вглубь, обратные ребра, глубина графа, приводимость. Все они понадобятся для последующего материала, посвященного поиску циклов и скорости сходимости итеративных анализов потоков данных.

9.6.1 Доминаторы

Мы говорим, что узел d графа потока *доминирует* (dominates) над узлом n (записывается как d *dom* n), если любой путь от входного узла графа потока к n проходит через d. Заметим, что при таком определении каждый узел доминирует над самим собой.

Пример 9.33. Рассмотрим граф потока (рис. 9.38) со входным узлом 1. Входной узел доминирует над всеми узлами (это утверждение справедливо для любого графа потока). Узел 2 доминирует только над самим собой, поскольку управление может достичь любого другого узла вдоль пути, начинающегося с $1 \to 3$. Узел 3 доминирует над всеми узлами, кроме 1 и 2. Узел 4 доминирует над всеми узлами, кроме 1, 2 и 3, так как все пути из 1 должны начинаться с $1 \to 2 \to 3 \to 4$ или $1 \to 3 \to 4$. Узлы 5 и 6 доминируют только над самими собой, поскольку управление может пропустить один узел, пройдя через другой. Наконец, узел 7 доминирует над 7, 8, 9 и 10; 8 — над 8, 9 и 10, а 9 и 10 — только над самими собой. □

Полезным способом представления информации о доминаторах является дерево, именуемое *деревом доминаторов*, в котором входной узел является корнем, а каждый узел d доминирует только над своими потомками в дереве. Например, на рис. 9.39 показано дерево доминаторов для графа потока на рис. 9.38.

Существование деревьев доминаторов следует из свойства доминаторов: каждый узел n имеет единственный *непосредственный доминатор* m, который явля-

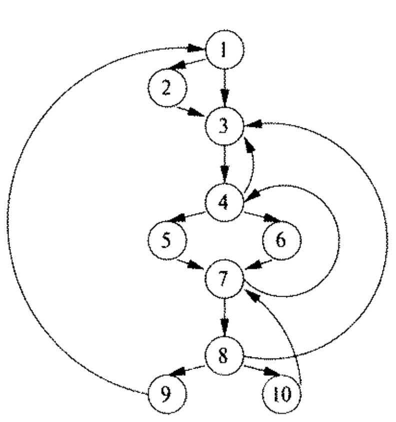


Рис. 9.38. Граф потока

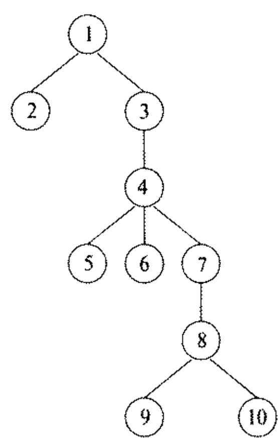


Рис. 9.39. Дерево доминаторов для графа потока на рис. 9.38

Свойства отношения *dom*

Ключевое наблюдение, касающееся доминаторов, заключается в том, что если мы возьмем любой ациклический путь от входного узла к узлу n, то все доминаторы n будут располагаться на этом пути, более того — они будут располагаться в *одном и том же порядке* вдоль любого пути. Чтобы понять, почему это так, предположим, что имеется один ациклический путь P_1 к n, на котором доминаторы a и b встречаются в указанном порядке, и другой путь P_2 к n, на котором b предшествует a. Тогда мы можем следовать по пути P_1 до a, а затем по пути P_2 до n, избежав тем самым прохождения через доминатор b. Таким образом, b не доминирует над n.

Эта аргументация позволяет нам доказать транзитивность отношения *dom*: если a *dom* b и b *dom* c, то a *dom* c. Кроме того, отношение *dom* антисимметрично: если $a \neq b$, невозможно одновременное выполнение a *dom* b и b *dom* a. Если и a, и b являются доминаторами n, то должно выполняться либо a *dom* b, либо b *dom* a. Наконец, отсюда следует, что каждый узел n за исключением входного должен иметь единственный непосредственный доминатор, т.е. доминатор, находящийся ближе всего к n вдоль любого ациклического пути от входного узла до n.

ется последним доминатором n на любом пути от входного узла до n. В терминах отношения *dom* непосредственный доминатор m обладает тем свойством, что если $d \neq n$ и d *dom* n, то d *dom* m.

Мы приведем простой алгоритм для вычисления доминаторов каждого узла n в графе потока, основанный на том принципе, что если $p_1, p_2, \ldots, p_k$ являются предшественниками n и $d \neq n$, то d *dom* n тогда и только тогда, когда d *dom* p_i для всех i. Данная задача может быть сформулирована как задача анализа потока данных в прямом направлении. Значениями потока данных являются множества базовых блоков. Множество доминаторов узла, исключая его самого, представляет собой пересечение доминаторов всех его предшественников; таким образом, оператором сбора является пересечение множеств. Передаточная функция для блока B просто добавляет B к входному множеству узлов. Граничное условие заключается в том, что доминатором входного узла является сам входной узел. Наконец, внутренние узлы инициализируются универсальным множеством, т.е. множеством всех узлов.

Алгоритм 9.34. Поиск доминаторов

ВХОД: граф потока G с множеством узлов N, множеством ребер E и входным узлом.

ВЫХОД: для всех узлов $n \in N$ значение $D(n)$, представляющее собой множество узлов, доминирующих над n.

МЕТОД: найти решение задачи потока данных, параметры которой показаны на рис. 9.40. Базовые блоки представляют собой узлы. $D(n) = \text{OUT}[n]$ для всех $n \in N$. □

	ДОМИНАТОРЫ
Область определения	Показательное множество N
Направление	Прямое
Передаточная функция	$f_B(x) = x \cup \{B\}$
Граничное условие	$\text{OUT}[\text{ВХОД}] = \{\text{ВХОД}\}$
Оператор сбора ($\wedge$)	$\cap$
Уравнения	$\text{OUT}[B] = f_B(\text{IN}[B])$ $\text{IN}[B] = \wedge_{P,pred(B)} \text{OUT}[P]$
Инициализация	$\text{OUT}[B] = N$

Рис. 9.40. Алгоритм потока данных для вычисления доминаторов

Поиск доминаторов с использованием алгоритма потока данных весьма эффективен. Узлы графа должны посещаться всего лишь несколько раз, как будет видно из раздела 9.6.7.

Пример 9.35. Вернемся к графу потока на рис. 9.38 и предположим, что цикл в строках 4–6 на рис. 9.23 посещает узлы в числовом порядке. Пусть $D(n)$ –

множество узлов в OUT $[n]$. Поскольку входной узел имеет номер 1, в строке 1 $D(1)$ присваивается значение $\{1\}$. Узел 2 в качестве предшественника имеет только узел 1, так что $D(2) = \{2\} \cup D(1)$. Таким образом, $D(2)$ устанавливается равным $\{1, 2\}$. Затем рассматриваем узел 3 с предшественниками 1, 2, 4 и 8. Поскольку все внутренние узлы инициализированы универсальным множеством N,

$$D(3) = \{3\} \cup (\{1\} \cap \{1,2\} \cap \{1,2,\ldots,10\} \cap \{1,2,\ldots,10\}) = \{1,3\}$$

Остальные вычисления приведены на рис. 9.41. Поскольку эти значения не изменяются при второй итерации внешнего цикла в строках 3–6 на рис. 9.23, *а*, они являются окончательным решением задачи о доминаторах. □

$$\begin{aligned}
D(4) &= \{4\} \cup (D(3) \cap D(7)) = \{4\} \cup (\{1,3\} \cap \{1,2,\ldots,10\}) = \{1,3,4\} \\
D(5) &= \{5\} \cup D(4) = \{5\} \cup \{1,3,4\} = \{1,3,4,5\} \\
D(6) &= \{6\} \cup D(4) = \{6\} \cup \{1,3,4\} = \{1,3,4,6\} \\
D(7) &= \{7\} \cup (D(5) \cap D(6) \cap D(10)) = \\
&= \{7\} \cup (\{1,3,4,5\} \cap \{1,3,4,6\} \cap \{1,2,\ldots,10\}) = \{1,3,4,7\} \\
D(8) &= \{8\} \cup D(7) = \{8\} \cup \{1,3,4,7\} = \{1,3,4,7,8\} \\
D(9) &= \{9\} \cup D(8) = \{9\} \cup \{1,3,4,7,8\} = \{1,3,4,7,8,9\} \\
D(10) &= \{10\} \cup D(8) = \{10\} \cup \{1,3,4,7,8\} = \{1,3,4,7,8,10\}
\end{aligned}$$

Рис. 9.41. Завершение вычисления доминаторов из упражнения 9.35

9.6.2 Упорядочение в глубину

Как говорилось в разделе 2.3.4, *поиск в графе в глубину* (depth-first search) однократно посещает все узлы графа, начиная с входного узла и посещая, в первую очередь, насколько это возможно, узлы, максимально удаленные от входного. Путь поиска в глубину образует *глубинное остовное дерево* (охватывающее вглубь дерево) (depth-first spanning tree — DFST). Вспомним из раздела 2.3.4, что обход в прямом прядке посещает узел перед посещением любого из его дочерних узлов, которые затем рекурсивно посещаются в порядке слева направо. Обход в обратном порядке сначала рекурсивно слева направо посещает узлы, дочерние по отношению к текущему, а затем посещает сам текущий узел.

Есть еще один вариант упорядочения, важный для анализа графа потока, — *упорядочение в глубину* (depth-first ordering), которое представляет собой обращение обратного порядка обхода. Иначе говоря, при упорядочении в глубину мы посещаем узел, затем обходим его крайний справа узел-преемник, после этого — узел, расположенный слева от него, и т.д. Однако перед тем как строить дерево для

графа потока, следует выбрать, какой из преемников является крайним справа, какой — его левым соседом и т.д. Перед тем как перейти к алгоритму упорядочения в глубину, рассмотрим пример.

Пример 9.36. Одно из возможных упорядочений в глубину графа на рис. 9.38 показано на рис. 9.42. Сплошные линии образуют дерево; пунктирные представляют собой прочие ребра графа потока. Обход дерева в глубину дает нам $1 \rightarrow 3 \rightarrow 4 \rightarrow$ $\rightarrow 6 \rightarrow 7 \rightarrow 8 \rightarrow 10$, после чего выполняется возврат к 8 и посещение 9. Затем мы снова возвращаемся в 8, отступаем в 7, 6 и 4 и посещаем 5. Из 5 мы возвращаемся в 4, оттуда — в 3 и 1. Из 1 посещаем 2, опять возвращаемся в 1, и на этом обход завершается.

Последовательность прямого обхода, таким образом, выглядит так:

$$1, 3, 4, 6, 7, 8, 10, 9, 5, 2$$

Последовательность обратного обхода дерева на рис. 9.42 выглядит так:

$$10, 9, 8, 7, 6, 5, 4, 3, 2, 1$$

Упорядочение в глубину, которое представляет собой обращение обратного порядка обхода, дает последовательность

$$1, 2, 3, 4, 5, 6, 7, 8, 9, 10$$

□

Рис. 9.42. Представление графа потока на рис. 9.38

Теперь приведем алгоритм, который находит глубинное остовное дерево и упорядочение графа в глубину. Этот алгоритм для графа на рис. 9.38 находит DFST на рис. 9.42.

Алгоритм 9.37. Глубинное остовное дерево и упорядочение графа в глубину

Вход: граф потока G.

Выход: DFST T графа G и порядок узлов G.

Метод: используется рекурсивная процедура *search* (n), показанная на рис. 9.43. Алгоритм инициализирует все узлы G значением "непосещен" и вызывает *search* (n_0), где n_0 — входной узел. При вызове *search* (n) узел n маркируется как "посещен", чтобы избежать повторного добавления n в дерево. Процедура использует c для счета от количества узлов в G до 1, присваивая поочередно номера в порядке в глубину *dfn* $[n]$ встречающимся узлам n. Множество ребер T образует глубинное остовное дерево G. □

```
void search (n) {
    Маркируем n как "посещен";
    for (каждый узел s, являющийся преемником n)
        if (s "непосещен") {
            Добавляем ребро n → s в T;
            search (s);
        }
    dfn [n] = c;
    c = c − 1;
}

main () {
    T = ∅; /* Множество ребер */
    for (каждый узел n из G)
        Маркируем n как "непосещен";
    c = количество узлов в G;
    search (n0);
}
```

Рис. 9.43. Алгоритм поиска в глубину

Пример 9.38. Для графа потока на рис. 9.42 алгоритм 9.37 устанавливает c равным 10 и начинает работу вызовом *search* (1). Остальная работа алгоритма приведена на рис. 9.44. □

9.6.3 Ребра в глубинном остовном дереве

При построении DFST для графа потока ребра графа потока можно разделить на три категории.

Вызов *search* (1)	Узел 1 имеет два преемника. Предположим, что сначала рассматривается $s = 3$; добавляем ребро $1 \to 3$ в T.
Вызов *search* (3)	Добавляем ребро $3 \to 4$ в T.
Вызов *search* (4)	Узел 4 имеет два преемника. Предположим, что сначала рассматривается $s = 6$; добавляем ребро $4 \to 6$ в T.
Вызов *search* (6)	Добавляем ребро $6 \to 7$ в T.
Вызов *search* (7)	Узел 7 имеет два преемника, 4 и 8. Но узел 4 уже маркирован как "посещен" вызовом *search* (4), так что для $s = 4$ никаких действий не выполняется. Для $s = 8$ добавляем ребро $7 \to 8$ в T.
Вызов *search* (8)	Узел 8 имеет два преемника, 9 и 10. Предположим, что сначала рассматривается $s = 10$; добавляем ребро $8 \to 10$ в T.
Вызов *search* (10)	10 имеет преемника — узел 7, но 7 уже помечен как "посещен". Таким образом, *search* (10) завершает работу установками *dfn* [10] = 10 и $c = 9$.
Возврат в *search* (8)	Устанавливаем $s = 9$ и добавляем ребро $8 \to 9$ в T.
Вызов *search* (9)	Единственным преемником 9 является уже посещенный узел 1, так что устанавливаем *dfn* [9] = 9 и $c = 8$.
Возврат в *search* (8)	Последним преемником 8 является посещенный узел 3, так что для $s = 3$ не выполняем никаких действий. В этот момент все преемники 8 рассмотрены, так что устанавливаем *dfn* [8] = 8 и $c = 7$.
Возврат в *search* (7)	Все преемники 7 рассмотрены, так что устанавливаем *dfn* [7] = 7 и $c = 6$.
Возврат в *search* (6)	Аналогично все преемники 6 рассмотрены, так что устанавливаем *dfn* [6] = 6 и $c = 5$.
Возврат в *search* (4)	Преемник 3 узла 4 был посещен, но 5 — еще нет, так что добавляем в дерево ребро $4 \to 5$.
Вызов *search* (5)	Преемник 7 узла 5 был посещен, так что устанавливаем *dfn* [5] = 5 и $c = 4$.
Возврат в *search* (4)	Все преемники 4 рассмотрены; устанавливаем *dfn* [4] = 4 и $c = 3$.
Возврат в *search* (3)	Устанавливаем *dfn* [3] = 3 и $c = 2$.
Возврат в *search* (1)	Узел 2 еще не был посещен, так что добавляем ребро $1 \to 2$ в T.
Вызов *search* (2)	Устанавливаем *dfn* [2] = 2 и $c = 1$.
Возврат в *search* (1)	Устанавливаем *dfn* [1] = 1 и $c = 0$.

Рис. 9.44. Выполнение алгоритма 9.37 для графа потока на рис. 9.42

1. Ребра, именуемые *наступающими* (advancing), идут от узла m к истинным преемникам m в дереве. Все ребра в DFST являются наступающими. Других наступающих ребер на рис. 9.42 нет, но, например, если бы имелось ребро $4 \to 8$, то оно также попадало бы в эту категорию.

2. Ребра, идущие от узла m к предку m в дереве (возможно, к самому m). Эти ребра называются *отступающими* (retreating). Например, на рис. 9.42 отступающими ребрами являются $4 \to 3$, $7 \to 4$, $10 \to 7$, $8 \to 3$ и $9 \to 1$.

3. Ребра $m \to n$, такие, что ни m, ни n не являются предками друг друга. На рис. 9.42 такими ребрами являются только ребра $2 \to 3$ и $5 \to 7$. Такие ребра называются *поперечными* (cross). Важным свойством поперечных ребер является то, что если изобразить DFST так, чтобы дочерние узлы некоторого узла располагались слева направо в порядке, в котором они добавлялись в дерево, то все поперечные ребра будут идти справа налево.

Следует заметить, что $m \to n$ является отступающим ребром тогда и только тогда, когда *dfn* $[m] \geqslant$ *dfn* $[n]$. Чтобы понять, почему это так, заметим, что если m является потомком n в DFST, то *search* (m) завершается раньше *search* (n), так что *dfn* $[m] \geqslant$ *dfn* $[n]$. И наоборот, если *dfn* $[m] \geqslant$ *dfn* $[n]$, то *search* (m) завершается до *search* (n), или $m = n$. Но вызов *search* (n) должен начинаться до *search* (m), если существует ребро $m \to n$, иначе тот факт, что n является преемником m, должен сделать n потомком m в DFST. Таким образом, время активности *search* (m) представляет собой подынтервал времени активности *search* (n), откуда следует, что n является предком m в DFST.

9.6.4 Обратные ребра и приводимость

Обратным ребром (back edge) называется ребро $a \to b$, у которого b доминирует над a. Для любого графа потока каждое обратное ребро является отступающим, но не всякое отступающее ребро является обратным. Граф потока называется *приводимым* (reducible), если все его отступающие ребра в любом DFST являются обратными. Другими словами, если граф приводим, то все его DFST имеют одно и то же множество отступающих ребер, в точности совпадающее с множеством обратных ребер. Если граф неприводим (*nonreducible*), то все обратные ребра в любом DFST являются отступающими, но каждое DFST имеет дополнительные отступающие ребра, не являющиеся обратными. Эти отступающие ребра могут различаться у разных DFST. Таким образом, если удалить все обратные ребра графа потока и оставшийся граф при этом будет цикличен, то такой граф неприводим, и наоборот.

Графы потоков, встречающиеся на практике, почти всегда приводимы. При использовании только структурированных инструкций потока управления, таких

Почему обратные ребра являются отступающими

Предположим, что $a \to b$ — обратное ребро, т.е. его заголовок доминирует над хвостом. Последовательность вызовов функции *search* на рис. 9.43, приводящая к узлу a, должна быть путем в графе потока. Этот путь, конечно, должен включать все доминаторы a. Отсюда следует, что вызов *search* (b) должен быть открыт, когда вызывается *search* (a). Следовательно, b уже находится в дереве, когда в него добавляется a, и a добавляется как потомок b. Следовательно, $a \to b$ должно быть отступающим ребром.

как if-then-else, while-do, continue и break, получаются программы, графы потоков которых всегда приводимы. Даже программы, написанные с использованием инструкции goto, зачастую можно сделать приводимыми, если программист логически мыслит в терминах циклов и ветвлений.

Пример 9.39. Граф потока на рис. 9.38 приводим. Все отступающие ребра в графе являются обратными, т.е. их заголовки доминируют над их хвостами. □

Пример 9.40. Рассмотрим граф потока на рис. 9.45, начальный узел которого — 1. Узел 1 доминирует над узлами 2 и 3, но ни 2 не доминирует над 3, ни 3 над 2. Таким образом, этот граф потока не имеет обратных ребер, поскольку ни у одного ребра заголовок не доминирует над хвостом. Для данного графа потока существует два возможных глубинных остовных дерева, в зависимости от того, будет первой из *search* (1) вызвана процедура *search* (2) или *search* (3). В первом случае ребро $3 \to 2$ является отступающим, но не обратным; во втором таковым является ребро $2 \to 3$. Интуитивно причина неприводимости данного графа потока в том, что в цикл 2–3 можно войти в двух разных местах — через узлы 2 и 3. □

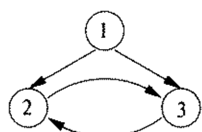


Рис. 9.45. Канонический неприводимый граф потока

9.6.5 Глубина графа потока

Для данного глубинного остовного дерева вглубь дерева графа *глубина* (depth) представляет собой наибольшее количество отступающих ребер на любом ациклическом пути. Можно доказать, что глубина никогда не превышает того, что можно

интуитивно назвать глубиной вложенности циклов графа потока. Если граф потока приводим, мы можем заменить в определении глубины "отступающие" на "обратные", поскольку отступающие ребра любого DFST являются обратными. Понятие глубины при этом становится независимым от реально выбранного DFST, и можно говорить о "глубине графа потока", а не о глубине графа потока в связи с одним из его глубинных остовных деревьев.

Пример 9.41. На рис. 9.42 глубина равна 3, поскольку существует путь

$$10 \to 7 \to 4 \to 3$$

с тремя отступающими ребрами, но нет ацикличного пути с четырьмя или более отступающими ребрами. То, что "глубочайший" путь содержит только отступающие ребра, — не более чем случайное совпадение; в общем случае на наиболее глубоком пути у нас может быть смесь отступающих, наступающих и поперечных ребер. □

9.6.6 Естественные циклы

Циклы в исходной программе могут определяться различными способами: они могут быть созданы как циклы for, while или repeat; они могут даже быть определены с использованием меток и инструкций goto. С точки зрения анализа программ не имеет значения, как именно выглядят циклы в исходном тексте программы. Имеет значение только то, обладают ли они свойствами, допускающими простую их оптимизацию. В частности, нас интересует, имеется ли у цикла одна точка входа. Если это так, то компилятор в ходе анализа может предполагать выполнение некоторых начальных условий в начале каждой итерации цикла. Эта возможность служит причиной определения "естественного цикла".

Такие циклы обладают двумя важными свойствами.

1. Цикл должен иметь единственный входной узел, называемый *заголовком* (header). Этот входной узел доминирует над всеми узлами цикла, иначе он не является единственной точкой входа в цикл.

2. Должно существовать обратное ребро, ведущее в заголовок цикла. В противном случае поток управления не сможет вернуться в заголовок непосредственно из "цикла", т.е. данная структура циклом в таком случае не является.

Для данного обратного ребра $n \to d$ определим естественный цикл ребра (natural loop of the edge) как d плюс множество узлов, которые могут достичь n, не проходя через d. Узел d является заголовком цикла.

Алгоритм 9.42. Построение естественного цикла обратной дуги

ВХОД: граф потока G и обратная дуга $n \to d$.

ВЫХОД: множество *loop*, состоящее из всех узлов естественного цикла $n \to d$.

МЕТОД: пусть *loop* представляет собой $\{n, d\}$. Пометим d как "посещенный", чтобы поиск не проходил дальше d. Выполним поиск в глубину на обратном графе потока, начиная с n. Внесем все посещенные при этом поиске узлы в *loop*. Такая процедура позволяет найти все узлы, достигающие n, минуя d. □

Пример 9.43. На рис. 9.38 имеется пять обратных ребер, заголовки которых доминируют над хвостами: $10 \to 7, 7 \to 4, 4 \to 3, 8 \to 3$ и $9 \to 1$. Это ребра, которые могут формировать циклы в графе потока.

Обратное ребро $10 \to 7$ имеет естественный цикл $\{7, 8, 10\}$, поскольку 8 и 10 — единственные узлы, которые достигают 10, не проходя через 7. Обратное ребро $4 \to 7$ имеет естественный цикл, состоящий из $\{4, 5, 6, 7, 8, 10\}$, а следовательно, включающий цикл для ребра $10 \to 7$. Таким образом, последний рассматривается как внутренний цикл, содержащийся в первом.

Естественные циклы для ребер $4 \to 3$ и $8 \to 3$ имеют один и тот же заголовок — узел 3, а также одно и то же множество узлов $\{3, 4, 5, 6, 7, 8, 10\}$. Следовательно, эти два цикла надо объединить в один. Этот цикл содержит два меньших цикла, рассмотренных ранее.

Наконец, ребро $9 \to 1$ в качестве естественного цикла имеет весь граф потока, а потому является внешним циклом. В нашем примере четыре цикла вложены один в другой. Однако типичной является ситуация, когда имеются два цикла, ни один из которых не является подмножеством другого. □

В приводимых графах потоков, поскольку все отступающие ребра являются обратными, с каждым отступающим ребром можно связать естественный цикл. Это утверждение для неприводимых графов несправедливо. Например, неприводимый граф потока на рис. 9.45 имеет цикл, состоящий из узлов 2 и 3. Ни одно из ребер в цикле не является обратным, так что этот цикл не подпадает под определение естественного. Таким образом, мы не идентифицируем данный цикл как естественный, так что он не оптимизируется как таковой. Такое решение приемлемо, поскольку наш анализ циклов может оказаться существенно проще в предположении, что все циклы имеют по одному входному узлу (тем более что на практике неприводимые программы встречаются очень редко).

Рассматривая в качестве "циклов" только естественные циклы, мы получаем полезное свойство, что если только два узла не имеют общего заголовка, то они либо непересекающиеся, либо один из них вложен в другой. Так мы получаем естественное понятие *наиболее глубоко вложенного внутреннего цикла* (innermost loop) — это цикл, в который не вложен никакой другой цикл.

Когда два естественных цикла имеют один и тот же заголовок, как на рис. 9.46, сложно сказать, какой из циклов является вложенным. Таким образом, мы будем считать, что когда два естественных цикла имеют один и тот же заголовок, то ни один из них не содержится в другом, — они объединяются и рассматриваются как единый цикл.

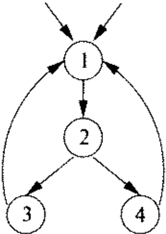


Рис. 9.46. Два цикла с одним и тем же заголовком

Пример 9.44. На рис. 9.46 естественными циклами для обратных ребер $3 \to 1$ и $4 \to 1$ являются соответственно $\{1, 2, 3\}$ и $\{1, 2, 4\}$. Мы объединим их в единый цикл $\{1, 2, 3, 4\}$.

Однако если бы на рис. 9.46 имелось обратное ребро $2 \to 1$, то его естественным циклом был бы цикл $\{1, 2\}$ — третий цикл с заголовком 1. Это множество узлов является истинным подмножеством $\{1, 2, 3, 4\}$, так что его не надо объединять с другими естественными циклами; оно рассматривается как внутренний цикл, вложенный в другой. □

9.6.7 Скорость сходимости итеративных алгоритмов потоков данных

Теперь мы готовы рассмотреть скорость сходимости итеративных алгоритмов. Как говорилось в разделе 9.3.3, максимальное количество итераций, которые может выполнить алгоритм, равно произведению высоты решетки на количество узлов в графе потока. Во многих анализах потоков данных можно так упорядочить вычисления, что алгоритм будет сходиться за существенно меньшее количество итераций. Интересующее нас свойство заключается в возможности распространения всех важных событий к узлу по некоторому ациклическому пути. Среди рассматривавшихся анализов потоков данных таким свойством обладают достигающие определения, доступные выражения и активные переменные, но не распространение констант. Сформулируем сказанное более конкретно.

- Если определение d находится в IN$[B]$, то существует некоторый ациклический путь из блока, содержащего d, в B, такой, что d находится во всех IN и OUT вдоль этого пути.

- Если выражение $x + y$ *не* доступно на входе в блок B, то существует некоторый ациклический путь, демонстрирующий, что либо путь идет от входного узла и не включает инструкции, уничтожающие или генерирующие $x + y$, либо путь идет от блока, который уничтожает $x + y$, и вдоль этого пути нет последующей генерации $x + y$.

- Если переменная x активна на выходе из блока B, то существует ациклический путь от B к использованию x, вдоль которого нет определений x.

Мы должны проверить, что в каждом из этих случаев пути с циклами ничего не добавляют. Например, если использование x достигается от конца блока B вдоль пути с циклом, мы можем убрать этот цикл, чтобы найти более короткий путь, вдоль которого x все равно достигается из B.

Распространение констант, напротив, этим свойством не обладает. Рассмотрим простую программу, в которой имеется один цикл, содержащий базовый блок с инструкциями

```
L: a = b
   b = c
   c = 1
   goto L
```

При первом посещении базового блока c принимает константное значение 1, но и a, и b не определены. При втором проходе по циклу мы находим, что и b, и c принимают константное значение 1. И только на третьем проходе константное значение c, равное 1, достигает переменной a.

Если вся полезная информация распространяется вдоль ациклических путей, мы имеем возможность подобрать порядок посещения узлов в итеративных алгоритмах потоков данных так, чтобы после относительно небольшого количества проходов по узлам мы могли быть уверены, что информация прошла по всем ациклическим путям.

Вспомним из раздела 9.6.3, что если $a \to b$ — ребро, то номер b при упорядочении в глубину меньше номера a, только если это ребро — отступающее. В случае задач потоков данных в прямом направлении желательно посещать узлы в соответствии с их упорядочением в глубину. В частности, модифицируем алгоритм на рис. 9.23, *а*, заменяя строку 4, в которой выполняется обход базовых блоков графа потока, строкой

for(каждый базовый блок B, отличный от входного, в порядке в глубину) {

Пример 9.45. Предположим, что у нас имеется путь, вдоль которого распространяется определение d

$$3 \to 5 \to 19 \to 35 \to 16 \to 23 \to 45 \to 4 \to 10 \to 17$$

Причина неприводимости графов потоков

Имеется одно место, где в общем случае нельзя ожидать приводимости графов потоков. Если обратить ребра графа потока программы, как это делалось в алгоритме 9.42 при поиске естественных циклов, то можно не получить приводимый граф. Интуитивно причина этого в том, что в то время как в типичной программе циклы имеют один вход, зачастую у них может оказаться несколько выходов, которые становятся входами при обращении ребер.

Целые числа в нем указывают номера блоков пути при упорядочении в глубину. Тогда при первом проходе цикла в строках 4–6 на рис. 9.23, *а* определение d будет распространяться от OUT [3] в IN [5], оттуда — в OUT [5] и так до OUT [35]. Оно не сможет достичь IN [16] в этом проходе, поскольку 16 предшествует 35, и IN [16] должно быть вычислено к тому времени, когда d помещается в OUT [35]. Однако на следующем проходе при вычислении IN [16] d будет включено в это множество, поскольку оно находится в OUT [35]. Определение d в этом проходе распространится в OUT [16], IN [23] и далее до OUT [45], где оно должно будет подождать, поскольку IN [4] уже вычислено в данном проходе. При третьем проходе d распространяется далее в IN [4],OUT [4], IN [10], OUT [10] и IN [17], так что после трех проходов выясняется, что d достигает блока 17. □

Из этого примера несложно вывести общий принцип. Если на рис. 9.23, *а* мы используем порядок "вглубь", то количество проходов, необходимое для распространения любого достигающего определения вдоль любого ацикличного пути, не более чем на единицу превышает число ребер пути, идущих от блока с бóльшим номером к блоку с меньшим. Эти ребра являются отступающими, так что необходимое количество проходов на единицу больше глубины. Конечно, алгоритму 9.11, для того чтобы определить достижение всех определений, требуется один дополнительный проход, так что верхняя граница количества проходов, необходимых алгоритму с упорядочением блоков вглубь, в действительности на 2 превышает глубину. Изучение[11] показало, что средняя глубина типичного графа потока равна 2.75. Таким образом, алгоритм сходится очень быстро.

В случае задач в направлении, обратном к направлению потока, таких как анализ активных переменных, мы посещаем узлы в порядке, обратном порядку "вглубь". Таким образом, мы можем распространить использование переменной в блоке 17 в обратном направлении по пути

$$3 \to 5 \to 19 \to 35 \to 16 \to 23 \to 45 \to 4 \to 10 \to 17$$

[11] D.E. Knuth, "An empirical study of FORTRAN programs", *Software – Practice and Experience* 1:2 (1971), pp. 105–133.

за один проход до IN [4], где мы должны дождаться следующего прохода, чтобы достичь OUT [45]. При втором проходе мы достигаем IN [16], а на третьем проходим от OUT [35] до OUT [3].

В общем случае, если мы выберем порядок посещения узлов при проходе, обратный нумерации вглубь, то в связи с тем, что распространение в данной ситуации по уменьшающейся последовательности номеров выполняется за один проход, нам будет достаточно количества проходов, на единицу превышающего глубину.

Описанная граница представляет собой верхнюю границу для всех задач, в которых циклические пути не добавляют информацию к анализу. В частных случаях, таких как задача о доминаторах, алгоритм сходится даже еще быстрее. Если входной граф потока приводим, корректное множество доминаторов для каждого узла получается при первой итерации алгоритма потока данных, который посещает узлы в порядке в глубину. Если заранее не известно, приводим ли входной граф, потребуется дополнительная итерация для того, чтобы убедиться, что решение сошлось.

9.6.8 Упражнения к разделу 9.6

Упражнение 9.6.1. Для графа потока на рис. 9.10 (см. упражнения к разделу 9.1) выполните следующее.

а) Вычислите отношение доминирования.

б) Найдите для каждого узла его непосредственный доминатор.

в) Постройте дерево доминаторов.

г) Найдите упорядочение вглубь графа потока.

д) Укажите в вашем ответе к п. *г* наступающие, отступающие, поперечные ребра и ребра дерева.

е) Является ли данный граф потока приводимым?

ж) Вычислите глубину графа потока.

з) Найдите естественные циклы в графе потока.

Упражнение 9.6.2. Повторите упражнение 9.6.1 для следующих графов потоков.

а) Граф потока на рис. 9.3.

б) Граф потока на рис. 8.9.

в) Граф потока, разработанного вами при выполнении упражнения 8.4.1.

г) Граф потока, разработанного вами при выполнении упражнения 8.4.2.

! **Упражнение 9.6.3.** Докажите справедливость следующих утверждений об отношении *dom*.

а) Если a *dom* b и b *dom* c, то a *dom* c (*транзитивность*).

б) Невозможно, чтобы одновременно выполнялось a *dom* b и b *dom* a, если только a не совпадает с b (*антисимметрия*).

в) Если a и b являются двумя доминаторами n, то должно выполняться либо a *dom* b, либо b *dom* a.

г) Каждый узел n, за исключением входного, имеет единственный *непосредственный доминатор* — доминатор, который ближе других к n вдоль любого ацикличного пути от входного узла к n.

! **Упражнение 9.6.4.** На рис. 9.42 приведено одно из представлений вглубь графа потока на рис. 9.38. Сколько имеется других представлений данного графа потока? Напомним, что представления различаются порядком дочерних узлов.

!! **Упражнение 9.6.5.** Докажите, что граф потока приводим тогда и только тогда, когда граф потока ацикличен после удаления из него всех обратных ребер (у которых заголовок доминирует над хвостом).

! **Упражнение 9.6.6.** *Полный граф потока* (complete flow graph) с n узлами содержит дуги $i \to j$ между любыми двумя узлами i и j (в обоих направлениях). Для каких значений n такой граф является приводимым?

! **Упражнение 9.6.7.** *Полный ациклический граф потока* (complete acyclic flow graph) с n узлами $1, 2, \ldots, n$ содержит дуги $i \to j$ для всех узлов i и j, таких, что $i < j$. Узел 1 является входным.

а) Для каких значений n такой граф является приводимым?

б) Изменится ли ваш ответ на вопрос *а*, если к каждому узлу i добавить петлю $i \to i$?

! **Упражнение 9.6.8.** Естественный цикл для обратного ребра $n \to h$ определен как h плюс множество узлов, которые могут достичь n, не проходя через h. Покажите, что h доминирует над всеми узлами естественного цикла для $n \to h$.

!! **Упражнение 9.6.9.** Мы утверждаем, что граф потока на рис. 9.45 неприводим. Если дуги заменить путями непересекающихся множеств узлов (за исключением конечных точек), то граф останется неприводимым. В действительности узел 1 не обязан быть входным; он может быть любым узлом, достижимым из входного узла

по пути, промежуточные узлы которого не являются частью никакого из четырех явно показанных на рисунке путей. Докажите обратное — что любой неприводимый граф потока содержит подграф наподобие показанного на рис. 9.45, но с дугами, которые могут быть заменены путями по непересекающимся множествам узлов, а узел 1 быть любым узлом, достижимым из входной точки по пути, который не включает узлы из четырех других путей.

!! **Упражнение 9.6.10.** Покажите, что любое представление вглубь любого неприводимого графа содержит отступающее ребро, не являющееся обратным.

!! **Упражнение 9.6.11.** Покажите, что если для любых функций f и g и значения a выполняется условие

$$f(a) \wedge g(a) \wedge a \leqslant f(g(a)),$$

то обобщенный итеративный алгоритм 9.25 с итерациями в порядке "вглубь" сходится за количество проходов, на 2 превышающее глубину.

! **Упражнение 9.6.12.** Найдите неприводимый граф потока с двумя различными DFST, которые имеют разную глубину.

! **Упражнение 9.6.13.** Докажите следующие утверждения.

а) Если определение d находится в $\text{IN}[B]$, то существует некоторый ациклический путь от блока, содержащего d, к блоку B, такой, что d находится во всех множествах IN и OUT вдоль этого пути.

б) Если выражение $x + y$ *не*доступно на входе в блок B, то существует некоторый ациклический путь, демонстрирующий этот факт. Либо этот путь представляет собой путь из входного узла и не включает инструкции, которые уничтожают или генерируют $x + y$, или путь ведет из блока, который уничтожает $x + y$ и вдоль которого нет последующей генерации $x + y$.

в) Если переменная x активна на выходе из блока B, то существует ациклический путь из B к использованию x, вдоль которого нет определений x.

9.7 Анализ на основе областей

Итеративный алгоритм анализа потока данных, рассматривавшийся нами, — всего лишь один из подходов к решению задач потоков данных. Сейчас мы рассмотрим еще один подход — анализ на основе областей (region-based analysis). Вспомним, что в случае итеративного подхода мы создавали передаточную функцию для базовых блоков, затем находили решение путем многократного прохода по блокам. Анализ на основе областей вместо передаточной функции для отдельных блоков находит передаточные функции, которые подытоживают выполнение

все больших и больших областей программы. В конечном счете строится и применяется передаточная функция для всей процедуры целиком, применяя которую, мы непосредственно получаем требуемые нам значения потока данных.

Структура потока данных, использующая итеративный алгоритм, определяется полурешеткой значений потока данных и семейством передаточных функций, замкнутых относительно композиции. Анализ же на основе областей требует большего количества элементов. Соответствующая структура включает как полурешетку значений потока данных, так и полурешетку передаточных функций, которая должна располагать оператором сбора, оператором композиции и оператором замыкания. Все эти элементы будут рассмотрены нами в разделе 9.7.4.

Анализ на основе областей в особенности применим к задачам потоков данных, в которых пути содержат циклы, которые могут изменять значения потоков данных. Оператор замыкания позволяет более эффективно по сравнению с итеративным анализом подытожить влияние цикла. Этот метод оказывается эффективным и при межпроцедурном анализе, когда передаточные функции, связанные с вызовом процедуры, могут рассматриваться так же, как и передаточные функции, связанные с базовыми блоками.

9.7.1 Области

В процессе анализа на основе областей программа рассматривается как иерархия *областей* (region), которые грубо можно считать частями графа потока, имеющими единственную точку входа. Такая концепция рассмотрения кода как иерархии областей должна быть интуитивно понятна, поскольку процедура, составленная из блоков, естественным образом организуется в виде иерархии областей. Каждая инструкция в блочно-структурированной программе является областью, поскольку поток управления может достичь инструкции только через ее начало. Каждый уровень вложенности инструкций соответствует уровню в иерархии областей.

Формально *область* (region) графа потока представляет собой набор узлов N и ребер E, таких, что

1. существует заголовок $h \in N$, доминирующий над всеми узлами в N;
2. если некоторый узел m может достичь узла $n \in N$, минуя h, то m также входит в N;
3. E — множество всех ребер потока управления между узлами n_1 и n_2 из N, за исключением, возможно, некоторых ребер, входящих в h.

Пример 9.46. Ясно, что естественный цикл представляет собой область, но область не обязательно содержит обратное ребро и не обязательно включает цикл.

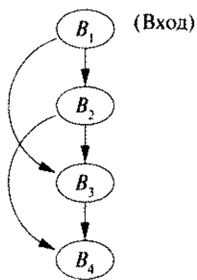


Рис. 9.47. Примеры областей

Например, на рис. 9.47 узлы B_1 и B_2 вместе с ребром $B_1 \to B_2$ образуют область; то же самое можно сказать и об узлах B_1, B_2 и B_3 и ребрах $B_1 \to B_2$, $B_2 \to B_3$ и $B_1 \to B_3$.

Однако подграф с узлами B_2 и B_3 и ребром $B_2 \to B_3$ область не образует, поскольку управление может попасть в подграф как через узел B_2, так и через узел B_3. Говоря более точно, ни B_2, ни B_3 не доминируют друг над другом, так что условие 1 для области оказывается не выполненным. Даже если мы выберем, скажем, B_2 в качестве "заголовка", то будет нарушено условие 2, поскольку можно достичь B_3 из B_1, не проходя через B_2, а B_1 не входит в "область". □

9.7.2 Иерархии областей для приводимых графов потоков

В приведенном далее материале предполагается, что граф потока приводим. Если вдруг нам придется иметь дело с неприводимыми графами, мы можем воспользоваться методом "расщепления узлов", рассматривающимся в разделе 9.7.6.

Для построения иерархии областей мы идентифицируем естественные циклы. Вспомним из раздела 9.6.6, что в приводимом графе потока любые два естественных цикла либо не пересекаются, либо один из них вложен в другой. Процесс "разбора" приводимого графа потока в иерархию циклов начинается с того, что каждый базовый блок рассматривается как область. Такие области мы будем называть *областями-листьями* (leaf region). Затем мы упорядочиваем естественные циклы изнутри наружу, т.е. начиная с наиболее внутренних циклов. При обработке цикла мы заменяем его узлом, выполняя следующие шаги.

1. *Тело* цикла L (все узлы и ребра, за исключением обратных ребер к заголовку) замещаем узлом, представляющим область R. Ребра, ведущие к заголовку L, входят теперь в узел R. Ребро из любого выхода из L замещается ребром от R в то же самое место назначения. Однако если ребро является

обратным, то оно становится петлей у R. Назовем R *областью тела* (body region).

2. Строим область R', представляющую естественный цикл L целиком, и называем R' *областью цикла* (loop region). Единственное отличие R от R' заключается в том, что последняя включает обратные ребра к заголовку цикла L. Другими словами, при замене в графе потока R областью R' все, что нам надо сделать, — это удалить ребро-петлю от R к R.

Таким образом мы сводим к отдельным узлам все бо́льшие и бо́льшие циклы, сначала с ребрами-петлями, а затем и без них. Поскольку циклы в приводимых графах потоков либо вложены, либо не пересекаются, узел области цикла может представлять все узлы естественного цикла в ряду графов потоков, строящихся описанным способом.

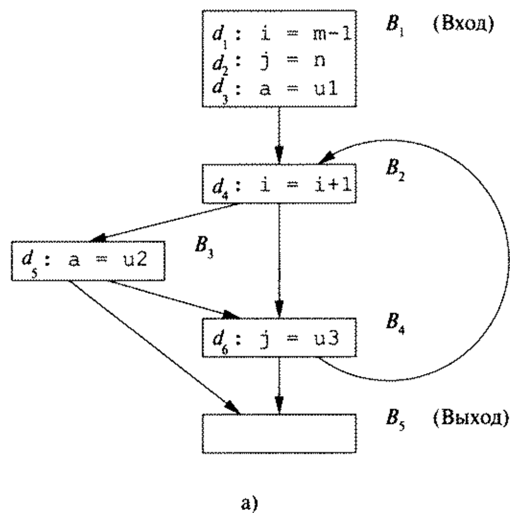
В конечном счете все естественные циклы сводятся к отдельным узлам. В этот момент граф потока может быть либо сведен к единственному узлу, либо состоять из нескольких узлов и не содержать циклов (т.е. приведенный граф потока представляет собой ациклический граф из нескольких узлов). В первом случае построение иерархии областей завершено, во втором мы строим еще одну область тела для всего графа потока.

Пример 9.47. Рассмотрим граф потока на рис. 9.48, *а*. У этого графа потока имеется одно обратное ребро, ведущее от B_4 к B_2. Иерархия областей показана на рис. 9.48, *б*. Всего имеется 8 областей.

1. Области $R_1, \ldots, R_5$ являются областями-листьями, представляющими блоки $B_1, \ldots, B_5$ соответственно. Каждый блок является выходным блоком в своей области.

2. Область тела R_6 представляет тело единственного цикла в графе потока; она состоит из областей R_2, R_3 и R_4 и межобластных ребер $B_2 \to B_3$, $B_2 \to B_4$ и $B_3 \to B_4$. Она содержит два выходных блока, B_3 и B_4, поскольку оба они имеют выходные ребра, не содержащиеся в области. На рис. 9.49, *а* показан граф потока с областью R_6, приведенной к единственному узлу. Заметим, что хотя оба ребра, $R_3 \to R_5$ и $R_4 \to R_5$, заменены ребром $R_6 \to R_5$, важно помнить, что это последнее ребро представляет два ребра. Это связано с тем, что мы будем распространять передаточные функции по этому ребру и должны знать, что выходящее из блоков B_3 и B_4 будет достигать заголовка R_5.

3. Область цикла R_7 представляет естественный цикл целиком. Она включает одну подобласть R_6 и одно обратное ребро $B_4 \to B_2$. Она также имеет два выходных узла — те же B_3 и B_4. На рис. 9.49, *б* показан граф потока после приведения естественного цикла к области R_7.



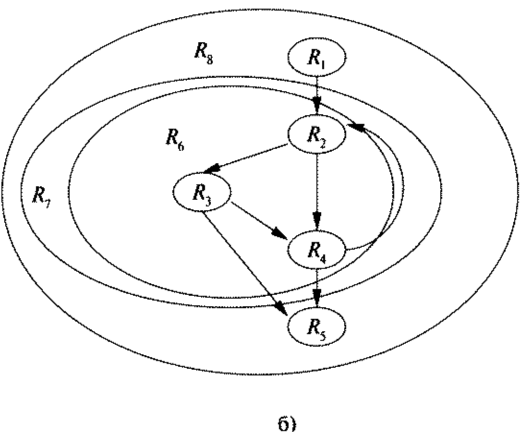


Рис. 9.48. Пример графа потока для задачи достигающих определений (а), и его иерархия областей (б)

4. Область тела R_8 представляет собой наивысшую область. Она включает три области, R_1, R_7, R_5, и три межобластных ребра, $B_1 \to B_2$, $B_3 \to B_5$ и $B_4 \to B_5$. При приведении графа потока к R_8 он становится единственным узлом. Поскольку обратных ребер к его заголовку B_1 нет, заключительный шаг, приводящий эту область тела к области цикла, не требуется. □

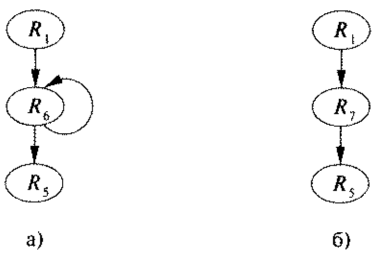


Рис. 9.49. Шаги приведения графа потока на рис. 9.48 к единственной области

Подытоживая процесс иерархической декомпозиции приводимого графа потока, мы получаем следующий алгоритм.

Алгоритм 9.48. Построение восходящего порядка областей приводимого графа потока

ВХОД: приводимый граф потока G.

ВЫХОД: список областей графа потока G, который может использоваться в задачах потоков данных на основе областей.

МЕТОД: выполняем следующие действия.

1. Начинаем со списка листьев-областей, состоящих из отдельных блоков G в произвольном порядке.
2. Неоднократно выбираем естественный цикл L, такой, что если существуют любые естественные циклы, содержащиеся в L, то тела и области этих циклов уже внесены в список. Сначала добавляем область, состоящую из тела L (т.е. L без обратных ребер к заголовку L), а затем — область цикла L.
3. Если весь граф не представляет собой естественный цикл, добавляем в конец списка область, состоящую из всего графа потока целиком. □

9.7.3 Обзор анализа на основании областей

Для каждой области R и для каждой подобласти R' в R мы вычисляем передаточную функцию $f_{R,\text{IN}[R']}$, которая суммирует влияние выполнения всех возможных путей, ведущих от входа в R ко входу в R', в пределах R. Мы говорим, что базовый блок B в R является *выходным блоком* (exit block) области R, если у него имеется выходное ребро к некоторому блоку вне R. Мы также вычисляем передаточные функции для каждого выходного блока B в R, которые обозначаются как $f_{R,\text{OUT}[B]}$ и которые суммируют влияние выполнения всех возможных путей в R, ведущих от входа в R к выходу из B.

Происхождение термина "приводимость"

Сейчас вы узнаете, почему приводимые графы получили такое название. Мы не будем доказывать этот факт, но использованное в данной книге определение "приводимого графа потока", включающее обратные ребра графа, эквивалентно нескольким определениям, в которых граф потока механически приводится к единственному узлу. Процесс свертки естественных циклов, описанный в разделе 9.7.2, — один из них. Другое интересное определение заключается в том, что приводимыми графами потоков являются те и только те графы, которые могут быть приведены к единственному узлу при помощи следующих двух преобразований:

T_1: удаление ребра-петли, входящего в узел, из которого оно вышло;

T_2: объединение узлов m и n, если узел n имеет единственного предшественника m и n не является входным для графа потока.

Затем мы обрабатываем иерархию областей, вычисляя передаточные функции для все бо́льших областей. Мы начинаем с областей, состоящих из отдельных блоков, для которых передаточная функция $f_{B,\text{IN}[B]}$ представляет собой тождественную функцию, а функция $f_{B,\text{OUT}[B]}$ является передаточной функцией для блока B. При перемещении вверх по иерархии выполняется следующее.

- Если R — область тела, то ребра, принадлежащие R, образуют ациклический граф на подобластях R. Для вычисления передаточных функций можно воспользоваться топологическим упорядочением областей.

- Если R — область цикла, то учитывается только влияние обратных ребер, ведущих к заголовку R.

В конечном счете мы достигаем вершины иерархии и вычисляем передаточные функции для области R_n, которая представляет собой весь граф целиком. Как именно выполняется каждое вычисление, вы узнаете из алгоритма 9.49.

Следующий шаг состоит в вычислении значений потока данных на входе и выходе каждого базового блока. Мы работаем с областями в обратном порядке, начиная с области R_n и опускаясь вниз по иерархии. Для каждой области мы вычисляем значения потока данных на входе. Для области R_n мы используем вызов $f_{R_n,\text{IN}[R]}(\text{IN}[\text{Вход}])$, чтобы получить значения потока данных на входе подобластей R области R_n. Эти действия повторяются, пока не будут достигнуты базовые блоки в листьях иерархии областей.

9.7.4 Необходимые предположения о передаточных функциях

Для анализа на основе областей требуется сделать определенные предположения о свойствах множества передаточных функций структуры. В частности, нам нужны три примитивные операции над передаточными функциями: композиция, сбор и замыкание. Для структур потоков данных с использованием итеративного алгоритма требуется только первая из них.

Композиция

Передаточная функция последовательности узлов может быть получена путем композиции функций, представляющих отдельные узлы. Пусть f_1 и f_2 — передаточные функции узлов n_1 и n_2. Влияние выполнения n_1 с последующим выполнением n_2 представлено как $f_2 \circ f_1$. Композиция функций рассматривалась в разделе 9.2.2, а пример использования достигающих определений был приведен в разделе 9.2.4. Вкратце повторим пройденное. Пусть gen_i и $kill_i$ обозначают множества *gen* и *kill* для f_i. Тогда

$$\begin{aligned} f_2 \circ f_1(x) &= gen_2 \cup ((gen_1 \cup (x - kill_1)) - kill_2) = \\ &= (gen_2 \cup (gen_1 - kill_2)) \cup (x - (kill_1 \cup kill_2)) \end{aligned}$$

Таким образом, множествами *gen* и *kill* для $f_2 \circ f_1$ являются $gen_2 \cup (gen_1 - kill_2)$ и $kill_1 \cup kill_2$ соответственно. Та же идея применима и для любой передаточной функции вида gen-kill. Другие передаточные функции также могут быть замкнуты, но каждый случай следует рассматривать отдельно.

Сбор

Сами по себе передаточные функции являются значениями полурешетки с оператором сбора $\wedge_f$. Сбор функций f_1 и f_2, $f_1 \wedge_f f_2$, определяется как $(f_1 \wedge_f f_2)(x) = f_1(x) \wedge f_2(x)$, где $\wedge$ — оператор сбора для значений потока данных. Оператор сбора для передаточных функций используется для объединения влияния альтернативных путей выполнения с одними и теми же конечными точками. Далее, где это не будет приводить к неоднозначности, мы будем обозначать оператор сбора для передаточных функций без индекса просто как $\wedge$. В случае структуры достигающих определений мы имеем

$$\begin{aligned} (f_1 \wedge f_2)(x) &= f_1(x) \wedge f_2(x) = \\ &= (gen_1 \cup (x - kill_1)) \cup (gen_2 \cup (x - kill_2)) = \\ &= (gen_1 \cup gen_2) \cup (x - (kill_1 \cap kill_2)) \end{aligned}$$

Иными словами, множествами *gen* и *kill* для $f_1 \wedge f_2$ являются $gen_1 \cup gen_2$ и $kill_1 \cap kill_2$ соответственно. Такая же аргументация применима к любым другим множествам gen-kill-функций.

Замыкание

Если f представляет собой передаточную функцию цикла, то f^n представляет собой влияние n проходов по циклу. В случае, когда количество итераций неизвестно, мы вынуждены считать, что цикл может выполняться 0 или больше раз. Передаточную функцию такого цикла мы представим как f^* — *замыкание* f, определяемое как

$$f^* = \bigwedge_{n \geqslant 0} f^n$$

Заметим, что f^0 должно быть тождественной функцией, поскольку она представляет собой влияние нулевого количества проходов по циклу, т.е. при отсутствии перемещения из входного узла. Если I представляет тождественную передаточную функцию, то можно записать

$$f^* = I \wedge \left(\bigwedge_{n > 0} f^n \right)$$

Предположим, что передаточная функция f в структуре достигающих определений имеет множества *gen* и *kill*. Тогда

$$\begin{aligned} f^2(x) &= f(f(x)) = \\ &= gen \cup ((gen \cup (x - kill)) - kill) = \\ &= gen \cup (x - kill) \\ f^3(x) &= f\left(f^2(x)\right) = \\ &= gen \cup (x - kill) \end{aligned}$$

И так далее: любая функция $f^n(x)$ имеет вид $gen \cup (x - kill)$. Таким образом, проход по циклу не влияет на передаточную функцию вида gen-kill. Итак,

$$\begin{aligned} f^*(x) &= I \wedge f^1(x) \wedge f^2(x) \wedge \ldots = \\ &= x \cup (gen \cup (x - kill)) = \\ &= gen \cup x \end{aligned}$$

Иначе говоря, множества *gen* и *kill* для f^* представляют собой *gen* и $\varnothing$ соответственно. Интуитивно понятно, что поскольку можно вообще миновать цикл, все достигающие определения из множества x будут достигать входа в цикл. Во всех последующих итерациях достигающие определения включают таковые из множества *gen*.

9.7.5 Алгоритм анализа на основе областей

Приведенный далее алгоритм решает задачу анализа потока данных в прямом направлении на приводимом графе потока в соответствии с некоторой структурой, удовлетворяющей предположениям из раздела 9.7.4. Вспомним, что $f_{R,\text{IN}[R']}$

и $f_{R,\text{OUT}[B]}$ обозначают передаточные функции, которые преобразуют значения потока данных на входе в область R в корректные значения на входе в подобласть R' и на выходе из блока B соответственно.

Алгоритм 9.49. Анализ на основе областей

Вход: структура потока данных со свойствами, изложенными в разделе 9.7.4, и приводимый граф потока G.

Выход: значения потока данных $\text{IN}[B]$ для каждого блока B в G.

Метод: выполняем следующие действия.

1. Используем алгоритм 9.48 для построения восходящей последовательности областей G, скажем, $R_1, R_2, \ldots, R_n$, где R_n – область верхнего уровня.

2. Выполняем восходящий анализ для вычисления передаточных функций, которые подытоживают влияние выполнения области. Для каждой области $R_1, R_2, \ldots, R_n$ в восходящем порядке выполняем следующее.

   а) Если R представляет собой область-лист, соответствующую блоку B, то положим $f_{R,\text{IN}[B]} = I$ и $f_{R,\text{OUT}[B]} = f_B$ – передаточной функции, связанной с блоком B.

   б) Если R – область тела, то выполняем вычисления, показанные на рис. 9.50, *а*.

   в) Если R – область цикла, то выполняем вычисления, показанные на рис. 9.50, *б*.

3. Выполняем нисходящий проход для поиска значений потока данных в начале каждой области.

   а) $\text{IN}[R_n] = \text{IN}[\text{Вход}]$.

   б) Для каждой области $R \in \{R_1, \ldots, R_{n-1}\}$ в нисходящем порядке вычисляем $IN[R] = f_{R',IN[R]}(IN[R'])$, где R' – область, непосредственно охватывающая область R.

Сначала подробно рассмотрим, как работает восходящий анализ. В строке 1 на рис. 9.50, *а* мы посещаем подобласти области тела в некотором топологическом порядке. В строке 2 вычисляется передаточная функция, представляющая все возможные пути от заголовка R к заголовку S. Затем в строках 3 и 4 мы вычисляем передаточные функции, представляющие все возможные пути от заголовка R к выходам из R, т.е. к выходам из всех блоков, которые имеют преемников за пределами S. Заметим, что все предшественники B' в R должны быть в областях, которые предшествуют S в топологическом порядке, построенном в строке 1. Таким образом, $f_{R,\text{OUT}[B']}$ будет уже вычислено в строке 4 предыдущей итерации внешнего цикла.

```
1)  for (каждая подобласть S, непосредственно содержащаяся в R,
            в топологическом порядке) {
2)      f_{R,IN[S]} = ∧_{Предшественники B заголовка S из R} f_{R,OUT[B]};
        /* Если S – заголовок области R, то f_{R,IN[S]} –
            сбор "по ничему", т.е. тождественная функция */
3)      for (каждый выходной блок B в S)
4)          f_{R,OUT[B]} = f_{S,OUT[B]} ∘ f_{R,IN[S]};
    }
```

а) Построение передаточной функции для области тела R

```
1)  Пусть S – область тела, непосредственно вложенная в R
        (т.е. S представляет собой R без обратных ребер из R
        в заголовок R;
2)  f_{R,IN[S]} = (∧_{Предшественники B заголовка S в R} f_{S,OUT[B]})*;
3)  for (каждого выходного блока B из R)
4)      f_{R,OUT[B]} = f_{S,OUT[B]} ∘ f_{R,IN[S]};
```

б) Построение передаточной функции для области цикла R'

Рис. 9.50. Детали вычислений потоков данных на основе областей

Для областей циклов мы выполняем шаги в строках 1–4 на рис. 9.50, *б*. В строке 2 вычисляется влияние обхода по циклу области тела S нуль или больше раз. В строках 3 и 4 вычисляется влияние выходов из цикла после одной или более итераций.

В нисходящем проходе алгоритма на шаге 3, *а* сначала устанавливаются граничные условия на входе в верхнюю область. Затем, если R содержится в R' непосредственно, для вычисления $\text{IN}[R]$ можно просто применить передаточную функцию $f_{R',\text{IN}[R]}$ к значению потока данных $\text{IN}[R']$. □

Пример 9.50. Применим алгоритм 9.49 для поиска достигающих определений в графе потока на рис. 9.48, *а*. На шаге 1 создается восходящий порядок, в котором посещаются области. Этот порядок определяет нумерацию индексов областей $R_1, R_2, \ldots, R_n$.

Значения множеств *gen* и *kill* для пяти базовых блоков приведены ниже.

B	B_1	B_2	B_3	B_4	B_5
gen_B	$\{d_1, d_2, d_3\}$	$\{d_4\}$	$\{d_5\}$	$\{d_6\}$	$\varnothing$
$kill_B$	$\{d_4, d_5, d_6\}$	$\{d_1\}$	$\{d_3\}$	$\{d_2\}$	$\varnothing$

Вспомним упрощенные правила для передаточных функций вида *gen-kill* из раздела 9.7.4.

- Для получения сбора передаточных функций вычисляются объединение множеств *gen* и пересечение множеств *kill*.
- Для вычисления композиции передаточных функций вычисляются объединения множеств *gen* и *kill*. Однако как исключение выражение, которое генерируется первой функцией, при этом не генерируется второй и уничтожается ею, *не* входит в множество *gen* результата.
- Для получения замыкания передаточной функции *gen* оставляется неизменным, а *kill* становится пустым множеством.

Первыми пятью областями $R_1, \ldots, R_5$ являются соответственно базовые блоки $B_1, \ldots, B_5$. Функция $f_{R_i,\text{IN}[B_i]}$ для $1 \leqslant i \leqslant 5$ является тождественной функцией, а функция $f_{R_i,\text{OUT}[B_i]}$ — передаточной функцией для блока B_i:

$$f_{B_i,\text{OUT}[B_i]}(x) = (x - \mathit{kill}_{B_i}) \cup \mathit{gen}_{B_i}$$

Остальные передаточные функции, построенные на шаге 2 алгоритма 9.49, подытожены на рис. 9.51. Область R_6, состоящая из областей R_2, R_3 и R_4, представляет тело цикла и, таким образом, не включает обратное ребро $B_4 \to B_2$. Порядок обработки этих областей представляет собой единственный топологический порядок: R_2, R_3, R_4. R_2 не имеет предшественников в пределах R_6; вспомним, что ребро $B_4 \to B_2$ выходит за пределы R_6. Таким образом, $f_{R_6,\text{IN}[B_2]}$ представляет собой тождественную функцию[12], а $f_{R_6,\text{OUT}[B_2]}$ — передаточную функцию для блока B_2.

Заголовок области B_3 имеет одного предшественника в R_6, а именно — R_2. Передаточная функция к его входу представляет собой просто передаточную функцию к выходу из B_2, $f_{R_6,\text{OUT}[B_2]}$, которая уже вычислена. Для получения передаточной функции к выходу из B_3 мы вычисляем композицию этой функции с передаточной функцией B_3 в ее собственной области.

Наконец, поскольку и B_2, и B_3 являются предшественниками B_4, заголовка R_4, для вычисления передаточной функции ко входу в R_4 мы должны вычислить

$$f_{R_6,\text{OUT}[B_2]} \wedge f_{R_6,\text{OUT}[B_3]}$$

Для получения требующейся функции $f_{R_6,\text{OUT}[B_4]}$ следует вычислить композицию этой передаточной функции и функции $f_{R_4,\text{OUT}[B_4]}$. Обратите внимание, например, что d_3 не уничтожается в этой передаточной функции, поскольку путь $B_2 \to B_4$ не переопределяет переменную a.

Теперь рассмотрим область цикла R_7. Она содержит только одну подобласть R_6, которая представляет тело ее цикла. Поскольку существует только одно обратное ребро $B_4 \to B_2$ к заголовку R_6, передаточная функция, представляющая

[12] Строго говоря, имеется в виду функция $f_{R_6,\text{IN}[R_2]}$, но когда область наподобие R_2 представляет собой единственный блок, в таком контексте более понятным оказывается использование имени блока вместо имени области.

	Передаточная функция	*gen*	*kill*
R_6	$f_{R_6,\text{IN}[R_2]} = I$	$\varnothing$	$\varnothing$
	$f_{R_6,\text{OUT}[B_2]} = f_{R_2,\text{OUT}[B_2]} \circ f_{R_6,\text{IN}[R_2]}$	$\{d_4\}$	$\{d_1\}$
	$f_{R_6,\text{IN}[R_3]} = f_{R_6,\text{OUT}[B_2]}$	$\{d_4\}$	$\{d_1\}$
	$f_{R_6,\text{OUT}[B_3]} = f_{R_3,\text{OUT}[B_3]} \circ f_{R_6,\text{IN}[R_3]}$	$\{d_4, d_5\}$	$\{d_1, d_3\}$
	$f_{R_6,\text{IN}[R_4]} = f_{R_6,\text{OUT}[B_2]} \wedge f_{R_6,\text{OUT}[B_3]}$	$\{d_4, d_5\}$	$\{d_1\}$
	$f_{R_6,\text{OUT}[B_4]} = f_{R_4,\text{OUT}[B_4]} \circ f_{R_7,\text{IN}[R_6]}$	$\{d_4, d_5, d_6\}$	$\{d_1, d_2\}$
R_7	$f_{R_7,\text{IN}[R_6]} = f^*_{R_6,\text{OUT}[B_4]}$	$\{d_4, d_5, d_6\}$	$\varnothing$
	$f_{R_7,\text{OUT}[B_3]} = f_{R_6,\text{OUT}[B_3]} \circ f_{R_7,\text{IN}[R_6]}$	$\{d_4, d_5, d_6\}$	$\{d_1, d_3\}$
	$f_{R_7,\text{OUT}[B_4]} = f_{R_6,\text{OUT}[B_4]} \circ f_{R_7,\text{IN}[R_6]}$	$\{d_4, d_5, d_6\}$	$\{d_1, d_2\}$
R_8	$f_{R_8,\text{IN}[R_1]} = I$	$\varnothing$	$\varnothing$
	$f_{R_8,\text{OUT}[B_1]} = f_{R_1,\text{OUT}[B_1]}$	$\{d_1, d_2, d_3\}$	$\{d_4, d_5, d_6\}$
	$f_{R_8,\text{IN}[R_7]} = f_{R_8,\text{OUT}[B_1]}$	$\{d_1, d_2, d_3\}$	$\{d_4, d_5, d_6\}$
	$f_{R_8,\text{OUT}[B_3]} = f_{R_7,\text{OUT}[B_3]} \circ f_{R_8,\text{IN}[R_7]}$	$\{d_2, d_4, d_5, d_6\}$	$\{d_1, d_3\}$
	$f_{R_8,\text{OUT}[B_4]} = f_{R_7,\text{OUT}[B_4]} \circ f_{R_8,\text{IN}[R_7]}$	$\{d_3, d_4, d_5, d_6\}$	$\{d_1, d_2\}$
	$f_{R_8,\text{IN}[R_5]} = f_{R_8,\text{OUT}[B_3]} \wedge f_{R_8,\text{OUT}[B_4]}$	$\{d_2, d_3, d_4, d_5, d_6\}$	$\{d_1\}$
	$f_{R_8,\text{OUT}[B_5]} = f_{R_5,\text{OUT}[B_5]} \circ f_{R_8,\text{IN}[R_5]}$	$\{d_2, d_3, d_4, d_5, d_6\}$	$\{d_1\}$

Рис. 9.51. Вычисление передаточных функций для графа потока на рис. 9.48, а с использованием анализа, основанного на областях

выполнение тела цикла 0 или более раз, — $f^*_{R_6,\text{OUT}[B_4]}$: множество *gen* представляет собой $\{d_4, d_5, d_6\}$, а множество *kill* — пустое. Из области R_7 имеется два выхода — базовые блоки B_3 и B_4. Таким образом, для получения соответствующих передаточных функций R_7 следует вычислить композиции упомянутой передаточной функции с каждой из передаточных функций R_6. Обратите внимание, например, что d_6 находится в множестве *gen* функции $f_{R_7,\text{OUT}[B_3]}$ из-за наличия путей наподобие $B_2 \to B_4 \to B_2 \to B_3$ или даже $B_2 \to B_3 \to B_4 \to B_2 \to B_3$.

Наконец, рассмотрим R_8, граф потока, целиком. Его подобластями являются R_1, R_7 и R_5, которые мы будем рассматривать в топологическом порядке. Как и ранее, передаточная функция $f_{R_8,\text{IN}[B_1]}$ представляет собой просто тождественную функцию, а передаточная функция $f_{R_8,\text{OUT}[B_1]}$ представляет собой просто $f_{R_1,\text{OUT}[B_1]}$, которая, в свою очередь, является f_{B_1}.

Заголовок R_7, представляющий собой B_2, имеет единственного предшественника, B_1, так что передаточная функция к его входу представляет собой просто передаточную функцию на выходе из B_1 в области R_8. Для получения соответствующих передаточных функций в R_8 мы вычисляем композиции $f_{R_8,\text{OUT}[B_1]}$

с передаточными функциями к выходам B_3 и B_4 в R_7. Последней рассмотрим область R_5. Ее заголовок B_5 имеет два предшественника в R_8, а именно — B_3 и B_4. Следовательно, для получения $f_{R_8,\text{IN}[B_5]}$ надо вычислить $f_{R_8,\text{OUT}[B_3]} \wedge f_{R_8,\text{OUT}[B_4]}$. Поскольку передаточная функция базового блока B_5 представляет собой тождественную функцию, $f_{R_8,\text{OUT}[B_5]} = f_{R_8,\text{IN}[B_5]}$.

На шаге 3 вычисляются действительные достигающие определения на основе передаточных функций. На шаге 3, *а* $\text{IN}[R_8] = \emptyset$, поскольку в начале программы достигающие определения отсутствуют. На рис. 9.52 показано, как на шаге 3, *б* вычисляются оставшиеся значения потока данных. Этот шаг начинается с подобласти R_8. Поскольку передаточные функции от начала R_8 к началу каждой подобласти уже вычислены, значение потока данных в начале каждой подобласти получается путем однократного применения передаточной функции. Мы повторяем эти шаги до тех пор, пока не получим значения потока данных для каждой области-листа, которые представляют собой отдельные базовые блоки. Заметим, что значения потока данных, показанные на рис. 9.52, в точности те же, которые мы получили бы при применении итеративного анализа потока данных к тому же графу потока, как, естественно, и должно быть. □

$$
\begin{aligned}
\text{IN}[R_8] &= \emptyset \\
\text{IN}[R_1] &= f_{R_8,\text{IN}[R_1]}(\text{IN}[R_8]) &&= \emptyset \\
\text{IN}[R_7] &= f_{R_8,\text{IN}[R_7]}(\text{IN}[R_8]) &&= \{d_1, d_2, d_3\} \\
\text{IN}[R_5] &= f_{R_8,\text{IN}[R_5]}(\text{IN}[R_8]) &&= \{d_2, d_3, d_4, d_5, d_6\} \\
\text{IN}[R_6] &= f_{R_7,\text{IN}[R_6]}(\text{IN}[R_7]) &&= \{d_1, d_2, d_3, d_4, d_5, d_6\} \\
\text{IN}[R_4] &= f_{R_6,\text{IN}[R_4]}(\text{IN}[R_6]) &&= \{d_2, d_3, d_4, d_5, d_6\} \\
\text{IN}[R_3] &= f_{R_6,\text{IN}[R_3]}(\text{IN}[R_6]) &&= \{d_2, d_3, d_4, d_5, d_6\} \\
\text{IN}[R_2] &= f_{R_6,\text{IN}[R_2]}(\text{IN}[R_6]) &&= \{d_1, d_2, d_3, d_4, d_5, d_6\}
\end{aligned}
$$

Рис. 9.52. Последние шаги анализа потока на основании областей

9.7.6 Обработка неприводимых графов потоков

Если ожидается, что в программе, обрабатываемой компилятором или другим специализированным инструментарием, неприводимые графы потоков будут распространенным явлением, то мы рекомендуем использовать итеративный подход вместо анализа, основанного на иерархии. Однако если требуется быть готовым только к редким неприводимым графам, то в этом случае вполне адекватен метод "расщепления узлов".

Если граф потока неприводим, мы обнаружим, что получающийся в результате граф областей содержит циклы, но не имеет обратных ребер, так что дальнейший

разбор этого графа невозможен. Типичная ситуация показана на рис. 9.53, *а*, где граф имеет ту же структуру, что и граф на рис. 9.45, хотя узлы на рис. 9.53 в действительности являются сложными областями (что видно из наличия внутри них узлов меньшего размера).

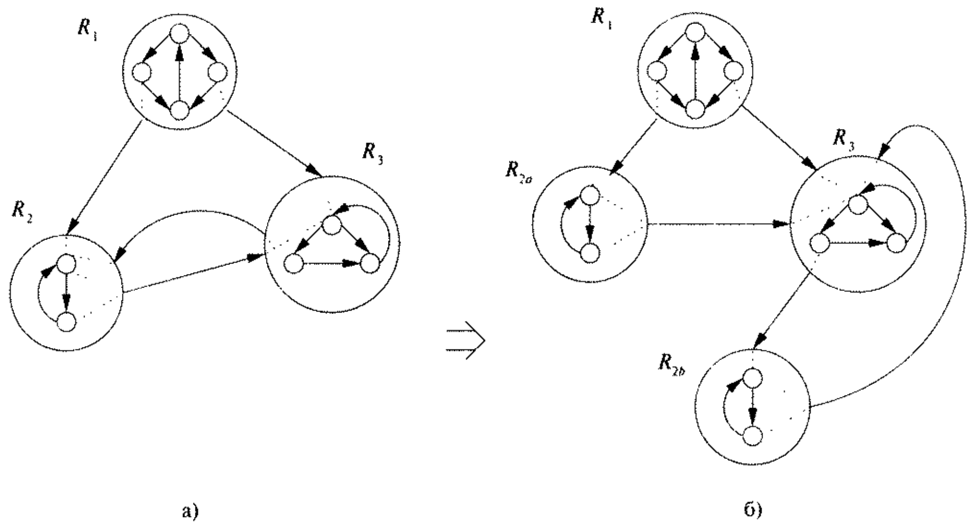


Рис. 9.53. Дублирование области делает неприводимый граф приводимым

Мы выбираем некоторую область R, у которой более одного предшественника и которая не является заголовком всего графа потока. Если у R имеется k предшественников, создаем k копий всего графа R и соединяем каждого предшественника заголовка R со своей копией графа R. Вспомните, что иметь предшественника вне области может только заголовок области. Оказывается, хотя мы и не будем доказывать этот факт, что такое расщепление узлов приводит после идентификации новых обратных ребер и построения их областей к снижению количества областей как минимум на единицу. Получающийся граф может остаться неприводимым, но чередование фаз расщепления с фазами обнаружения и свертывания в области естественных циклов в конечном итоге дает одну область, т.е. граф потока оказывается приведенным.

Пример 9.51. Расщепление, показанное на рис. 9.53, *б*, превращает ребро $R_{2b} \rightarrow R_3$ в обратное ребро, поскольку R_3 теперь доминирует над R_{2b}. Эти две области можно скомбинировать в одну область тела. Таким образом, мы приводим весь граф к одной области. В общем случае могут потребоваться дополнительные расщепления, а в наихудшем случае общее количество базовых блоков может экспоненциально зависеть от количества блоков в исходном графе потока. □

Следует также подумать и о том, как результат анализа потока данных в расщепленном графе потока соотносится с интересующим нас ответом для исходного графа потока. Существует два подхода.

1. Расщепление областей может благотворно сказаться на процессе оптимизации, и мы можем просто изменить граф потока так, чтобы он имел копии некоторых блоков. Поскольку в каждый дублированный блок можно попасть только по некоторому подмножеству путей, достигающих оригинала, значения потока данных в этих дублированных блоках будут, вообще говоря, содержать более специализированную информацию по сравнению с доступной в оригинале. Например, каждого дублированного блока может достигать меньшее количество определений, чем количество определений, достигающих исходного блока.

2. Если мы хотим сохранить исходный граф потока, без расщепления, то после анализа расщепленного графа потока мы рассматриваем каждый расщепленный блок B и соответствующее ему множество блоков $B_1, B_2, \ldots, B_k$. Можно вычислить $\text{IN}[B] = \text{IN}[B_1] \wedge \text{IN}[B_2] \wedge \cdots \wedge \text{IN}[B_k]$; значения OUT вычисляются аналогично.

9.7.7 Упражнения к разделу 9.7

Упражнение 9.7.1. Для графа потока на рис. 9.10 (см. упражнения к разделу 9.1) выполните следующее.

а) Найдите все возможные области. Области, состоящие из единственного узла и без ребер, можно не учитывать.

б) Найдите множество вложенных областей, построенное алгоритмом 9.52.

в) Выполните приведение графа потока при помощи преобразований T_1 и T_2, описанных во врезке "Происхождение термина «приводимость»" в разделе 9.7.2.

Упражнение 9.7.2. Повторите упражнение 9.7.1 для следующих графов потоков.

а) Граф на рис. 9.3.

б) Граф на рис. 8.9.

в) Созданный вами граф из упражнения 8.4.1.

г) Созданный вами граф из упражнения 8.4.2.

Упражнение 9.7.3. Докажите, что каждый естественный цикл является областью.

!! **Упражнение 9.7.4.** Покажите, что граф потока приводим тогда и только тогда, когда он может быть преобразован в единственный узел с использованием

а) операций T_1 и T_2, описанных во врезке "Происхождение термина «приводимость»" в разделе 9.7.2;

б) определения области, введенного в разделе 9.7.2.

! **Упражнение 9.7.5.** Покажите, что выполнение расщепления узлов к неприводимому графу потока с последующим применением приведения $T_1 - T_2$ к получившемуся расщепленному графу дает строго меньшее количество узлов, чем было до выполнения указанных действий.

! **Упражнение 9.7.6.** Что произойдет, если для приведения полного ориентированного графа поочередно применять расщепление узлов и приведение $T_1 - T_2$?

9.8 Символический анализ

В этом разделе мы используем символический анализ для иллюстрации использования анализа на основе областей. В этом анализе мы символически отслеживаем значения переменных программы как выражения от входных переменных и других переменных, которые мы назовем *ссылочными переменными* (reference variables). Выражение переменных через одно и то же множество ссылочных переменных выявляет их взаимоотношения. Символический анализ может использоваться для разных целей, таких как оптимизация, распараллеливание и анализ для понимания программы.

Пример 9.52. Рассмотрим простой пример программы на рис. 9.54. Здесь x используется как единственная ссылочная переменная. Символический анализ обнаруживает, что y имеет значение $x - 1$, а z — значение $x - 2$ после соответствующих присваиваний в строках 2 и 3. Эта информация полезна, например, при выяснении, выполняют ли инструкции в строках 4 и 5 запись в разные ячейки памяти (и, таким образом, могут ли они выполняться параллельно). Кроме того, можно утверждать, что условие $z > x$ никогда не будет истинным, что позволяет оптимизатору удалить условную инструкцию из строк 6 и 7. □

9.8.1 Аффинные выражения ссылочных переменных

Поскольку невозможно создать краткие аналитические символические выражения для всех вычисляемых значений, мы выбираем абстрактную область определения и используем приближения вычислений наиболее близкими выражениями из этой области. Мы уже встречались ранее с примером подобной стратегии при рассмотрении распространения констант. В этом случае наша абстрактная

```
1) x = input();
2) y = x - 1;
3) z = y - 1;
4) A[x] = 10;
5) A[y] = 11;
6) if (z > x)
7)     z = x;
```

Рис. 9.54. Пример программы, поясняющий символический анализ

область определения состояла из констант, и специальный символ NAC использовался в случае, когда обнаруживалось, что переменная не является константой.

Представленный здесь символический анализ по возможности выражает значения как *аффинные* выражения ссылочных переменных. Выражение является аффинным по отношению к переменным $v_1, v_2, \ldots, v_n$, если оно может быть выражено как $c_0 + c_1v_1 + \cdots + c_nv_n$, где $c_0, c_1, \ldots, c_n$ — константы. Такие выражения неформально известны как линейные. Строго говоря, аффинное выражение является линейным, только если c_0 равно 0. Аффинные выражения интересуют нас потому, что они часто используются для индексирования массивов в циклах — такая информация весьма полезна при оптимизации и распараллеливании. Существенно больше информации по этой теме имеется в главе 11.

Переменные индукции

Вместо программных переменных в качестве ссылочных аффинные выражения могут использовать количество итераций в цикле. Переменные, значения которых могут быть выражены как $c_1i + c_0$, где i — количество итераций ближайшего охватывающего цикла, известны как *переменные индукции* (induction variables), или индуктивные переменные.

Пример 9.53. Рассмотрим следующий фрагмент кода:

```
for (m = 10; m < 20; m++ )
    { x = m * 3; A[x] = 0; }
```

Предположим, что мы ввели в цикл переменную, скажем, i, подсчитывающую количество выполненных итераций. На первой итерации цикла значение i равно 0, на второй — 1 и т.д. Переменную m можно записать как аффинное выражение от i, а именно как $m = i + 10$. Переменная x, значение которой — $3m$, в процессе последовательных итераций цикла принимает значения 30, 33, $\ldots$, 57. Таким образом, аффинное выражение для x имеет вид $x = 30 + 3i$. Мы заключаем, что и m, и x являются переменными индукции этого цикла. □

Запись переменных в виде аффинных выражений индексов циклов дает возможность выполнять некоторые преобразования, в частности вычислять ряд зна-

чений переменной индукции при помощи сложения, а не умножения. Такое преобразование известно как снижение стоимости и рассматривалось в разделах 8.7 и 9.1. Например, можно удалить умножение x = m*3 из цикла в примере 9.53, переписав код следующим образом:

```
x = 27;
for (m = 10; m < 20; m++ )
    { x = x + 3; A[x] = 0; }
```

Заметим, кроме того, что адреса ячеек, в которые записывается нулевое значение, а именно $\&A{+}30, \&A{+}33, \ldots, \&A{+}57$, также представляют собой аффинное выражение от индекса цикла. Фактически этот ряд целых чисел — единственное, что должно быть вычислено, так что можно полностью удалить из кода одну из переменных. Так, рассматриваемый код можно переписать просто как

```
for (x = &A+30; x <= &A+57; x += 3 )
    *x = 0
```

Помимо ускорения вычислений, символический анализ полезен и для распараллеливания. Когда индексы массива в цикле представляют собой аффинные выражения от индексов цикла, можно сделать определенные выводы о взаимоотношениях данных, к которым выполняется обращение в разных итерациях. Например, можно выяснить, что в каждой итерации запись ведется в новые ячейки памяти, а следовательно, все итерации цикла могут выполняться параллельно на разных процессорах. Такая информация используется в главах 10 и 11 для выделения параллельности в последовательных программах.

Другие ссылочные переменные

Если переменная не является линейной функцией уже выбранных ссылочных переменных, то ее значение можно рассматривать как ссылочное для будущих операций. Например, рассмотрим следующий фрагмент кода:

```
a = f();
b = a + 10;
c = a + 11;
```

Значение, хранящееся в a, после вызова функции не может быть выражено как линейная функция каких-либо ссылочных переменных, но может использоваться в качестве таковой для последующих инструкций. Например, используя a в качестве ссылочной переменной, можно выяснить, что c на единицу больше b в конце приведенного фрагмента.

Пример 9.54. В этом разделе рабочим примером служит код на рис. 9.55. Внутренний и внешний циклы легко различимы и понятны, поскольку переменные f

и g не изменяются нигде, кроме самой конструкции **for**. Таким образом, можно заменить f и g ссылочными переменными i и j, которые подсчитывают количество итераций внешнего и внутреннего циклов соответственно. Иначе говоря, мы можем положить $f = i + 99$ и $g = j + 9$ и подставить эти выражения вместо f и g. При трансляции в промежуточный код можно воспользоваться тем фактом, что каждый цикл выполняется как минимум однократно, и отложить проверки $i \leqslant 100$ и $j \leqslant 10$ до концов циклов.

```
1)  a = 0;
2)  for (f = 100; f < 200; f++) {
3)      a = a + 1;
4)      b = 10 * a;
5)      c = 0;
6)      for (g = 10; g < 20; g++) {
7)          d = b + c;
8)          c = c + 1;
        }
    }
```

Рис. 9.55. Исходный код к примеру 9.54

На рис. 9.56 показан граф потока для кода на рис. 9.55 после введения переменных i и j и рассмотрения циклов for в качестве циклов repeat.

Как выясняется, переменные a, b, c и d являются переменными индукции. Последовательность значений, присваиваемых этим переменным в каждой строке кода, показана на рис. 9.57. Как вы увидите, для этих переменных можно найти аффинные выражения от ссылочных переменных i и j, т.е. в строке 4 $a = j$, в строке 7 $d = 10j + j - 1$, а в строке 8 $c = j$. □

9.8.2 Формулировка задачи потока данных

Данный анализ находит аффинные выражения от ссылочных переменных, введенных 1) для подсчета количества выполненных в каждом цикле итераций и 2) для хранения значений на входе в области там, где это необходимо. Этот анализ также находит переменные индукции, инварианты циклов и константы — как вырожденные аффинные выражения. Заметим, что данному анализу не под силу найти все константы, поскольку он отслеживает только аффинные выражения от ссылочных переменных.

Значения потоков данных: символические отображения

Областью определения значений потоков данных в данном анализе являются символические отображения (symbolic maps), представляющие собой функции,

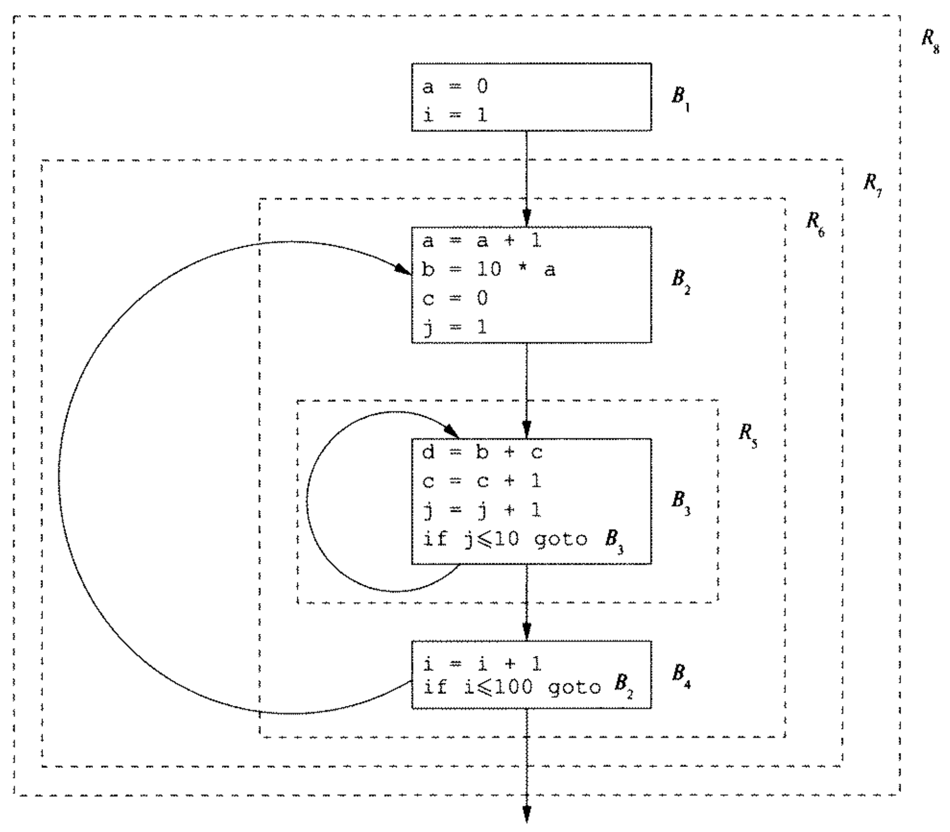


Рис. 9.56. Граф потока и его иерархия областей к примеру 9.54

Строка	Переменная	$i = 1$ $j = 1, \ldots, 10$	$i = 2$ $j = 1, \ldots, 10$	$1 \leqslant i \leqslant 100$ $j = 1, \ldots, 10$	$i = 100$ $j = 1, \ldots, 10$
2	a	1	2	i	100
3	b	10	20	$10i$	1000
7	c	$10, \ldots, 19$	$20, \ldots, 29$	$10i, \ldots, 10i + 9$	$1000, \ldots, 1009$
8	d	$1, \ldots, 10$	$1, \ldots, 10$	$1, \ldots, 10$	$1, \ldots, 10$

Рис. 9.57. Последовательность значений переменных в точках программы в примере 9.54

отображающие каждую переменную в программе на значение. Это значение является либо аффинной функцией ссылочных переменных, либо специальным символом NAA, представляющим неаффинное выражение. Если имеется только одна переменная, то нижнее значение полурешетки представляет собой отображение переменной на NAA. Полурешетка для n переменных представляет собой просто произведение отдельных полурешеток. Для указания нижнего элемента полуре-

шетки, который отображает все переменные на NAA, используется обозначение m_{NAA}. Можно определить символическое отображение всех переменных на неизвестное значение, которое будет верхним элементом полурешетки, как мы делали для распространения констант. Однако в анализе на основе областей верхние значения нам не требуются.

Пример 9.55. Символические отображения, связанные с каждым блоком кода из примера 9.54, показаны на рис. 9.58. Позже вы узнаете, как находятся эти отображения, — они являются результатом анализа потока данных на основе областей графа потока на рис. 9.56.

m	$m(a)$	$m(b)$	$m(c)$	$m(d)$
IN $[B_1]$	NAA	NAA	NAA	NAA
OUT $[B_1]$	0	NAA	NAA	NAA
IN $[B_2]$	$i-1$	NAA	NAA	NAA
OUT $[B_2]$	i	$10i$	0	NAA
IN $[B_3]$	i	$10i$	$j-1$	NAA
OUT $[B_3]$	i	$10i$	j	$10i+j-1$
IN $[B_4]$	i	$10i$	j	$10i+j-1$
OUT $[B_4]$	$i-1$	$10i-10$	j	$10i+j-11$

Рис. 9.58. Символические отображения программы из примера 9.54

Символическое отображение, связанное с входом программы, — m_{NAA}. На выходе из B_1 значение a устанавливается равным 0. На входе в блок B_2 на первой итерации a имеет значение 0 и увеличивается на 1 в каждой последующей итерации внешнего цикла. Таким образом, a равно $i-1$ на входе в i-ю итерацию и i в ее конце. Символическое отображение на входе в базовый блок B_2 отображает переменные b, c, d на NAA, поскольку при входе во внешний цикл значения этих переменных неизвестны (их значения зависят от количества итераций внешнего цикла). Символическое отображение на выходе из B_2 отражает инструкции присваивания переменным a, b и c в этом блоке. После того как установлена корректность отображений на рис. 9.58, каждое из присваиваний a, b, c и d на рис. 9.55 можно заменить соответствующим аффинным выражением, т.е. код на рис. 9.55 можно заменить кодом, приведенным на рис. 9.59. □

Передаточная функция инструкции

Передаточная функция в этой задаче потока данных отображает символическое отображение на символическое отображение. Для вычисления передаточной функции инструкции присваивания мы интерпретируем семантику присваивания и определяем, может ли присваиваемая переменная быть выражена в виде аффин-

```
1)  a = 0;
2)  for (i = 1; i <= 100; i++) {
3)      a = i;
4)      b = 10 * i;
5)      c = 0;
6)      for (j = 1; j <= 10; j++) {
7)          d = 10 * i + j - 1;
8)          c = j;
        }
    }
```

Рис. 9.59. Код на рис. 9.55, присваивания в котором заменены аффинными выражениями от ссылочных переменных i и j

Предостережение о передаточных функциях для отображений значений

В нашем определении передаточных функций от символических отображений имеется тонкость, заключающаяся в выборе способа выражения результата вычисления. Если m — отображение входа передаточной функции, $m(x)$ просто означает "какое бы значение переменная x ни принимала на входе". Мы пытаемся выразить результат передаточной функции как аффинное выражение от ссылочных переменных, используемых входным отображением.

Вы должны правильно интерпретировать выражения наподобие $f(m)(x)$, где f — передаточная функция, m — отображение, а x — переменная. Как принято в математике, мы применяем функции слева направо, так что сначала вычисляется значение $f(m)$, которое представляет собой отображение. Поскольку отображение является функцией, оно затем может быть применено к переменной x для получения значения.

ного выражения значений справа от знака присваивания. Значения всех остальных переменных остаются неизменными.

Передаточная функция инструкции s, обозначаемая как f_s, определяется следующим образом.

1. Если s не является инструкцией присваивания, то f_s является тождественной функцией.

2. Если s — инструкция присваивания переменной x, то

$$f_s(m)(x) = \begin{cases} m(v) & \text{для всех переменных } v \neq x; \\ c_0 + c_1 m(y) + c_2 m(z) & \text{если } x \text{ присваивается } c_0 + c_1 y + c_2 z, \\ & (c_1 = 0, \text{ или } m(y) \neq \text{NAA}) \text{ и} \\ & (c_2 = 0, \text{ или } m(z) \neq \text{NAA}); \\ NAA & \text{в противном случае.} \end{cases}$$

Выражение $c_0 + c_1 m(y) + c_2 m(z)$ предназначено для представления всех возможных видов выражений, включающих произвольные переменные y и z из правой части присваивания переменной x, которые являются аффинными преобразованиями значений этих переменных. Во многих случаях одна или несколько из констант c_0, c_1 и c_2 равны 0.

Пример 9.56. В случае присваивания `x = y+z` $c_0 = 0$ и $c_1 = c_2 = 1$. Если присваивание — `x = y/5`, то $c_0 = c_2 = 0$, а $c_1 = 1/5$. □

Композиция передаточных функций

Для вычисления $f_2 \circ f_1$, где f_1 и f_2 определены в терминах входного отображения m, мы подставляем вместо значения $m(v_i)$ в определении f_2 определение $f_1(m)(v_i)$. Все операторы над значениями NAA заменяются NAA, т.е.

1. если $f_2(m)(v) = \text{NAA}$, то $(f_2 \circ f_1)(m)(v) = \text{NAA}$;

2. если $f_2(m)(v) = c_0 + \sum_i c_i m(v_i)$, то

$$(f_2 \circ f_1)(m)(v) = \begin{cases} \text{NAA}, & \text{если } f_1(m)(v_i) = \text{NAA} \\ & \text{для некоторых } i \neq 0,\ c_i \neq 0 \\ c_0 + \sum_i c_i f_1(m)(v_i) & \text{в противном случае.} \end{cases}$$

Пример 9.57. Передаточные функции блоков из примера 9.54 могут быть вычислены путем композиции передаточных функций инструкций, составляющих блоки. Эти передаточные функции показаны на рис. 9.60. □

Решение задачи потока данных

Для входных и выходных значений потока данных блока B_3 на j-й итерации внутреннего цикла и на i-й итерации внешнего цикла используем обозначения $\text{IN}_{i,j}[B_3]$ и $\text{OUT}_{i,j}[B_3]$. Для тех же значений на i-й итерации внешнего цикла в других блоках используем обозначения $\text{IN}_i[B_k]$ и $\text{OUT}_i[B_k]$. Можно также показать, что символические отображения, показанные на рис. 9.58, удовлетворяют ограничениям, навязанным передаточными функциями и показанным на рис. 9.61.

f	$f(m)(a)$	$f(m)(b)$	$f(m)(c)$	$f(m)(d)$
f_{B_1}	0	$m(b)$	$m(c)$	$m(d)$
f_{B_2}	$m(a)+1$	$10m(a)+10$	0	$m(d)$
f_{B_3}	$m(a)$	$m(b)$	$m(c)+1$	$m(b)+m(c)$
f_{B_4}	$m(a)$	$m(b)$	$m(c)$	$m(d)$

Рис. 9.60. Передаточные функции из примера 9.54

$$
\begin{array}{rll}
\text{OUT}[B_k] = f_B(\text{IN}[B_k]), & \text{для всех } B_k \\
\text{OUT}[B_1] \geqslant \text{IN}_1[B_2] & \\
\text{OUT}_i[B_2] \geqslant \text{IN}_{i,1}[B_3], & 1 \leqslant i \leqslant 10 \\
\text{OUT}_{i,j-1}[B_3] \geqslant \text{IN}_{i,j}[B_3], & 1 \leqslant i \leqslant 100, 2 \leqslant j \leqslant 10 \\
\text{OUT}_{i,10}[B_3] \geqslant \text{IN}_i[B_4], & 2 \leqslant i \leqslant 100 \\
\text{OUT}_{i-1}[B_4] \geqslant \text{IN}_i[B_2], & 1 \leqslant i \leqslant 100
\end{array}
$$

Рис. 9.61. Ограничения, которые удовлетворяются на каждой итерации вложенных циклов

Первое ограничение гласит, что выходное отображение базового блока получается путем применения передаточной функции блока ко входному отображению. Остальные ограничения говорят о том, что выходное отображение базового блока должно быть не меньше, чем входное отображение следующего блока в порядке выполнения.

Заметим, что наш итеративный алгоритм потока данных не позволяет получить приведенное решение из-за отсутствия концепции выражения значений потока данных через номер выполняемой итерации. Для поиска подобных решений, как вы увидите в следующем разделе, может применяться анализ на основе областей.

9.8.3 Символический анализ на основе областей

Можно расширить анализ на основе областей, описанный в разделе 9.7, для поиска выражений переменных на i-й итерации цикла. Символический анализ на основе областей имеет восходящий и нисходящий проходы, как и другие алгоритмы на основе областей. Восходящий проход суммирует влияние области при помощи передаточной функции, которая получает символическое отображение на входе и возвращает символическое отображение на выходе. При нисходящем проходе значения символического отображения распространяются вниз ко внутренним областям.

Различие заключается в обработке циклов. В разделе 9.7 влияние циклов подытоживается при помощи оператора замыкания. Для данного цикла с телом f его замыкание f^* определяется как бесконечный сбор всех возможных количеств применений f. Однако для поиска переменных индукции надо определить, является

ли значение переменной аффинной функцией от количества выполненных итераций. Символическое отображение должно быть параметризовано количеством выполненных итераций. Более того, если мы знаем общее количество выполненных итераций цикла, то можем использовать это число для поиска значений переменных индукции после цикла. Так, в примере 9.54 мы утверждали, что a имеет значение i после выполнения i-й итерации. Поскольку цикл состоит из 100 итераций, по его завершении значение a должно быть равно 100.

Далее мы определим примитивные операторы сбора и композиции передаточных функций для символического анализа, а затем покажем, как они используются при выполнении анализа переменных индукции на основе областей.

Сбор передаточных функций

При вычислении сбора двух функций значение переменной равно NAA, если только две функции не отображают переменную на одно и то же значение, и это значение не равно NAA. Таким образом,

$$(f_1 \wedge f_2)(m)(v) = \begin{cases} f_1(m)(v), & \text{если } f_1(m)(v) = f_2(m)(v), \\ \text{NAA} & \text{в противном случае.} \end{cases}$$

Параметризованные композиции функций

Чтобы выразить переменную как аффинную функцию от индекса цикла, требуется вычислить влияние композиции функции, взятой некоторое количество раз. Если влияние одной итерации подытоживается передаточной функцией f, то влияние выполнения i итераций для некоторого $i \geqslant 0$ обозначается как f^i. Заметим, что при $i = 0$ мы получаем тождественную функцию: $f^i = f^0 = I$.

Переменные в программе разделяются на следующие категории.

1. Если $f(m)(x) = m(x) + c$, где c — константа, то $f^i(m)(x) = m(x) + ci$ для каждого значения $i \geqslant 0$. Мы говорим, что x является *базовой переменной индукции* (basic induction variable) цикла, тело которого представлено передаточной функцией f.

2. Если $f(m)(x) = m(x)$, то $f^i(m)(x) = m(x)$ для всех $i \geqslant 0$. Переменная x остается неизменной после любого количества итераций цикла с передаточной функцией f. Мы говорим, что x является *символической константой* цикла.

3. Если $f(m)(x) = c_0 + c_1 m(x_1) + \cdots + c_n m(x_n)$, где каждая переменная x_k является либо базовой переменной индукции, либо символической константой, то при $i > 0$

$$f^i(m)(x) = c_0 + c_1 f^i(m)(x_1) + \cdots + c_n f^i(m)(x_n)$$

Мы говорим, что x также является переменной индукции, но не базовой. Заметим, что приведенная формула не применима при $i = 0$.

4. Во всех прочих случаях $f^i(m)(x) = \text{NAA}$.

Чтобы найти влияние выполнения фиксированного количества итераций, надо просто заменить i этим числом. В случае, когда количество итераций неизвестно, f^* дает значение в начале последней итерации. В этом случае инвариантными относительно цикла переменными являются только те переменные, значения которых могут быть выражены в аффинном виде:

$$f^*(m)(v) = \begin{cases} m(v), & \text{если } f(m)(v) = m(v), \\ \text{NAA} & \text{в противном случае.} \end{cases}$$

Пример 9.58. Для самого внутреннего цикла из примера 9.54 влияние выполнения $i > 0$ итераций подытоживается в $f^i_{B_3}$. Из определения f_{B_3} мы видим, что a и b являются символическими константами, c — базовой переменной индукции, поскольку на каждой итерации она увеличивается на 1. Переменная d является переменной индукции, поскольку она является аффинной функцией от символической константы b и базовой переменной индукции c. Таким образом,

$$f^i_{B_3}(m)(v) = \begin{cases} m(a), & \text{если } v = a, \\ m(b), & \text{если } v = b, \\ m(c) + i, & \text{если } v = c, \\ m(b) + m(c) + i, & \text{если } v = d. \end{cases}$$

Если мы не можем сказать, сколько выполняется итераций блока B_3, то для выражения условий в конце цикла мы должны использовать не f^i, а f^*. В нашем случае мы получим

$$f^*_{B_3}(m)(v) = \begin{cases} m(a), & \text{если } v = a, \\ m(b), & \text{если } v = b, \\ \text{NAA}, & \text{если } v = c, \\ \text{NAA}, & \text{если } v = d. \end{cases}$$

□

Алгоритм, основанный на областях

Алгоритм 9.59. Символический анализ на основе областей

Вход: приводимый граф потока G.

Выход: символические отображения $\text{IN}[B]$ для каждого блока B из G.

Метод: вносим следующие изменения в алгоритм 9.53.

1. Изменяем способ построения передаточной функции для области цикла. В исходном алгоритме мы использовали передаточную функцию $f_{R,\text{IN}[S]}$ для отображения символического отображения на входе области цикла R на символическое отображение на входе области тела S после выполнения неизвестного количества итераций. Она определена как замыкание передаточной функции, представляющей все пути, ведущие назад ко входу в цикл, как показано на рис. 9.50, *б*. Здесь же мы определяем $f_{R,i,\text{IN}[S]}$ как представляющую влияние выполнения от начала области цикла до входа в i-ю итерацию, т.е.

$$f_{R,i,\text{IN}[S]} = \left(\bigwedge_{\text{Предшественники } B \text{ заголовка } S \text{ в } R} f_{S,\text{OUT}[B]} \right)^{i-1}$$

2. Если количество итераций области известно, итог области вычисляем, заменяя i фактическим количеством итераций.

3. При нисходящем проходе для того, чтобы найти символическое отображение, связанное со входом i-й итерации цикла, вычисляем $f_{R,i,\text{IN}[B]}$.

4. В случае, когда в правой части символического отображения в области R используется входное значение переменной $m(v)$ и на входе в область $m(v) = \text{NAA}$, вводим новую ссылочную переменную t и присваивание `t = v` в начале области R и заменяем все обращения к $m(v)$ обращениями к t. Если в этом месте не ввести ссылочную переменную, то хранившееся в переменной v значение NAA проникнет во внутренние циклы. □

$$\begin{aligned}
f_{R_5,j,\text{IN}[B_3]} &= f_{B_3}^{j-1} \\
f_{R_5,j,\text{OUT}[B_3]} &= f_{B_3}^{j} \\[1em]
f_{R_6,\text{IN}[B_2]} &= I \\
f_{R_6,\text{IN}[R_5]} &= f_{B_2} \\
f_{R_6,\text{OUT}[B_4]} &= I \circ f_{R_5,10,\text{OUT}[B_3]} \circ f_{B_2} \\[1em]
f_{R_7,i,\text{IN}[R_6]} &= f_{R_6,\text{OUT}[B_4]}^{i-1} \\
f_{R_7,i,\text{OUT}[B_4]} &= f_{R_6,\text{OUT}[B_4]}^{i} \\[1em]
f_{R_8,\text{IN}[B_1]} &= I \\
f_{R_8,\text{IN}[R_7]} &= f_{B_1} \\
f_{R_8,\text{OUT}[B_4]} &= f_{R_7,100,\text{OUT}[B_4]} \circ f_{B_1}
\end{aligned}$$

Рис. 9.62. Отношения передаточных функций в восходящем проходе в примере 9.54

Пример 9.60. Покажем, как в примере 9.54 на восходящем проходе вычисляются передаточные функции программы, представленные на рис. 9.62. Область R_5 представляет собой внутренний цикл с телом B_5. Передаточная функция $f_{B_3}^{j-1}$ представляет путь от входа в область R_5 к началу j-й итерации, где $j \geqslant 1$. Путь к концу j-й итерации ($j \geqslant 1$) представляет функция $f_{B_3}^{j}$.

Область R_6 состоит из блоков B_2 и B_4 с областью цикла R_5 посредине. Передаточные функции от входов в B_2 и R_5 можно вычислить так же, как и в исходном алгоритме. Передаточная функция $f_{R_6,\text{OUT}[B_3]}$ представляет собой композицию блока B_2 и всего выполнения внутреннего цикла, поскольку передаточная функция f_{B_4} является тождественной функцией. Так как известно, что внутренний цикл выполняется 10 раз, чтобы подытожить влияние внутреннего цикла, можно заменить j десяткой. Остальные передаточные функции можно вычислить аналогично. Вычисленные передаточные функции приведены на рис. 9.63.

f	$f(m)(a)$	$f(m)(b)$	$f(m)(c)$	$f(m)(d)$
$f_{R_5,j,\text{IN}[B_3]}$	$m(a)$	$m(b)$	$m(c)+j-1$	NAA
$f_{R_5,j,\text{OUT}[B_3]}$	$m(a)$	$m(b)$	$m(c)+j$	$m(b)+m(c)+j-1$
$f_{R_6,\text{IN}[B_2]}$	$m(a)$	$m(b)$	$m(c)$	$m(d)$
$f_{R_6,\text{IN}[R_5]}$	$m(a)+1$	$10m(a)+10$	0	$m(d)$
$f_{R_6,\text{OUT}[B_4]}$	$m(a)+1$	$10m(a)+10$	10	$10m(a)+9$
$f_{R_7,i,\text{IN}[R_6]}$	$m(a)+i-1$	NAA	NAA	NAA
$f_{R_7,i,\text{OUT}[B_4]}$	$m(a)+i$	$10m(a)+10i$	10	$10m(a)+10i+9$
$f_{R_8,\text{IN}[B_1]}$	$m(a)$	$m(b)$	$m(c)$	$m(d)$
$f_{R_8,\text{IN}[R_7]}$	0	$m(b)$	$m(c)$	$m(d)$
$f_{R_8,\text{OUT}[B_4]}$	100	1000	10	1009

Рис. 9.63. Передаточные функции, вычисленные в процессе нисходящего прохода в примере 9.54

Символическое отображение на входе в программу — просто m_{NAA}. Нисходящий проход используется для вычисления символического отображения на вход в последовательные вложенные области, пока не будут найдены все символические отображения для каждого базового блока. Начнем с вычисления значений потока данных для блока B_1 в области R_8:

$$\text{IN}[B_1] = m_{\text{NAA}}$$
$$\text{OUT}[B_1] = f_{B_1}(\text{IN}[B_1])$$

Опускаясь вниз к областям R_6 и R_7, получаем

$$\text{IN}_i[B_2] = f_{R_7,i,\text{IN}[R_6]}(\text{OUT}[B_1])$$
$$\text{OUT}_i[B_2] = f_{B_2}(\text{IN}_i[B_2])$$

И наконец в области R_5 получаем

$$\text{IN}_{i,j}[B_3] = f_{R_5,j,\text{IN}[B_3]}(\text{OUT}_i[B_2])$$
$$\text{OUT}_{i,j}[B_3] = f_{B_3}(\text{IN}_{i,j}[B_3])$$

Ничего удивительного, что мы получили результаты, которые уже были показаны на рис. 9.58. □

Пример 9.54 демонстрирует простую программу, в которой каждая переменная, используемая в символическом отображении, имеет аффинное выражение. В примере 9.61 будет показано, зачем и как в алгоритме 9.59 вводятся ссылочные переменные.

Пример 9.61. Рассмотрим простой пример, показанный на рис. 9.64, *а*. Пусть f_i — передаточная функция, подытоживающая влияние выполнения j итераций внутреннего цикла. Несмотря на то что в процессе выполнения цикла переменная a изменяется, видно, что переменная b является переменной индукции, основанной на значении переменной a на входе в цикл, т.е. $f_j(m)(b) = m(a) - 1 + j$. Поскольку a присваивается входное значение, возвращаемое функцией `input()`, символическое отображение на входе во внутренний цикл отображает a на NAA. Мы вводим новую ссылочную переменную t для сохранения значения a на входе и выполняем подстановку, показанную на рис. 9.64, *б*. □

9.8.4 Упражнения к разделу 9.8

Упражнение 9.8.1. Для графа потока на рис. 9.10 (см. упражнения к разделу 9.1) укажите передаточные функции

а) для блока B_2;

б) для блока B_4;

в) для блока B_5.

Упражнение 9.8.2. Рассмотрим внутренний цикл на рис. 9.10, состоящий из блоков B_3 и B_4. Если i — количество выполнений цикла, а f — передаточная функция тела цикла (т.е. без ребра от B_4 к B_3) от входа в цикл (начала блока B_3) до выхода из B_4, то что собой представляет f^i? Помните, что f получает в качестве аргумента отображение m, которое присваивает значение каждой из переменных a, b, d и e. Мы обозначаем эти значения как $m(a)$, $m(b)$ и так далее, хотя и не знаем их точные величины.

```
1) for (i = 1; i < n; i++) {
2)     a = input();
3)     for (j = 1; j < 10; j++) {
4)         a = a - 1;
5)         b = j + a;
6)         a = a + 1;
       }
   }
```

а) Цикл с изменяющейся переменной a

```
for (i = 1; i < n; i++) {
    a = input();
    t = a;
    for (j = 1; j < 10; j++) {
        a = t - 1;
        b = t - 1 + j;
        a = t;
    }
}
```

б) Ссылочная переменная t делает b переменной индукции

Рис. 9.64. Необходимость введения ссылочных переменных

! **Упражнение 9.8.3.** Рассмотрим теперь внешний цикл на рис. 9.10, состоящий из блоков B_2, B_3, B_4 и B_5. Пусть g — передаточная функция тела цикла от входа в цикл в B_2 и до выхода из B_5. Пусть i — количество итераций внутреннего цикла из блоков B_3 и B_4 (которое нам неизвестно), а j — количество итераций внешнего цикла (также неизвестное нам). Что собой представляет g^j?

9.9 Резюме к главе 9

- *Глобальные общие подвыражения.* Поиск вычислений одного и того же выражения в двух разных базовых блоках представляет собой важный метод оптимизации. Если одно из них предшествует другому, можно сохранить результат первого вычисления и использовать сохраненное значение вместо последующих вычислений.

- *Распространение копирований.* Инструкция копирования $u = v$ присваивает значение одной переменной, v, другой переменной, u. В некоторых ситуациях можно заменить все использования u использованиями v, устраняя таким образом как присваивание, так и переменную u.

- *Перемещение кода.* Еще одним важным методом оптимизации кода является перемещение вычислений из цикла, в котором они находятся. Это изменение кода корректно лишь в том случае, если при каждой итерации цикла вычисления дают одно и то же значение.

- *Переменные индукции.* Многие циклы содержат переменные индукции, которые принимают значения из линейной последовательности при каждой очередной итерации. Некоторые из них используются только для подсчета итераций и часто могут быть устранены, тем самым снижая время выполнения итерации цикла.

- *Анализ потока данных.* Схема анализа потока данных определяет значение в каждой точке программы. С инструкциями программы сопоставлены передаточные функции, которые связывают значение до инструкции и после нее. Значения у инструкций с более чем одним предшественником определяются путем комбинирования значений предшественников с использованием оператора сбора.

- *Анализ потоков данных в базовых блоках.* Поскольку обычно распространение данных в базовых блоках достаточно простое, уравнения потока данных обычно имеют по две переменные для каждого блока, IN и OUT, которые представляют значения потоков данных соответственно в начале и конце блока. Для получения передаточной функции базового блока вычисляется композиция передаточных функций инструкций блока.

- *Достигающие определения.* Значения в структуре потока данных для достигающих определений представляют собой множества инструкций программы, которые определяют значения одной или нескольких переменных. Передаточная функция блока уничтожает определения переменных, которые переопределяются в блоке, и добавляет ("генерирует") определения переменных, которые встречаются в этом блоке. Оператор сбора представляет собой объединение, поскольку определения достигают некоторой точки, если они достигают любого из ее предшественников.

- *Активные переменные.* Еще одна важная структура потока данных вычисляет в каждой точке активные переменные (т.е. те переменные, которые используются до их переопределения). Эта структура похожа на достигающие определения, с тем отличием, что передаточная функция работает

в обратном направлении. Переменная активна в начале блока, если она либо используется до ее определения в блоке, либо активна в конце блока и не переопределяется в нем.

- *Доступные выражения.* Для поиска глобальных общих подвыражений в каждой точке программы определяются доступные выражения, которые уже вычислены к этому моменту, причем после последнего вычисления аргументы этих выражений не переопределялись. Структура потока данных в этом случае аналогична структуре достигающих определений, но оператор сбора в данном случае представляет собой пересечение множеств.

- *Абстракция задач потока данных.* Распространенные задачи потоков данных наподобие упомянутых выше, имеют общую математическую структуру. Значения потока данных являются элементами полурешетки, оператор сбора — оператором сбора полурешетки, а передаточная функция отображает элементы решетки на элементы решетки. Множество допустимых передаточных функций должно быть замкнуто относительно композиции и включать тождественную функцию.

- *Монотонные структуры.* Полурешетка имеет отношение $\leqslant$, определяемое следующим образом: $a \leqslant b$ тогда и только тогда, когда $a \wedge b = a$. Монотонные структуры обладают тем свойством, что каждая передаточная функция сохраняет отношение $\leqslant$, т.е. из $a \leqslant b$ следует $f(a) \leqslant f(b)$ для любых элементов решетки a и b и передаточной функции f.

- *Дистрибутивные структуры.* Эти структуры удовлетворяют условию $f(a \wedge b) = f(a) \wedge f(b)$ для любых элементов решетки a и b и передаточной функции f. Можно показать, что условие дистрибутивности влечет за собой монотонность.

- *Итеративное решение абстрактной структуры.* Все монотонные структуры потоков данных могут быть решены с использованием итеративного алгоритма, в котором соответствующим образом (в зависимости от структуры) инициализируются значения IN и OUT для каждого блока, а новые значения этих переменных вычисляются путем многократного применения передаточных функций и операторов сбора. Это решение всегда безопасно (предлагаемая им оптимизация не изменяет результат работы программы), но является наилучшим из возможных только в случае дистрибутивности структуры.

- *Структура распространения констант.* Наряду с дистрибутивными базовыми структурами, такими как достигающие определения, имеются интересные монотонные, но не дистрибутивные структуры. Одним из примеров

является распространение констант, использующее полурешетку, элементами которой являются отображения переменных программы на константы, а также два специальных значения, представляющих "нет информации" и "точно не константа".

- *Устранение частичной избыточности.* Многие полезные оптимизации, такие как перемещение кода и устранение глобальных подвыражений, могут быть обобщены в единую задачу, которая называется устранением частичной избыточности. Необходимые выражения, которые доступны только вдоль некоторых путей к точке, вычисляются только вдоль тех путей, где они недоступны. Корректное применение этой идеи требует решения последовательности из четырех различных задач потоков данных, а также выполнения некоторых дополнительных операций.

- *Доминаторы.* Узел в графе потока доминирует над другим узлом, если все пути к последнему проходят через первый. Истинным доминатором является доминатор, отличный от самого рассматриваемого узла. Каждый узел, кроме входного, имеет непосредственный доминатор — один из истинных доминаторов, над которым доминируют все остальные.

- *Упорядочение графов потоков в глубину.* Если выполнить поиск графа потока в глубину, начиная с его входа, то мы получим глубинное остовное дерево. Упорядочение узлов в глубину представляет собой обращенный обратный порядок обхода.

- *Классификация ребер.* При построении глубинного остовного дерева все ребра графа можно разбить на три группы: наступающие ребра (идущие от предка к истинному потомку), отступающие (идущие от потомка к предку) и поперечные (все остальные). Важное свойство заключается в том, что все поперечные ребра идут в дереве справа налево. Другим важным свойством является то, что при использовании упорядочения в глубину среди этих ребер только у отступающих заголовок меньше хвоста.

- *Обратные ребра.* Обратным называется ребро, заголовок которого доминирует над хвостом. Любое обратное ребро является отступающим, независимо от того, какое глубинное остовное дерево выбрано для его графа потока.

- *Приводимые графы потоков.* Если все отступающие ребра являются обратными, независимо от выбранного глубинного остовного дерева, то граф потока называется приводимым. Подавляющее большинство графов потоков приводимо; точно приводимы графы потоков, инструкциями управления

потоком которых являются только обычные циклы и инструкции ветвления.

- *Естественные циклы*. Естественный цикл представляет собой множество узлов с заглавным узлом, который доминирует над всеми узлами множества, содержащее как минимум одно обратное ребро, входящее в этот узел. Для заданного обратного ребра можно построить его естественный цикл, беря заголовок ребра плюс все узлы, которые могут достичь хвоста ребра, не проходя через заголовок. Два естественных цикла с разными заголовками либо не пересекаются, либо один из них полностью содержится в другом; этот факт позволяет нам говорить об иерархии вложенных циклов, если под "циклами" подразумеваются естественные циклы.

- *Упорядочение в глубину делает итеративный алгоритм более эффективным*. Итеративный алгоритм требует нескольких проходов, достаточных для распространения информации вдоль ациклических путей. Если посещать узлы в порядке в глубину, любая структура потока данных, распространяющая информацию в прямом направлении, например достигающие определения, будет сходиться не более чем за количество итераций, равное увеличенному на 2 наибольшему количеству отступающих ребер на любом ациклическом пути. То же самое справедливо и для структур с распространением информации в обратном направлении (например, для активных переменных), если посещать узлы в порядке, обратном порядку в глубину (в порядке обратного обхода).

- *Области*. Области представляют собой множества узлов и ребер с заголовком h, доминирующим над всеми узлами в области. Предшественники любого узла области, отличного от h, также должны находиться в области. Все ребра области проходят между узлами области, за возможным исключением некоторых (или всех) ребер, входящих в заголовок.

- *Области и приводимые графы потоков*. Приводимые графы потоков могут давать в результате разбора иерархию областей. Эти области являются либо областями циклов, которые включают ребра, ведущие в заголовок, либо области тел, которые не содержат ребер, ведущих в заголовок.

- *Анализ потока данных на основе областей*. Альтернативой итеративному подходу к анализу потока данных является работа в восходящем и нисходящем направлениях с иерархией областей, при которой вычисляются передаточные функции от заголовка каждой области к каждому узлу этой области.

- *Обнаружение индуктивных переменных на основе областей.* Важным применением анализа на основе областей является структура потока данных, которая пытается вычислить формулы для каждой переменной в области цикла, значения которой представляют собой аффинную функцию от количества выполненных итераций цикла.

9.10 Список литературы к главе 9

Два ранних компилятора, активно использовавших оптимизацию, – это Alpha [16] и Fortran H [16]. Фундаментальным исследованием по методам оптимизации циклов (например, перемещению кода) считается работа [1], хотя ранние версии некоторых из идей появились в [8]. Многие идеи по оптимизации кода можно найти в [4].

Первое описание итеративного алгоритма для анализа потока данных приводится в неопубликованном отчете Высоцкого (Vyssotsky) и Вегнера (Wegner) [20]. Научное изучение анализа потоков данных началось с пары статей Аллена (Allen) [2] и Кука (Cocke) [3].

Теоретическая абстракция решетки, описанная в нашей книге, основана на работе Килдалла (Kildall) [13]. В его работе структуры считаются дистрибутивными, но имеется множество структур, не являющихся таковыми. После того как было обнаружено достаточное количество структур, не являющихся дистрибутивными, в модель в работах [5 и 11] было введено условие монотонности.

Впервые устранение частичной избыточности рассмотрено в [17]. Алгоритм отложенного перемещения кода, описанный в данной главе, основан на работе [14].

Доминаторы впервые использованы в компиляторе, описанном в [13]. Однако впервые идея появилась в [18].

Понятие приводимых графов потоков происходит из [2]. Структура этих графов потоков, представленная в данной главе, основана на [9 и 10]. В работах [12 и 15] впервые связаны приводимость графов потоков с распространенными вложенными структурами управления потоками, что поясняет широкую распространенность этого класса графов потоков.

Определение приводимости с помощью приведения $T_1 - T_2$, используемое в анализе на основе областей, описано в [19]. Подход на основе областей впервые использован в компиляторе, описанном в [21].

Статическое промежуточное представление с единственным присваиванием содержит как поток данных, так и поток управления. Такое представление упрощает реализацию многих оптимизирующих представлений в распространенных структурах [6].

1. Allen, F. E., “Program optimization”, *Annual Review in Automatic Programming* **5** (1969), pp. 239–307.

2. Allen, F. E., “Control flow analysis”, *ACM Sigplan Notices* **5**:7 (1970), pp. 1–19.

3. Cocke, J., “Global common subexpression elimination”, *ACM SIGPLAN Notices* **5**:7 (1970), pp. 20–24.

4. Cocke, J. and J. T. Schwartz, *Programming Languages and Their Compilers: Preliminary Notes*, Courant Institute of Mathematical Sciences, New York Univ., New York, 1970.

5. Cousot, P. and R. Cousot, “Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints”, *Fourth ACM Symposium on Principles of Programming Languages* (1977), pp. 238–252.

6. Cytron, R., J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck, “Efficiently computing static single assignment form and the control dependence graph”, *ACM Transactions on Programming Languages and Systems* **13**:4 (1991), pp. 451–490.

7. Ershov, A. P., “Alpha — an automatic programming system of high efficiency”, *J. ACM* **13**:1 (1966), pp. 17–24.

8. Gear, C. W., “High speed compilation of efficient object code”, *Comm. ACM* **8**:8 (1965), pp. 483–488.

9. Hecht, M. S. and J. D. Ullman, “Flow graph reducibility”, *SIAM J. Computing* **1** (1972), pp. 188–202.

10. Hecht, M. S. and J. D. Ullman, “Characterizations of reducible flow graphs”, *J. ACM* **21** (1974), pp. 367–375.

11. Kam, J. B. and J. D. Ullman, “Monotone data flow analysis frameworks”, *Acta Informatica* **7**:3 (1977), pp. 305–318.

12. Kasami, T., W. W. Peterson, and N. Tokura, “On the capabilities of while, repeat, and exit statements”, *Comm. ACM* **16**:8 (1973), pp. 503–512.

13. Kildall, G., “A unified approach to global program optimization”, *ACM Symposium on Principles of Programming Languages* (1973), pp. 194–206.

14. Knoop, J., “Lazy code motion”, *Proc. ACM SIGPLAN 1992 conference on Programming Language Design and Implementation*, pp. 224–234.

15. Kosaraju, S. R., "Analysis of structured programs", *J. Computer and System Sciences* **9**:3 (1974), pp. 232–255.

16. Lowry, E. S. and C. W. Medlock, "Object code optimization", *Comm. ACM* **12**:1 (1969), pp. 13–22.

17. Morel, E. and C. Renvoise, "Global optimization by suppression of partial redundancies", *Comm. ACM* **22** (1979), pp. 96–103.

18. Prosser, R. T., "Application of boolean matrices to the analysis of flow diagrams", *AFIPS Eastern Joint Computer Conference* (1959), Spartan Books, Baltimore MD, pp. 133–138.

19. Ullman, J. D., "Fast algorithms for the elimination of common subexpressions", *Acta Informatica* **2** (1973), pp. 191–213.

20. Vyssotsky, V. and P. Wegner, "A graph theoretical Fortran source language analyzer", unpublished technical report, Bell Laboratories, Murray Hill NJ, 1963.

21. Wulf, W. A., R. K. Johnson, C. B. Weinstock, S. O. Hobbs, and C. M. Geschke, *The Design of an Optimizing Compiler*, Elsevier, New York, 1975.

Глава 10

Параллелизм на уровне команд

Все современные высокопроизводительные процессоры могут выполнять за один такт несколько операций. Вопрос "на засыпку": насколько быстро может выполняться программа на процессоре с параллелизмом на уровне команд? Ответ зависит от нескольких факторов.

1. Потенциальный параллелизм программы.

2. Доступный параллелизм процессора.

3. Наша способность выделить параллелизм в исходной последовательной программе.

4. Наша способность найти наилучшее планирование параллельного выполнения при заданных ограничениях планирования.

Если все операции в программе сильно зависят одна от другой, то никакая аппаратная параллельность или методы распараллеливания не ускорят выполнение программы. Была выполнена масса исследовательских работ, посвященных лучшему пониманию, что может дать параллельность и где находятся границы ее применимости. Типичное не численное приложение содержит массу неотъемлемых зависимостей. Например, в таких программах много ветвлений, зависящих от данных, и эти ветвления затрудняют даже предсказание того, какие именно команды будут выполняться, не говоря уже о том, чтобы решать, какие команды могут выполняться параллельно. Следовательно, работа в этой области должна быть сосредоточена не столько на методах планирования, сколько на ослаблении ограничений планирования, включая добавление новых архитектурных возможностей.

Численные приложения, такие как научные расчеты и обработка сигналов, обычно допускают большую степень распараллеливания. Эти приложения работают с большими агрегированными структурами данных; зачастую операции над различными элементами структуры не зависят одна от другой и могут выполняться параллельно. Дополнительные аппаратные ресурсы могут использовать преимущества этой параллельности, так что они часто применяются в высокопроизводительных машинах общего назначения и процессорах для цифровой обработки сигналов. Обычно в таких программах используются простые управляющие

структуры и регулярные шаблоны обращения к данным, так что для выделения параллельности в таких программах разработаны различные статические методы. Планирование выполнения кода для таких приложений представляет собой интересную и важную задачу, в которой большое количество независимых операций отображается на большое количество ресурсов.

Как выделение параллелизма, так и планирование параллельного выполнения может быть выполнено либо статически программным обеспечением, либо динамически — аппаратным. В действительности программное планирование может помочь даже при использовании машин с аппаратным планированием. Эта глава начинается с рассмотрения фундаментальных вопросов использования параллелизма на уровне команд, которые одинаковы как при аппаратном, так и при программном управлении параллелизмом. Затем мы рассмотрим основы анализа зависимости данных, необходимого для выделения параллелизма. Эти виды анализа полезны не только для параллелизма на уровне команд, но и, как вы увидите в главе 11, для других оптимизаций.

Наконец, мы представим основные идеи планирования кода. Мы опишем методы планирования базовых блоков, метод работы с управлением потоком, сильно зависящим от данных (ситуация, распространенная в программах общего назначения), и метод программной конвейерной обработки, используемый, в первую очередь, для планирования численных программ.

10.1 Архитектуры процессоров

Когда мы говорим о параллелизме на уровне команд, то обычно представляем процессор, выполняющий несколько команд за один такт. В действительности возможна машина, выполняющая за один такт только одну операцию, но при этом достигающая параллелизма на уровне команд благодаря использованию концепции *конвейерной обработки* (pipelining). Далее мы сначала рассмотрим конвейерную обработку, а затем — многоадресные команды.

10.1.1 Конвейерная обработка команд и задержки ветвления

Практически каждый процессор — будь то в высокопроизводительной супермашине или в обычном настольном компьютере — использует *конвейер (обработки) команд* (instruction pipeline). При использовании конвейера команд выборка новой команды может выполняться в то время, когда предыдущая команда все еще находится в конвейере. На рис. 10.1 показан простой пятиэтапный конвейер команд, который сначала выполняет выборку команды (IF), затем — ее декодирование (ID), выполняет команду (EX), обращается к памяти (MEM) и записывает результат (WB). На диаграмме показано, как одновременно могут выполняться

команды i, $i+1$, $i+2$, $i+3$ и $i+4$. Каждая строка соответствует такту системных часов, а каждый столбец на рисунке определяет, когда выполняются этапы каждой из команд.

	i	$i+1$	$i+2$	$i+3$	$i+4$
1.	IF				
2.	ID	IF			
3.	EX	ID	IF		
4.	MEM	EX	ID	IF	
5.	WB	MEM	EX	ID	IF
6.		WB	MEM	EX	ID
7.			WB	MEM	EX
8.				WB	MEM
9.					WB

Рис. 10.1. Пять последовательных команд в пятиэтапном конвейере команд

Если результат команды доступен в тот момент, когда последующая команда нуждается в данных, процессор может выполнять по команде каждый такт. Выполнение команд ветвления особенно проблематично, поскольку, пока они не будут выбраны, декодированы и выполнены, процессору неизвестно, какая команда будет выполняться следующей. Многие процессоры умозрительно выбирают и декодируют непосредственно следующие друг за другом команды, пока не встречается команда ветвления. Если же в потоке обнаруживается команда ветвления, конвейер опустошается, и выполняется выборка целевой команды ветвления. Таким образом, ветвления вносят задержку в выборку целевой команды и приводят к "иканию" конвейера. Мощные интеллектуальные процессоры используют аппаратное предсказание результата ветвления на основе истории выполнения и выполняют упреждающую выборку из предсказанной целевой ячейки памяти. Тем не менее при неверном предсказании задержки ветвления наблюдаются и в таких процессорах.

10.1.2 Конвейерное выполнение

Некоторые команды требуют для выполнения нескольких тактов. Распространенным примером может служить команда загрузки из памяти. Даже если память, к которой выполняется обращение, находится в кэше, обычно требуется несколько тактов для того, чтобы получить данные из кэша. Мы говорим, что выполнение команды конвейерное (pipelined), если может выполняться следующая команда, которая не зависит от результата предыдущей. Таким образом, даже если процессор может выполнять только одну операцию за такт, одновременно на разных стадиях выполнения могут находиться несколько команд. Если конвейер содер-

жит n этапов, то потенциально "в полете" могут находиться одновременно n команд. Заметим, что не все команды полностью конвейеризуемы. В то время как сложения и умножения чисел с плавающей точкой зачастую полностью конвейеризуемы, более сложное и менее часто используемое деление чисел с плавающей точкой таковым зачастую не является.

Большинство процессоров общего назначения динамически определяют зависимости между последовательными командами и автоматически приостанавливают выполнение команды в случае недоступности ее операндов. Некоторые процессоры, в особенности встраиваемые в портативные устройства, оставляют проверку зависимостей программному обеспечению по соображениям простоты и снижения энергопотребления процессора. В этом случае компилятор отвечает за вставку в код команд "нет операции" там, где необходимо гарантировать доступность результатов предыдущей операции.

10.1.3 Многоадресные команды

Выполняя несколько операций за такт, процессор может одновременно работать с еще бо́льшим количеством команд. Наибольшее количество одновременно выполняемых операций может быть вычислено путем умножения количества предварительно обрабатываемых команд на среднее количество этапов конвейера выполнения.

Подобно конвейеру, параллельность на многокомандных машинах может управляться как аппаратно, так и программно. Машины с программной обработкой параллельности известны под аббревиатурой *VLIW* (Very Long Instruction Word — архитектура процессора с командными словами очень большой длины); машины же с аппаратной обработкой известны под названием *суперскалярных*. VLIW-машины, как следует из их названия, имеют команды большого размера, которые кодируют операции, выполняющиеся за один такт. Компилятор решает, какие операции будут выполняться параллельно, и явно указывает эту информацию в машинном коде. Суперскалярные же машины автоматически обнаруживают зависимости между командами и выполняют команды, как только их операнды становятся доступными. Некоторые процессоры включают как VLIW, так и суперскалярную функциональность.

Простые аппаратные планировщики выполняют команды в том порядке, в котором выполняется их выборка. Если планировщик обнаруживает зависимую команду, она и все последующие должны дождаться, пока зависимости не будут разрешены (то есть пока не станут доступны все требующиеся результаты выполнения предыдущих команд). Очевидно, что таким машинам выгодна работа со статическим планировщиком, который размещает независимые операции одна за другой в порядке выполнения.

Более сложные планировщики могут выполнять команды “не по порядку”. Операции независимо приостанавливаются и не выполняются до тех пор, пока не будут получены все значения, от которых эти операции зависят. Но статические планировщики могут помочь в работе даже столь интеллектуальным планировщикам, поскольку аппаратные планировщики ограничены объемом памяти, в котором могут храниться отложенные операции. Статический же планировщик может разместить независимые операции поближе одна к другой с тем, чтобы более эффективно использовать аппаратные возможности процессора. Что еще более важно, независимо от степени интеллектуальности аппаратного планировщика, он не в состоянии работать с командами, которые еще не были выбраны. Когда процессор выполняет неожиданное ветвление, параллельное выполнение доступно только для вновь выбираемых команд. Компилятор может повысить производительность динамического планировщика, обеспечивая возможность параллельного выполнения вновь выбираемых команд.

10.2 Ограничения планирования кода

Планирование кода (code scheduling) представляет собой оптимизацию программы, применяемую к машинному коду, произведенному генератором кода. На планирование кода накладываются три типа ограничений.

1. *Ограничения управления.* Все операции, выполнимые в исходной программе, должны выполняться и в оптимизированной.

2. *Ограничения данных.* Операции в оптимизированной программе должны выдать те же результаты, что и соответствующие операции в исходной программе.

3. *Ограничения ресурсов.* Планирование не должно требовать чрезмерного количества ресурсов машины.

Эти ограничения гарантируют, что оптимизированная программа даст те же результаты, что и исходная. Однако, поскольку планирование кода изменяет порядок выполнения операций, состояние памяти в произвольной точке может не соответствовать состоянию памяти при последовательном выполнении программы. Это вызывает проблемы, например, при генерации исключения или при вставленной пользовательской точке прерывания, так что оптимизированные программы сложнее отлаживать. Заметим, что данная проблема относится не только к планированию кода, но и к оптимизации вообще, включая, например, устранение частичной избыточности (см. раздел 9.5) или распределение регистров (см. раздел 8.8).

10.2.1 Зависимость через данные

Легко видеть, что если мы изменяем порядок выполнения двух операций, которые не затрагивают ни одной общей переменной, то это никак не влияет на результат их выполнения. Более того, если даже обе команды считывают значение одной и той же переменной, мы все равно можем переставить их местами. Только если операция записывает переменную, которую считывает или которую перезаписывает другая операция, то изменение порядка этих операций может привести к изменению результата. О таких парах операций говорят, что они *зависимы через данные* (data dependent) и их относительный порядок выполнения должен быть сохранен[1]. Имеется три вида зависимости через данные.

1. *Истинная зависимость*: чтение после записи. Если за записью следует чтение из той же ячейки памяти, то чтение зависит от записанного значения; такая зависимость известна как истинная зависимость.

2. *Антизависимость*: запись после чтения. Если за чтением следует запись в ту же ячейку памяти, то мы говорим о наличии антизависимости от чтения к записи. Запись как таковая не зависит от чтения, но если запись выполняется до чтения, то последнее даст неверное значение. Антизависимость является побочным продуктом императивного программирования, когда одна и та же ячейка памяти используется для хранения разных значений. Это не "истинная" зависимость и потенциально она может быть устранена путем хранения значений в разных ячейках памяти.

3. *Зависимость через выход*: запись после записи. Две записи в одну и ту же ячейку памяти приводят к зависимости через выход. При нарушении порядка записи после выполнения обеих операций в ячейке будет записано неверное значение.

Антизависимость и зависимость через выход объединяются в одну категорию зависимостей, *связанных с хранением* (storage-related dependences). Это не "истинные" зависимости, которые могут быть устранены путем использования разных ячеек памяти для разных значений. Заметим, что зависимость через данные применима к обращениям как к памяти, так и к регистрам.

10.2.2 Поиск зависимостей среди обращений к памяти

Чтобы проверить, зависимы ли два обращения к памяти, надо только выяснить, обращаются ли они к одной и той же ячейке памяти; знать точное расположение

[1] Для краткости мы часто будем говорить просто о *зависимости данных*, где из контекста очевидно, что речь идет о зависимости операций через данные. — *Прим. пер.*

этой ячейки не требуется. Например, можно с уверенностью сказать, что обращения `*p` и `*(p+4)` не могут обращаться к одной и той же ячейке памяти, несмотря на то, что мы не знаем, куда именно указывает p. В общем случае во время компиляции проверка зависимости не разрешима. Компилятор обязан предполагать, что операции могут обращаться к одним и тем же ячейкам памяти, если он не в состоянии доказать обратное.

Пример 10.1. В случае кода

```
1)  a = 1
2)  *p = 2
3)  x = a
```

если только компилятор не может гарантировать, что p не может указывать на a, он должен считать, что данные операции должны выполняться последовательно. Здесь имеется зависимость через выход между инструкциями 1 и 2 и две истинные зависимости между инструкциями 1 и 2 и инструкцией 3. □

Анализ зависимости данных очень чувствителен к языку программирования, использованному при написании программы. Для небезопасных с точки зрения типов языков наподобие C и C++, в которых указатель может быть преобразован в указатель на объект любого типа, для доказательства независимости между произвольной парой обращений к памяти посредством указателей необходим сложный анализ. Даже к локальным или глобальным скалярным переменным может иметься косвенное обращение, если только мы не сможем доказать, что их адреса не были сохранены где-либо какой-то из инструкций программы. В языках программирования, безопасных с точки зрения типов, наподобие Java объекты различных типов с необходимостью отличаются друг от друга. Аналогично локальные примитивные переменные в стеке не могут оказаться псевдонимами при обращении через другие имена.

Корректный поиск зависимостей данных требует проведения большого количества разных видов анализа. Мы остановимся на основных вопросах, которые должны быть разрешены, если компилятор ищет все зависимости, существующие в программе, и на использовании этой информации при планировании кода. В следующих главах будет рассмотрено выполнение этих анализов.

Анализ зависимости данных для массива

Зависимость данных для массива представляет собой задачу устранения неоднозначностей между значениями индексов при обращениях к элементам массива. Например, цикл

```
for(i=0; i<n; ++i)
    A[2*i] = A[2*i+1];
```

копирует нечетные элементы массива A в четные элементы, предшествующие им. Поскольку все считываемые и записываемые ячейки памяти отличаются друг от друга, между обращениями к памяти нет зависимостей, и все итерации цикла могут быть выполнены параллельно. Анализ зависимости данных для массива, о котором зачастую просто говорят как об *анализе зависимости данных* (data-dependence analysis), очень важен для оптимизации численных приложений. Эта тема будет рассмотрена в разделе 11.6.

Анализ псевдонимов указателей

Мы говорим, что два указателя являются *псевдонимами* (aliased), если они могут обращаться к одному и тому же объекту. Анализ псевдонимов указателей сложен в силу многочисленности потенциальных псевдонимов в программе, причем по ходу программы каждый из них может указывать на неограниченное количество динамических объектов. Для достижения хоть какой-то точности анализ должен применяться ко всем функциям программы. Этот вопрос будет рассматриваться начиная с раздела 12.4.

Межпроцедурный анализ

Для языков, передающих параметры по ссылке, межпроцедурный анализ используется, чтобы выяснить, не передана ли одна и та же переменная в процедуру в качестве двух (или более) разных параметров. Такие псевдонимы могут создать зависимости между кажущимися различными параметрами. Аналогично в качестве параметров могут использоваться глобальные переменные, что может создать зависимости между обращениями к параметрам и обращениями к глобальным переменным. Для выявления таких псевдонимов требуется межпроцедурный анализ, рассматривающийся в главе 12.

10.2.3 Компромиссы между использованием регистров и параллелизмом

В этой главе мы будем полагать, что машинно-независимое промежуточное представление исходной программы использует неограниченное количество *псевдорегистров* для представления переменных, которые могут быть размещены в регистрах. Эти переменные включают скалярные переменные исходной программы, обращение к которым невозможно ни по какому иному имени, а также временные переменные, генерируемые компилятором для хранения промежуточных результатов при вычислении выражений. В отличие от ячеек памяти регистры имеют уникальные имена. Таким образом, для регистров легко могут быть сгенерированы точные ограничения, связанные с зависимостями данных.

Неограниченное количество псевдорегистров, использованное в промежуточном представлении, в конечном счете должно быть отображено на небольшое

Переименование аппаратных регистров

Параллелизм на уровне команд был впервые использован в компьютерной архитектуре как средство повышения скорости выполнения обычного последовательного кода. Компиляторам в то время не было ничего известно о параллелизме, и они разрабатывались с учетом требования оптимизации использования регистров. Они преднамеренно переупорядочивали команды так, чтобы минимизировать количество используемых регистров, а в результате они одновременно минимизировали и возможности параллельного выполнения команд. В примере 10.3 проиллюстрировано, как минимизация использования регистров при вычислении деревьев выражения ограничивает параллелизм.

В результате в последовательном коде оставалось так мало возможностей для параллельных вычислений, что разработчики процессоров разработали концепцию *аппаратного переименования процессоров* для отмены результатов оптимизации компилятором использования регистров. Аппаратное переименование регистров динамически изменяет назначение регистров в процессе выполнения программы. Оно интерпретирует машинный код, сохраняет значения, предназначенные для одного и того же регистра, в различных внутренних регистрах, и изменяет все их использования так, чтобы обращения к соответствующим регистрам выполнялись корректно.

Поскольку, в первую очередь, ограничения, связанные с зависимостями регистров, возникают благодаря работе компилятора, они могут быть устранены путем использования алгоритма распределения регистров, который учитывает параллелизм на уровне команд. Аппаратное переименование регистров остается полезным в случае, когда набор машинных команд может работать только с небольшим количеством регистров. Эта возможность позволяет реализовать архитектуру с динамическим отображением небольшого количества архитектурных регистров в коде на существенно большее количество внутренних регистров.

количество физических регистров, доступное на целевой машине. Отображение нескольких псевдорегистров на один и тот же физический регистр имеет нежелательный побочный эффект создания искусственных зависимостей, ограничивающих параллелизм на уровне команд. И обратно, параллельное выполнение команд приводит к требованию большего количества памяти для хранения параллельно вычисляемых значений. Таким образом, цель минимизировать количество используемых регистров конфликтует с целью максимизировать параллелизм на

уровне команд. В приведенных далее примерах 10.2 и 10.3 проиллюстрирован поиск компромиссов между использованием памяти и параллелизмом.

Пример 10.2. Приведенный ниже код копирует значения переменных в ячейках памяти `a` и `c` в переменные в ячейках памяти `b` и `d` соответственно с использованием псевдорегистров `t1` и `t2`.

```
LD t1, a     // t1 = a
ST b, t1     // b = t1
LD t2, c     // t2 = c
ST d, t2     // d = t2
```

Если известно, что все ячейки памяти, к которым выполняется обращение, отличаются одна от другой, то операции копирования могут быть выполнены параллельно. Однако если переменным `t1` и `t2` назначен один и тот же регистр для минимизации количества используемых регистров, то операции копирования обязательно должны выполняться последовательно. □

Пример 10.3. Традиционные методы распределения регистров преследуют цель минимизации количества регистров, используемых при выполнении вычислений. Рассмотрим показанное в виде синтаксического дерева на рис. 10.2 выражение

```
(a + b) + c + (d + e)
```

Вычисление можно выполнить с использованием трех регистров, как продемонстрировано на рис. 10.3.

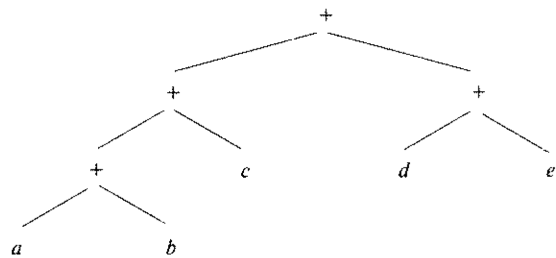


Рис. 10.2. Дерево выражения из примера 10.3

Повторное использование регистров требует последовательного вычисления. Единственные операции, которые могут быть выполнены параллельно, — это загрузки значений из ячеек `a` и `b` и загрузки значений из ячеек `d` и `e`. Таким образом, всего для полного вычисления выражения с максимальным использованием параллельности требуется 7 шагов.

Если же использовать различные регистры для каждой промежуточной суммы, то выражение можно вычислить за 4 шага, что равно высоте дерева выражения на рис. 10.2. Распараллеленное вычисление показано на рис. 10.4. □

```
LD  r1, a          // r1 = a
LD  r2, b          // r2 = b
ADD r1, r1, r2     // r1 = r1+r2
LD  r2, c          // r2 = c
ADD r1, r1, r2     // r1 = r1+r2
LD  r2, d          // r2 = d
LD  r3, e          // r3 = e
ADD r2, r2, r3     // r2 = r2+r3
ADD r1, r1, r2     // r1 = r1+r2
```

Рис. 10.3. Машинный код для выражения на рис. 10.2

r1 = a	r2 = b	r3 = c	r4 = d	r5 = e
r6 = r1+r2	r7 = r4+r5			
r8 = r6+r3				
r9 = r8+r7				

Рис. 10.4. Параллельное вычисление выражения на рис. 10.2

10.2.4 Упорядочение фаз распределения регистров и планирования кода

Если регистры распределяются до планирования, то получающийся код проявляет склонность к множественным зависимостям, ограничивающим планирование кода. С другой стороны, планирование кода до распределения регистров может привести к тому, что потребуется так много регистров, что *сброс* регистров (register spilling), т.е. сохранение их содержимого в памяти, чтобы регистры могли использоваться для других целей, может свести на нет все преимущества параллельного параллелизма на уровне команд. Должен ли компилятор распределять регистры до планирования кода? Должен ли он делать это после? Или надо решать обе задачи одновременно?

Чтобы ответить на поставленные вопросы, следует рассмотреть характеристики компилируемой программы. Многие нечисловые приложения не допускают сильного распараллеливания. Достаточно выделить для хранения временных результатов при вычислении выражений небольшое количество регистров. Можно начать с применения алгоритма раскрашивания из раздела 8.8.4 для распределения регистров для всех переменных, не являющихся временными, затем спланировать код и распределить регистры для временных переменных.

Такой подход не работает в случае численных приложений, в которых в наличии имеется много очень больших выражений. Здесь можно применить иерархический подход, при котором код оптимизируется изнутри наружу, начиная с наиболее внутренних циклов. В начале команды планируются в предположении, что

каждому псевдорегистру будет выделен собственный физический регистр. Распределение регистров выполняется после планирования кода; при необходимости осуществляется добавление кода для сброса регистров и планирование кода выполняется заново. Этот процесс повторяется для кода внешних циклов. Когда несколько внутренних циклов рассматриваются вместе в общем внешнем цикле, одной и той же переменной могут быть назначены разные регистры. Назначение регистров в таком случае может быть изменено, чтобы избежать копирований из одного регистра в другой. В разделе 10.5 мы обсудим взаимодействие распределения регистров и планирования в контексте конкретного алгоритма планирования.

10.2.5 Зависимость от управления

Планирование операций внутри базового блока относительно простое, поскольку все команды гарантированно выполняются, если поток управления достигает начала блока. Команды базового блока могут быть произвольным образом переупорядочены при условии удовлетворения всем зависимостям данных. К сожалению, базовые блоки, в особенности в нечисленных программах, обычно очень малы — в среднем всего около пяти команд в базовом блоке. Кроме того, операции в одном и том же блоке зачастую сильно связаны и, таким образом, допускают параллелизм только в малой степени. Соответственно, особую роль приобретает параллелизм, пересекающий границы базовых блоков.

Оптимизированная программа должна выполнять все операции исходной программы. Она может выполнять больше команд, чем имеется в исходной программе, при условии, что эти дополнительные команды не изменяют функциональность программы. Как выполнение дополнительных команд может ускорить выполнение программы? Если мы знаем, что некоторая команда, скорее всего, будет выполняться, и при этом имеется простаивающий ресурс, позволяющий "бесплатно" выполнить эту операцию, то команду можно выполнить с *опережением*, "на всякий случай" (speculatively). Если такое опережение оказалось оправданным, программа будет выполняться быстрее.

Команда i_1 называется зависящей по управлению (control-dependent) от команды i_2, если выход команды i_2 определяет, будет ли выполняться команда i_1. Понятие зависимости от управления соответствует концепции вложенных уровней в блочной структуре программы. В частности, в инструкции if-else

```
if (c) s1; else s2;
```

`s1` и `s2` зависят от `c` по управлению. Аналогично в инструкции while

```
while (c) s;
```

тело `s` зависит по управлению от `c`.

Пример 10.4. В фрагменте кода

```
if (a > t)
    b = a * a;
d = a + c;
```

инструкции `b = a*a` и `d = a+c` не зависят по данным от других частей фрагмента. Инструкция `b = a*a` зависит от сравнения $a > t$ по управлению. Инструкция же `d = a+c` не зависит от сравнения и может быть выполнена в любой момент времени. В предположении, что умножение $a * a$ не дает никаких побочных эффектов, эта инструкция может быть выполнена с опережением, лишь бы переменная b при этом записывалась только после того, как будет выяснено, что a больше t. □

10.2.6 Поддержка опережающего выполнения

Загрузка памяти — один из типов инструкций, которые могут получить существенные выгоды от опережающего выполнения. Загрузка памяти — очень распространенная операция с относительно долгими задержками. Адреса, по которым выполняется чтение из памяти, обычно известны заранее, а результат может быть сохранен в новой временной переменной без уничтожения значения в другой переменной. К сожалению, загрузка памяти может вызвать генерацию исключения, если адрес считывания некорректен; поэтому опережающая загрузка может привести к аварийному останову корректной программы. Кроме того, неверно предсказанные загрузки памяти могут привести к промахам кэша и ошибкам отсутствия страницы, что может существенно замедлить работу.

Пример 10.5. В фрагменте

```
if (p != NULL)
    q = *p;
```

опережающее разыменование p приведет аварийному останову совершенно корректной программы, если значение p — `NULL`. □

Многие высокопроизводительные процессоры предоставляют специальные возможности для поддержки опережающей загрузки памяти. Мы упомянем только важнейшие из них.

Упреждающая выборка

Упреждающая выборка (prefetching) команды была разработана для переноса данных из памяти в кэш перед их использованием. Команда упреждающей выборки указывает процессору, что в ближайшее время ожидается использование некоторого слова памяти. Если указанный адрес некорректен или если обращение по этому адресу приводит к ошибке отсутствия страницы, процессор просто

игнорирует эту операцию. В противном случае процессор переносит данные из памяти в кэш, если, конечно, они уже не находятся там.

Биты незавершенного обращения к памяти

Еще одна архитектурная особенность, именуемая *poison bits*[2], разработана для упреждающей загрузки памяти в регистр. Каждый регистр машины снабжен дополнительным битом незавершенности обращения к памяти. Если выполнено обращение к памяти по некорректному адресу или соответствующей страницы нет в памяти, процессор не генерирует исключение, а просто устанавливает бит незавершенного обращения к памяти целевого регистра. Исключение генерируется только при использовании содержимого регистра с установленным битом незавершенного обращения к памяти.

Предикатное выполнение

Из-за высокой стоимости операций ветвления и еще более высокой стоимости ошибки в предсказании ветвления (см. раздел 10.1) для снижения количества ветвлений в программе были разработаны *предикатные команды* (predicated instructions). Предикатные команды похожи на обычные, но содержат дополнительный предикатный операнд; команда выполняется, только если предикат является истинным.

В качестве примера приведем команду `CMOVZ R2,R3,R1`, семантика которой заключается в перемещении содержимого регистра R3 в регистр R2 только в том случае, когда регистр R1 содержит нулевое значение. Код наподобие

```
if (a == 0)
    b = c + d
```

в предположении, что переменные a, b, c и d размещены в регистрах R1, R2, R4 и R5 соответственно, может быть реализован машинными командами

```
ADD   R3,R4,R5
CMOVZ R2,R3,R1
```

Такое преобразование заменяет последовательность команд с зависимостями по управлению командами с зависимостями только по данным. Затем такие команды могут быть объединены с соседними базовыми блоками в базовый блок большего размера. Что еще более важно, в случае такого кода процессор не может ошибиться в своих предсказаниях; таким образом, гарантируется эффективная работа конвейера команд.

Предикатное выполнение имеет свою стоимость. Предикатные команды выбираются и декодируются, даже если в конечном итоге они не будут выполнены.

[2]Дословно — "ядовитый бит". — *Прим. пер.*

Машины с динамическим планированием

Набор команд машины со статическим планированием явным образом определяет, какие именно команды будут выполняться параллельно. Вспомним, однако, из раздела 10.1.2, что некоторые архитектуры позволяют принимать решение о том, какие команды будут выполняться параллельно, в процессе работы программы. При динамическом планировании один и тот же машинный код может выполняться на различных машинах одного семейства (т.е. на машинах, реализующих один и тот же набор команд) с разной степенью параллелизма. По сути, одним из главных преимуществ машин с динамическим планированием является совместимость по машинному коду.

Статические планировщики, программно реализованные в компиляторе, могут работать в паре с динамическими планировщиками (реализованными аппаратно), повышая эффективность использования машинных ресурсов. При построении статического планировщика для машины с динамическим планированием можно использовать практически тот же алгоритм, что и для машин со статическим планированием, с тем отличием, что при этом не требуется явная генерация команд "нет операции". Этот вопрос будет рассматриваться в разделе 10.4.7.

Статические планировщики должны резервировать все ресурсы, необходимые для выполнения этих команд, и гарантировать удовлетворение всех потенциальных зависимостей данных. Предикатное выполнение не должно применяться слишком "агрессивно", если только машина не имеет существенно больше ресурсов, чем могло бы использоваться в противном случае.

10.2.7 Базовая модель машины

Многие машины можно представить с использованием следующей простой модели. Машина $M = \langle R, T \rangle$ состоит из

1. множества типов операций T, таких как загрузки, сохранения, арифметические операции и т.д.;
2. вектора $R = [r_1, r_2, \ldots]$, представляющего аппаратные ресурсы, где r_i — количество доступных единиц i-го ресурса. Примеры типичных типов ресурсов включают модули обращения к памяти (далее для краткости будем обозначать этот ресурс как МОП), АЛУ, модули для работы с числами с плавающей точкой.

Каждая операция имеет множество входных операндов, множество выходных операндов и требующиеся для ее выполнения ресурсы. С каждым входным операндом связана задержка, указывающая, когда должно быть доступно входное значение (относительно начала операции). Типичные входные операнды имеют нулевую задержку, означающую, что их значения нужны немедленно, в момент выполнения операции. Аналогично с каждым выходным операндом связана выходная задержка, указывающая, когда становится доступен результат (относительно начала операции).

Использование ресурсов для каждой машинной команды типа t моделируется двумерной *таблицей резервирования ресурсов* (resource-reservation table) RT_t. Ширина таблицы равна количеству видов ресурсов машины, а длина — продолжительности использования ресурсов в операции. Элементы $RT_t[i,j]$ указывают количество единиц j-го ресурса, используемого операцией типа t спустя i тактов после начала ее выполнения. Для простоты обозначений мы считаем, что $RT_t[i,j] = 0$, если i указывает на несуществующий элемент таблицы (т.е. i больше количества тактов, затрачиваемых на выполнение операции). Конечно, для любых t, i и j значение $RT_t[i,j]$ должно быть не больше $R[j]$ — количества ресурсов типа j, имеющегося в машине.

Типичные машинные команды используют только одну единицу ресурсов во время выполнения операции. Некоторые операции могут использовать более одного функционального устройства. Например, операция умножения со сложением может на первом такте использовать умножитель, а на втором — сумматор. Некоторые операции, такие как деление, могут занимать ресурс несколько тактов. Операции *полностью конвейеризованные* (fully pipelined) — это операции, которые могут выполняться все такты подряд без каких-либо приостановок выполнения, несмотря на то, что результат их выполнения может стать доступным только спустя несколько тактов. Нам не нужно явно моделировать ресурсы на каждой стадии конвейера; достаточно одного модуля для представления первой стадии. Все операции, находящиеся на первой стадии конвейера, гарантированно пройдут все остальные стадии в последующие моменты времени.

10.2.8 Упражнения к разделу 10.2

Упражнение 10.2.1. Присваивания на рис. 10.5 имеют определенные зависимости. Для каждой из указанных пар инструкций классифицируйте зависимости как 1) истинные зависимости, 2) антизависимости, 3) зависимости через выход или 4) отсутствие зависимостей (т.е. инструкции могут выполняться в любом порядке).

а) Инструкции 1 и 4.

б) Инструкции 3 и 5.

в) Инструкции 1 и 6.

г) Инструкции 3 и 6.

д) Инструкции 4 и 6.

```
1)  a = b
2)  c = d
3)  b = c
4)  d = a
5)  c = d
6)  a = b
```

Рис. 10.5. Последовательность присваиваний с зависимостями через данные

Упражнение 10.2.2. Вычислите выражение $((u + v) + (w + x)) + (y + z)$ в точном соответствии со скобками (т.е. не используя законов коммутативности или ассоциативности для переупорядочения сложений). Приведите машинный код на уровне регистров для обеспечения максимальной степени параллельности.

Упражнение 10.2.3. Повторите упражнение 10.2.2 для следующих выражений:

а) $(u + (v + (w + x))) + (y + z)$;

б) $(u + (v + w)) + (x + (y + z))$.

Сколько шагов потребуется для вычислений при максимальной степени параллельности? Сколько шагов потребуется для выполнения вычислений, если вместо достижения максимальной степени параллельности мы будем минимизировать использование регистров?

Упражнение 10.2.4. Вычисления в упражнении 10.2.2 могут быть выполнены при помощи последовательности команд, показанной на рис. 10.6. Если в нашем распоряжении имеется машина параллельностью, удовлетворяющей любые наши запросы, сколько шагов потребуется для вычисления приведенных команд?

! **Упражнение 10.2.5.** Транслируйте фрагмент кода, рассматривавшийся в примере 10.4, с использованием команды условного копирования CMOVZ из раздела 10.2.6. Какие зависимости по данным имеются в полученном вами коде?

10.3 Планирование базовых блоков

Теперь мы готовы к тому, чтобы поговорить об алгоритме планирования кода. Начнем с простейшей задачи: планирования операций в базовом блоке, состоящем из машинных команд. Оптимальное решение этой задачи — NP-полное. Но на

```
 1) LD  r1, u          // r1 = u
 2) LD  r2, v          // r2 = v
 3) ADD r1, r1, r2     // r1 = r1 + r2
 4) LD  r2, w          // r2 = w
 5) LD  r3, x          // r3 = x
 6) ADD r2, r2, r3     // r2 = r2 + r3
 7) ADD r1, r1, r2     // r1 = r1 + r2
 8) LD  r2, y          // r2 = y
 9) LD  r3, z          // r3 = z
10) ADD r2, r2, r3     // r2 = r2 + r3
11) ADD r1, r1, r2     // r1 = r1 + r2
```

Рис. 10.6. Реализация арифметического выражения с минимальным использованием регистров

практике типичный базовый блок содержит только небольшое количество ограниченных операций, так что обычно должно быть вполне достаточно применения простых методов. Мы рассмотрим простой, но высокоэффективный алгоритм для решения этой задачи, называющийся *планирование списка* (list scheduling).

10.3.1 Графы зависимости данных

Представим каждый базовый блок машинных команд при помощи *графа зависимости данных* (data-dependence graph) $G = (N, E)$, содержащего множество узлов N, представляющих операции в машинных командах базового блока, и множества ориентированных ребер E, представляющих ограничения зависимостей данных между операциями. Узлы и ребра графа G строятся следующим образом.

1. Каждая операция $n \in N$ имеет таблицу резервирования ресурсов RT_n, которая представляет собой таблицу резервирования ресурсов для типа операции n.

2. Каждое ребро $e \in E$ помечено задержкой d_e, указывающей, что целевой узел должен быть выполнен не ранее, чем через d_e тактов после выполнения исходного узла. Предположим, что за операцией n_1 следует операция n_2, причем обе обращаются к одной и той же ячейке памяти с запаздываниями l_1 и l_2 соответственно. Иначе говоря, значение в ячейке создается через l_1 тактов после начала первой команды и требуется второй команде через l_2 тактов после ее начала (типичными являются значения $l_1 = 1$ и $l_2 = 0$). В таком случае в множестве E имеется ребро $n_1 \to n_2$, помеченное задержкой $l_1 - l_2$.

Пример 10.6. Рассмотрим простую машину, которая может выполнять в каждый момент времени две операции. Первая должна быть либо операцией ветвления, либо операцией АЛУ вида

```
OP dst, src1, src2
```

Вторая операция должна быть операцией загрузки или сохранения вида

```
LD dst, addr
ST addr, src
```

Операция загрузки (LD) полностью конвейеризуема и занимает два такта. Однако немедленно за загрузкой может следовать сохранение (ST), которое записывает считанные данные в память. Все прочие операции выполняются за один такт.

На рис. 10.7 приведен граф зависимости данных для примера базового блока и его требования к ресурсам. Можно представить, что R1 – указатель на стек, используемый для обращения к данным в стеке со смещениями 0 и 12. Первая команда загружает значение в регистр R2, и это значение становится доступным только спустя два такта после начала команды. В связи с этим ребра от первой команды ко второй и пятой, в каждой из которых требуется значение R2, помечены меткой 2. Аналогично задержка 2 указана и у ребра от третьей команды к четвертой; значение, загруженное в регистр R3, требуется четвертой команде и становится доступно через два такта после начала третьей команды.

Поскольку мы не знаем, как связаны между собой значения R1 и R7, мы должны рассматривать возможность того, что адрес наподобие 8(R1) тот же, что и адрес 0(R7). Иначе говоря, последняя команда может сохранять значение по тому же самому адресу, из которого выполняет чтение третья. Используемая нами модель машины позволяет выполнять сохранение в ячейке памяти через один такт после того, как мы выполнили чтение из нее, даже если за этот срок значение еще не появилось в регистре. Это замечание объясняет метку 1 у ребра от третьей команды к последней. Такая же аргументация поясняет наличие ребра от первой команды к последней и его метку 1. Оставшиеся ребра с метками 1 объясняются либо явными зависимостями, либо возможными зависимостями, связанными со значением R7. □

10.3.2 Планирование списков базовых блоков

Простейший подход к планированию базовых блоков включает посещение каждого узла графа зависимости данных в "приоритетном топологическом порядке". Поскольку в графе зависимости данных циклы отсутствуют, всегда имеется

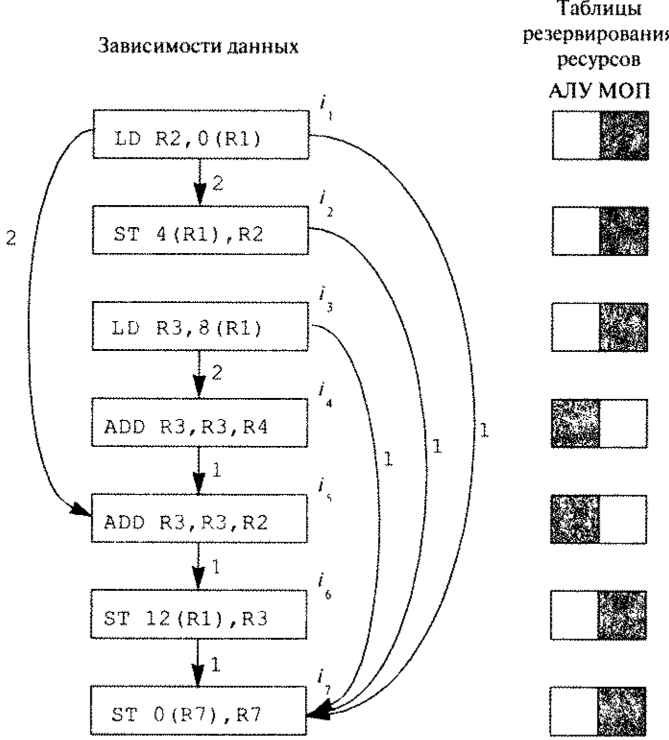


Рис. 10.7. Граф зависимости данных к примеру 10.6

как минимум одно топологическое упорядочение его узлов. Однако среди возможных топологических упорядочений одни предпочтительнее других. В разделе 10.3.3 будут рассмотрены некоторые стратегии выбора топологического упорядочения, но пока что мы просто будем считать, что существует некоторый алгоритм выбора предпочтительного упорядочения.

Описанный далее алгоритм планирования списка посещает узлы в выбранном приоритетном топологическом порядке. Узлы могут быть спланированы к выполнению как в порядке посещения, так и в некотором ином порядке. Но при этом команды планируются к выполнению как можно ранее, так что порядок планирования команд приближается к порядку посещения узлов.

Говоря более детально, алгоритм вычисляет наиболее раннее время, когда может быть вычислен каждый из узлов, в соответствии с ограничениями зависимости данных ранее спланированных узлов. Затем выполняется проверка наличия требующихся для узла ресурсов в таблице резервирования ресурсов, в которой собраны все выделенные к настоящему времени ресурсы. Узел планируется к выполнению в самый ранний момент, в который имеется достаточно ресурсов.

Графическое изображение таблиц резервирования ресурсов

Зачастую удобно визуализировать таблицу резервирования ресурсов для операции в виде сетки с закрашенными и пустыми ячейками. Каждый столбец соответствует одному из ресурсов машины, а каждая строка — одному из тактов выполнения операции. В предположении, что никакая операция не требует более одной единицы одного ресурса, можно представить единицы закрашенными квадратами, а нули — пустыми. Кроме того, если операция полностью конвейеризуема, то все, что нам надо, — указать используемые ресурсы в первой строке, и таблица резервирования ресурсов при этом превратится в единственную строку.

Такое представление использовано, например, в примере 10.6. На рис. 10.7 таблицы резервирования ресурсов показаны в виде строк. Две операции сложения требуют ресурс АЛУ, а операции загрузки и сохранения — ресурс МОП.

Алгоритм 10.7. Планирование списка базового блока

ВХОД: вектор ресурсов машины $R = [r_1, r_2, \ldots]$, где r_i — количество доступных единиц ресурса i-го вида, и граф зависимости данных $G = (N, E)$. Каждой операции $n \in N$ сопоставлена таблица резервирования ресурсов RT_n; каждое ребро $e = n_1 \to n_2$ из E имеет метку d_e, указывающую, что n_2 должна выполняться не ранее чем через d_e тактов после n_1.

ВЫХОД: план S, который отображает операции из N на временны́е интервалы, когда удовлетворяются все ограничения по ресурсам для данных операций и когда может начинаться выполнение последних.

МЕТОД: выполнить программу, приведенную на рис. 10.8. Вопрос о том, что такое "приоритетный топологический порядок", рассматривается в разделе 10.3.3. □

10.3.3 Приоритетные топологические порядки

Планирование списка работает без откатов: каждый узел планируется раз и только раз. Это планирование использует эвристическую функцию приоритета для выбора очередного узла среди узлов, готовых к планированию. Вот некоторые наблюдения о возможных приоритетных упорядочениях узлов.

- При отсутствии ограничений, связанных с ресурсами, кратчайший план получается при помощи *критического пути* (critical path) — самого длинного пути в графе зависимости данных. Метрика, используемая в качестве

```
RT = пустая таблица резервирования ресурсов;
for (каждый n ∈ N в приоритетном топологическом порядке) {
    s = max_{(e=p→n)∈E} (S(p) + d_e);
        /* Поиск самого раннего времени возможного начала команды
           на основе заданного времени начала ее предшественницы. */
    while (существует i такое, что RT[s + i] + RT_n[i] > R)
        s = s + 1;
            /* Задержка в выполнении команды до тех пор, пока не будут
               доступны все необходимые ресурсы. */
    S(n) = s;
    for (всех i)
        RT[s + i] = RT[s + i] + RT_n[i]
}
```

Рис. 10.8. Алгоритм планирования списка

функции приоритета, — *высота* узла, представляющая собой длину самого длинного пути от данного узла в графе.

- С другой стороны, если все операции независимы, то длина плана ограничена доступными ресурсами. Критический ресурс — это ресурс с наибольшим отношением количества использований к количеству доступных единиц ресурса. Более высокий приоритет могут иметь операции, использующие более критические ресурсы.

- Наконец, для разрешения неопределенностей можно использовать исходный порядок операций — операция, находящаяся в исходной программе раньше других, должна быть спланирована первой.

Пример 10.8. В случае графа зависимости данных, приведенного на рис. 10.7, критический путь имеет длину 6 тактов, включая время выполнения последней команды. Это путь, включающий пять последних узлов, — от загрузки R3 до сохранения R7. Суммарные задержки вдоль ребер этого пути равны 5, и к ним мы добавляем еще одну единицу — такт, необходимый для выполнения последней команды.

При использовании высоты в качестве функции приоритета алгоритм 10.7 дает оптимальный план, показанный на рис. 10.9. Обратите внимание, что план начинает с загрузки R3, поскольку эта команда имеет наибольшую высоту. Сложение R3 и R4 используют ресурсы, которые позволяют спланировать эту команду на второй такт, однако задержка 2 у команды загрузки заставляет нас спланировать эту команду на третий такт — иначе нельзя гарантировать, что регистр R3 будет иметь необходимое значение при выполнении сложения. □

План		Таблица резервирования ресурсов АЛУ	МОП
	LD R3,8(R1)		■
	LD R2,0(R1)		■
ADD R3, R3,R4		■	
ADD R3, R3,R2	ST 4(R1),R2	■	■
	ST 12(R1),R3		■
	ST 0(R7),R7		■

Рис. 10.9. Результат применения планирования списка к примеру на рис. 10.7

10.3.4 Упражнения к разделу 10.3

Упражнение 10.3.1. Для каждого из фрагментов кода на рис. 10.10 постройте граф зависимости данных.

	а)	б)	в)
1)	LD R1, a	LD R1, a	LD R1, a
2)	LD R2, b	LD R2, b	LD R2, b
3)	SUB R3, R1, R2	SUB R1, R1, R2	SUB R3, R1, R2
4)	ADD R2, R1, R2	ADD R2, R1, R2	ADD R4, R1, R2
5)	ST a, R3	ST a, R1	ST a, R3
6)	ST b, R2	ST b, R2	ST b, R4

Рис. 10.10. Машинный код к упражнению 10.3.1

Упражнение 10.3.2. Предположим, что у машины имеется только один ресурс АЛУ (для операций ADD и SUB) и один ресурс МОП (для операций LD и ST). Предположим, что все операции выполняются за один такт, кроме операции LD, требующей для выполнения два такта. Однако, как в примере 10.6, операция ST при работе с той же ячейкой памяти, что и предшествующая ей операция LD, может начинаться через один такт после начала операции LD. Найдите кратчайшие планы для каждого из фрагментов кода на рис. 10.10.

Упражнение 10.3.3. Повторите упражнение 10.3.2 в предположении, что

1. у машины один ресурс АЛУ и два ресурса МОП;
2. у машины два ресурса АЛУ и один ресурс МОП;

3. у машины два ресурса АЛУ и два ресурса МОП.

Упражнение 10.3.4. Предположим, что мы работаем с моделью машины из примера 10.6 (как в упражнении 10.3.2).

а) Изобразите граф зависимости данных для кода на рис. 10.11.

б) Укажите критические пути для построенного вами графа.

! в) Укажите все возможные планы для этих семи команд в предположении неограниченности ресурса МОП.

```
1) LD R1, a
2) ST b, R1
3) LD R2, c
4) ST c, R1
5) LD R1, d
6) ST d, R2
7) ST a, R1
```

Рис. 10.11. Машинный код к упражнению 10.3.4

10.4 Глобальное планирование кода

Для машины со средним параллелизмом на уровне команд планы, создаваемые путем работы с индивидуальными базовыми блоками, как правило, приводят к простаиванию большого количества ресурсов. Для более эффективного использования машинных ресурсов необходимо рассмотреть стратегии генерации кода, которые перемещают команды между базовыми блоками. Такие стратегии, которые рассматривают одновременно больше одного базового блока, называются алгоритмами *глобального планирования*. Для корректного глобального планирования следует рассматривать не только зависимости данных, но и зависимости управления. Мы должны гарантировать, что

1. оптимизированная программа выполняет все команды исходной программы;

2. хотя оптимизированная программа и может выполнять некоторые команды с упреждением, они не имеют никаких нежелательных побочных действий.

10.4.1 Примитивное перемещение кода

Рассмотрим вопросы, возникающие при перемещении кода, на простом примере.

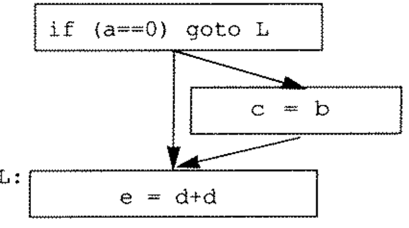
Пример 10.9. Предположим, что наша машина в состоянии выполнить любые две операции за один такт. Каждая операция выполняется с задержкой в один такт, за исключением операции загрузки, задержка которой составляет два такта. Для простоты мы считаем, что все обращения к памяти в примере корректны и удовлетворяются кэшем. На рис. 10.12, *а* показан простой граф потока с тремя базовыми блоками. Преобразованный в машинные команды код показан на рис. 10.12, *б*. Все команды в каждом базовом блоке из-за зависимости данных должны выполняться последовательно, так что в каждый базовый блок вставлены команды "нет операции".

Предположим, что адреса переменных a, b, c, d и e различны и что эти адреса хранятся в регистрах с R1 по R5 соответственно. Таким образом, вычисления в разных базовых блоках не зависимы по данным. Заметим, что все операции в базовом блоке B_3 выполняются независимо от того, по какому из путей ветвления пойдет программа, а значит, они могут выполняться параллельно с операциями из базового блока B_1. Перемещать операции из базового блока B_1 в базовый блок B_3 мы не можем, поскольку они необходимы для определения выбираемого пути ветвления.

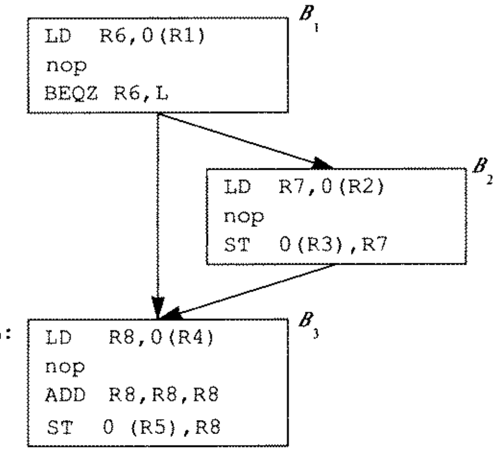
Операции в базовом блоке B_2 зависят по управлению от проверки в блоке B_1. Можно выполнить упреждающую загрузку из блока B_2 в блоке B_1, сэкономив тем самым 2 такта времени выполнения.

Упреждающие сохранения выполнять нельзя, поскольку они переписывают старые значения в ячейках памяти. Однако операции сохранения можно отложить. Мы не можем просто перенести операцию сохранения из базового блока B_2 в блок B_3, поскольку она должна выполняться только в том случае, когда поток управления проходит через базовый блок B_2. Однако можно поместить операцию сохранения в копию блока B_3. На рис. 10.12, *в* показан такой оптимизированный план. Оптимизированный код выполняется за четыре такта (это то же время, которое требовалось для выполнения одного лишь блока B_3 до оптимизации). □

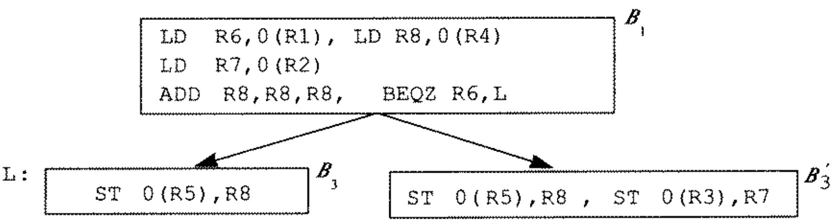
В примере 10.9 показана допустимость перемещения операций вверх и вниз по пути выполнения. Каждая пара базовых блоков в данном примере имеет различное "отношение доминирования", а потому различно и определение, когда и как можно выполнить перемещение команд в пределах пары. Как говорилось в разделе 9.6.1, базовый блок B доминирует над блоком B', если любой путь от входа в граф потока управления к базовому блоку B' проходит через блок B. Аналогично базовый блок B *постдоминирует* (postdominate) над блоком B', если любой путь от базового блока B' к выходу из графа потока проходит через базовый блок B. Если базовый блок B доминирует над базовым блоком B', а ба-



а) Исходная программа



б) Локально спланированный машинный код



в) Глобально спланированный машинный код

Рис. 10.12. Графы потоков до и после глобального планирования в примере 10.9

зовый блок B' постдоминирует над базовым блоком B, мы говорим, что базовые блоки B и B' *эквивалентны с точки зрения управления*, что означает, что один из

этих базовых блоков выполняется тогда и только тогда, когда выполняется второй. В примере на рис. 10.12, считая базовый блок B_1 входом, а базовый блок B_3 — выходом, мы получим следующие взаимоотношения базовых блоков.

1. Базовые блоки B_1 и B_3 эквивалентны с точки зрения управления: B_1 доминирует над B_3, а B_3 постдоминирует над B_1.

2. Базовый блок B_1 доминирует над базовым блоком B_2, но блок B_2 не постдоминирует над базовым блоком B_1.

3. Базовый блок B_2 не доминирует над базовым блоком B_3, но базовый блок B_3 постдоминирует над блоком B_2.

Пара блоков вдоль пути выполнения может также не быть связана ни отношением доминирования, ни отношением постдоминирования.

10.4.2 Восходящее перемещение кода

Давайте теперь внимательно рассмотрим, что собой представляет перемещение кода вверх по пути выполнения. Предположим, что мы хотим переместить операцию из блока *src* вверх по пути управления в базовый блок *dst*. Предположим, что такое перемещение не нарушает никакие зависимости данных и что оно делает пути, проходящие через *dst* и *src*, более быстро выполняющимися. Если *dst* доминирует над *src*, то перемещенная операция выполняется раз и только раз, когда она должна быть выполнена.

Если *src* не постдоминирует над *dst*

В таком случае существует путь, который проходит через *dst*, но не достигает *src*, и в результате могут быть выполнены лишние операции. Код не должен приводить к нежелательным побочным эффектам, иначе такое перемещение кода некорректно. Если выполнение перемещенной операции оказывается "бесплатным", т.е. оно использует только те ресурсы, которые иначе простаивали бы, то такое перемещение также оказывается бесплатным и повышает эффективность выполнения программы только в том случае, когда поток управления достигает *src*.

Если *dst* не доминирует над *src*

В этом случае существует путь, который достигает *src*, не проходя при этом через *dst*. Мы должны вставить копии перемещаемой операции вдоль каждого такого пути. Как это делается, мы знаем из рассмотрения устранения частичной избыточности в разделе 9.5. Мы помещаем копии операции в базовых блоках, которые образуют разрез, отделяющий входной блок от *src*. В каждом месте, в которое вставляется операция, должны удовлетворяться следующие ограничения.

1. В операндах операции должны храниться те же значения, что и в исходной операции.
2. Результат выполнения операции не должен перезаписывать значение, которое потребуется в дальнейшем.
3. Этот результат также никем не перезаписывается до достижения *src*.

Такие копии делают исходную команду в *src* полностью избыточной, и она может быть удалена.

Будем говорить о дополнительных копиях операции как о *компенсирующем коде* (compensation code). Как говорилось в разделе 9.5, для хранения таких копий могут использоваться базовые блоки, вставленные на критических путях. Компенсирующий код потенциально в состоянии сделать некоторые из путей более медленными, так что перемещение кода повышает производительность программы, только если оптимизированные пути выполняются более часто, чем неоптимизированные.

10.4.3 Нисходящее перемещение кода

Предположим, что мы хотим перенести операцию из базового блока *src* вдоль пути потока управления вниз, в базовый блок *dst*. К такому переносу применимы те же рассуждения, что и при восходящем переносе.

Если *src* не доминирует над *dst*

В таком случае существует путь, который достигает *dst*, не посетив перед этим *src*. В этом случае, как и ранее, будут выполняться дополнительные операции. К сожалению, нисходящее перемещение кода зачастую применяется к операциям записи, которые обладают побочным действием — перезаписыванием старого значения. Обойти эту неприятность можно, создав дубликаты базовых блоков на пути от *src* к *dst* и разместив операции только в новых копиях *dst*. Другой подход состоит в использовании предикатных команд (конечно, если таковые доступны). Перемещенная операция защищена тем же предикатом, что и блок *src*. Заметим, что предикатная команда может размещаться только в блоке, над которым доминирует вычисление предиката, поскольку иначе предикат будет недоступен.

Если *dst* не постдоминирует над *src*

Как и ранее, следует использовать компенсирующий код, чтобы перемещаемая операция выполнялась на всех путях, не проходящих через *dst*. Такая трансформация аналогична устранению частичной избыточности, но копии размещаются ниже блока *src*, в разрезе, который отделяет базовый блок *src* от выхода.

Резюме по восходящему и нисходящему перемещениям кода

Из вышесказанного ясно, что существует область возможных глобальных перемещений кода, в которой оказываются различными стоимость, сложность реализации и повышение эффективности программы. На рис. 10.13 сведены различные варианты перемещения кода. Строки таблицы соответствуют следующим четырем случаям.

	↑:*src* постдоминирует над *dst*	*dst* доминирует над *src*	Упреждение	Компенсирующий код
	↓:*src* доминирует над *dst*	*dst* постдоминирует над *src*	Дублирование кода	
1	Да	Да	Нет	Нет
2	Нет	Да	Да	Нет
3	Да	Нет	Нет	Да
4	Нет	Нет	Да	Да

Рис. 10.13. Перемещение кода

1. Перемещение команд между эквивалентными с точки зрения управления базовыми блоками — простейшее и наиболее эффективное с точки зрения стоимости. Не требуется никаких дополнительных операций и никакого компенсирующего кода.

2. Если исходный блок не постдоминирует (доминирует) над целевым базовым блоком при восходящем (нисходящем) перемещении, может потребоваться выполнение дополнительных операций. Такое перемещение кода выгодно в том случае, когда дополнительные операции могут быть выполнены бесплатно, т.е. с использованием только тех ресурсов, которые иначе простаивали бы, и когда выполняется путь, проходящий через исходный блок.

3. Если целевой блок не доминирует (постдоминирует) над исходным базовым блоком при восходящем (нисходящем) перемещении, требуется компенсирующий код. Пути с компенсирующим кодом могут оказаться замедленными, поэтому важно, чтобы оптимизированные пути выполнялись чаще, чем неоптимизированные.

4. Последний случай объединяет недостатки второго и третьего случаев: требуется компенсирующий код и может потребоваться выполнение дополнительных операций.

10.4.4 Обновление зависимостей данных

Как показано в приведенном ниже примере 10.10, перемещение кода может изменить отношения зависимости данных между операциями. Таким образом, после каждого перемещения кода следует выполнять обновление зависимостей данных.

Пример 10.10. В случае графа потока на рис. 10.14 в верхний блок может быть перенесено любое присваивание x, поскольку при таком преобразовании сохраняются все зависимости исходной программы. Однако после того, как одно из присваиваний будет перенесено вверх, второе переносить будет нельзя. Точнее говоря, переменная x на выходе из верхнего блока до перемещения не активна, но становится таковой после перемещения. Если в некоторой точке программы переменная активна, то упреждающее определение этой переменной не может быть перемещено выше данной точки. □

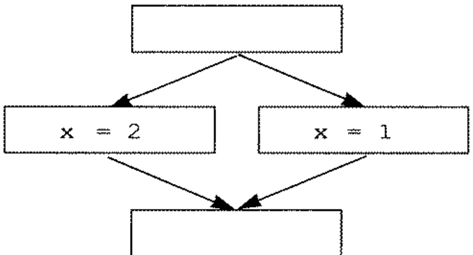


Рис. 10.14. Пример, иллюстрирующий изменение зависимости данных из-за перемещения кода

10.4.5 Алгоритм глобального планирования

В последнем разделе мы видели, что перемещение кода может привести к повышению эффективности некоторых из путей выполнения и снижению эффективности прочих. Хорошая же новость состоит в том, что не все команды равны — одни из них более равны, чем другие. Не секрет, что в действительности 90% времени работы программы тратится на выполнение 10% кода. Таким образом, наша цель должна состоять в том, чтобы сделать часто выполнимые пути как можно более быстрыми, возможно, за счет того, что более редкие пути станут более медленными.

Имеется множество методов, которые компилятор может использовать для оценки частоты выполнения. Достаточно разумно считать, что команды во внутренних циклах выполняются гораздо чаще, чем код внешних циклов, и что при ветвлении чаще выбираются пути, ведущие назад, чем вперед. Кроме того, крайне

редко должны отрабатываться ветвления, ведущие к выходу из программы или в подпрограммы обработки исключений. Однако наилучшие оценки дает динамическое профилирование. При использовании этого метода в программу добавляется код для записи выбранных ветвей, после чего выполняется тестовый запуск программы с типичными входными данными. Результаты, полученные при помощи этого метода, оказываются достаточно точными. Такая информация может затем использоваться компилятором в процессе оптимизации программы.

Планирование на основе областей

Опишем несложный планировщик, который поддерживает две простейшие разновидности перемещения кода.

1. Восходящее перемещение операций в блоки, эквивалентные с точки зрения управления.

2. Упреждающее восходящее перемещение операций по одной из ветвей в предшествующий доминирующий блок.

Вспомним, что в разделе 9.7.1 область определялась как подмножество графа потока управления, которое может быть достигнуто через один входной базовый блок. Любая процедура может быть представлена как иерархия областей. Полностью процедура состоит из области верхнего уровня со вложенными в нее подобластями, представляющими естественные циклы в функции. Граф потока управления мы считаем приводимым.

Алгоритм 10.11. Планирование на основе областей

ВХОД: граф потока управления и описание машинных ресурсов.

ВЫХОД: план S, отображающий каждую команду на базовый блок и интервал времени.

МЕТОД: выполняем программу, приведенную на рис. 10.15. Терминология программы должна быть очевидна: *ControlEquiv* (B) — множество блоков, эквивалентных с точки зрения управления с базовым блоком B, а функция *DominatedSucc*, примененная к множеству базовых блоков, дает множество базовых блоков, являющихся преемниками как минимум одного блока множества, для которых все блоки множества являются доминирующими.

Планирование кода в алгоритме 10.11 идет от наиболее глубоких внутренних областей к наружным. При планировании области каждая вложенная подобласть рассматривается как черный ящик; команды не могут перемещаться в подобласти или из них. Однако они могут перемещаться вокруг подобласти при условии удовлетворения ограничений зависимостей данных и управления.

Все ребра управления и зависимости, идущие назад к заголовку области, игнорируются, так что получающиеся в результате графы потоков управления и зави-

```
for (каждая область R в топологическом порядке, так что внутренние
        области обрабатываются до внешних) {
    Вычисление зависимостей данных;
    for (каждый базовый блок B области R в приоритетном топологическом
            порядке) {
        CandBlocks = ControlEquiv(B) ∪ DominatedSucc(ControlEquiv(B));
        CandInsts = готовые команды в CandBlocks;
        for (t = 0, 1, ..., пока не будут спланированы все команды из B) {
            for (каждая команда n из CandInsts в приоритетном порядке)
                if (n не имеет конфликтов ресурсов в момент t) {
                    S(n) = ⟨B, t⟩;
                    Обновление имеющихся ресурсов;
                    Обновление зависимостей данных;
                }
            Обновление CandInsts;
        }
    }
}
```

Рис. 10.15. Алгоритм глобального планирования на основе областей

симости данных ацикличны. Базовые блоки в каждой области посещаются в топологическом порядке. Такое упорядочение гарантирует, что базовый блок не будет планироваться до тех пор, пока не будут спланированы все команды, от которых он зависит. Команды, которые должны быть спланированы в базовом блоке B, выводятся изо всех блоков, эквивалентных с точки зрения управления блоку B (включая сам базовый блок B), а также из непосредственных преемников блока B, над которыми он доминирует.

Алгоритм планирования списка используется для создания плана для каждого базового блока. Этот алгоритм поддерживает список команд-кандидатов *CandInsts*, который содержит все команды в блоках-кандидатах, все предшественники которых уже спланированы. План создается такт за тактом. Для каждого такта в порядке приоритета проверяется каждая команда из списка *CandInsts* и планируется, если позволяют ресурсы. Затем алгоритм 10.11 обновляет список *CandInsts* и повторяет процесс до тех пор, пока не будут спланированы все команды из базового блока B.

Приоритетный порядок команд в списке *CandInsts* использует функцию приоритета, подобную рассмотренной в разделе 10.3, в которую внесено одно важное изменение. Командам из блоков, эквивалентных с точки зрения управления блоку B, назначается более высокий приоритет, чем командам из блоков-преемников.

Причина этого в том, что команды из блоков-преемников в базовом блоке B могут вычисляться только с упреждением. □

Развертка циклов

При планировании на основе областей граница итерации цикла является барьером для перемещения кода. Операции из одной итерации не могут перекрываться операциями из другой. Один простой, но эффективный метод смягчения этой проблемы состоит в небольшой развертке цикла перед планированием кода. Цикл for наподобие

```
for(i = 0; i < N; i++) {
    S(i);
}
```

можно переписать так, как показано на рис. 10.16, *а*. Аналогично цикл repeat

```
repeat
    S;
until C;
```

может быть переписан так, как показано на рис. 10.16, *б*. Развертывание создает большее количество команд в теле цикла, позволяя алгоритму глобального планирования достигать большей степени параллелизма.

Уплотнение окрестностей

Алгоритм 10.11 поддерживает только два первых вида перемещения кода, описанных в разделе 10.4.1. Однако иногда полезным может оказаться и перемещение, которое требует добавления компенсирующего кода. Один из вариантов поддержки такого перемещения кода состоит в следовании планированию на основе областей с простым проходом. При выполнении этого прохода можно просмотреть все пары базовых блоков, выполняющиеся один за другим, и проверить, нельзя ли переместить какую-либо операцию между ними для снижения общего времени выполнения этих базовых блоков. Если такая пара найдена, то следует проверить, не надо ли продублировать перемещаемую команду на других путях. Перемещение выполняется, если оно дает чистый выигрыш времени выполнения.

Такое простое расширение алгоритма может оказаться достаточно эффективным для повышения производительности циклов. Например, оно может переместить операцию в начале одной итерации в конец предыдущей, а операцию из первой итерации вынести за пределы цикла. Такая оптимизация особенно привлекательна для небольших циклов, итерации которых состоят всего лишь из нескольких команд. Однако возможности этого метода ограничены тем фактом, что каждое решение о перемещении кода принимается локально и независимо от других.

```
for (i = 0; i+4 < N; i+=4) {
    S(i);
    S(i+1);
    S(i+2);
    S(i+3);
}
for ( ; i < N; i++) {
    S(i);
}
```

а) Развертка цикла for

```
repeat {
    S;
    if (C) break;
    S;
    if (C) break;
    S;
    if (C) break;
    S;
} until C;
```

б) Развертка цикла repeat

Рис. 10.16. Развертка циклов

10.4.6 Усовершенствованные методы перемещения кода

Если наша целевая машина статически планируема и обладает высоким параллелизмом на уровне команд, то может потребоваться более агрессивный алгоритм. Вот высокоуровневое описание дальнейших усовершенствований алгоритма.

1. Для упрощения рассматривающихся ниже расширений можно добавить новые базовые блоки вдоль ребер потока управления, исходящих из блоков с более чем одним предшественником. Эти базовые блоки будут устранены в конце планирования кода, если окажутся пустыми. Одна из полезных эвристик состоит в перемещении команд из почти пустых базовых блоков, чтобы затем полностью устранить такой блок.

2. В алгоритме 10.11 код, который будет выполняться в каждом базовом блоке, планируется раз и навсегда при посещении блока. Такого простого подхода достаточно, поскольку алгоритм может перемещать операции только вверх в доминирующие блоки. Чтобы разрешить перемещения, которые требуют

добавления компенсирующего кода, используется несколько иной подход. При посещении базового блока B мы планируем только команды этого блока B и блоков, эквивалентных ему с точки зрения управления. Мы сначала пытаемся разместить эти команды в предшествующих блоках, которые уже были посещены и для которых уже существуют частичные планы. Мы пытаемся найти целевой блок, который может привести к улучшению часто выполняемого пути, а затем помещаем копии команды на других путях для обеспечения корректности программы. Если команда не может быть перемещена вверх, она, как и ранее, планируется в текущем базовом блоке.

3. Реализация нисходящего перемещения кода в алгоритме, который посещает базовые блоки в топологическом порядке, оказывается более сложной, поскольку целевые блоки при этом еще не спланированы. Однако все равно имеется несколько возможностей и для такого перемещения кода. Итак, мы перемещаем все операции, которые

   а) могут быть перемещены;

   б) не могут быть выполнены бесплатно в их исходных блоках.

   Такая простая стратегия хорошо работает на целевых машинах с большим количеством неиспользуемых аппаратных ресурсов.

10.4.7 Взаимодействие с динамическими планировщиками

Динамический планировщик обладает тем преимуществом, что он может создавать новые планы в зависимости от условий времени выполнения, не рассматривая заблаговременно все возможные планы. Если целевая машина оснащена динамическим планировщиком, основная функция статического планировщика заключается в обеспечении ранней выборки команд с большими задержками, чтобы динамический планировщик мог выполнить их как можно раньше.

Промахи кэша представляют собой класс непредсказуемых событий, которые могут привести к большим разбросам производительности программы. Если доступны команды предвыборки данных, статический планировщик может существенно помочь динамическому, размещая эти команды как можно раньше, чтобы к тому моменту, когда потребуются некоторые данные, они уже были в кэше. Если же такие команды у целевой машины недоступны, то неплохо, если компилятор в состоянии оценить, какие данные вызовут промахи кэша, и попытается загрузить их как можно ранее.

Если целевая машина не оснащена динамическим планировщиком, статический планировщик должен быть консервативен и разделять все пары команд

с зависимостями данных минимальной задержкой. Однако при наличии динамического планировщика от компилятора требуется только разместить зависимые по данным операции в правильном порядке, чтобы гарантировать корректность программы. Для достижения наилучшей производительности компилятор должен назначать большие задержки наиболее вероятным зависимостям и небольшие — маловероятным.

Важной причиной потери производительности является неверное предсказание ветвлений. В связи с этим наивысший приоритет при планировании должен назначаться командам, которые позволяют снизить стоимость неверного предсказания ветвления.

10.4.8 Упражнения к разделу 10.4

Упражнение 10.4.1. Покажите, каким образом можно развернуть обобщенный цикл while: `while (C) S;`.

Упражнение 10.4.2. Рассмотрим следующий фрагмент кода:

```
if (x == 0) a = b;
else a = c;
d = a;
```

Будем считать, что машина использует модель задержек из примера 10.6 (загрузка занимает два такта, все прочие команды — по одному такту). Предположим также, что машина в состоянии выполнять одновременно две команды. Найдите наикратчайшее возможное выполнение этого фрагмента. Не забудьте рассмотреть вопрос о том, какие регистры лучше всего использовать для каждого шага копирования. Чтобы избежать излишних загрузок и сохранений, не забудьте воспользоваться информацией, которую дают дескрипторы регистров, описанные в разделе 8.6.

10.5 Программная конвейеризация

Как говорилось во введении к данной главе, обычно численные приложения обладают высокой степенью параллелизма. В частности, в них зачастую имеются циклы, итерации которых совершенно независимы одна от другой. Такие циклы, известные также как *универсальные* (do-all cycles), особенно привлекательны с точки зрения параллелизма, позволяя полностью использовать все ресурсы и все возможности параллельных вычислений процессора. В этом разделе будет рассмотрен алгоритм, известный как *программная конвейеризация* (software pipelining), планирующий весь цикл целиком, который обеспечивает максимальное использование возможностей параллельных вычислений.

10.5.1 Введение

В примере 10.12 приведен универсальный цикл, который будет использоваться во всем разделе для пояснения программной конвейеризации. Сначала будет показана важность межитерационного планирования, связанная с относительно малым параллелизмом операций в пределах одной итерации. Затем мы покажем, как развертка цикла повышает производительность путем перекрытия развернутых итераций. Однако граница развернутого цикла остается непреодолимым барьером для перемещения кода, и развертка оставляет немало неиспользованных возможностей для повышения производительности кода. Метод же программной конвейеризации допускает перекрытие нескольких последовательных итераций, пока все они не будут выполнены, тем самым позволяя получить высокоэффективный и компактный код.

Пример 10.12. Вот пример типичного универсального цикла:

```
for(i = 0; i < n; i++)
    D[i] = A[i] * B[i] + c;
```

Итерации этого цикла выполняют запись в разные ячейки памяти, которые, в свою очередь, не совпадают ни с одним из адресов, по которым выполняется чтение данных. Следовательно, между итерациями не имеется никаких зависимостей данных, и все итерации могут выполняться параллельно.

В этом разделе мы примем следующую модель целевой машины.

- За один такт машина может выполнить: одну загрузку, одно сохранение, одну арифметическую операцию или одну операцию ветвления.
- Машина имеет операцию цикла вида

  ```
  BL R, L
  ```

  Эта операция уменьшает значение регистра R и, если оно не равно 0, осуществляет переход к L.
- Операции с памятью могут выполняться в автоинкрементном режиме, на что указывают символы ++ после регистра. Значение регистра автоматически увеличивается с тем, чтобы после каждого обращения указывать на следующий адрес в памяти.
- Арифметические операции полностью конвейеризуемы. Они могут быть инициированы на любом такте, но их результаты становятся доступны два такта спустя. Задержка всех прочих команд — один такт.

Если итерации выполняются по одной, то наилучший план, который можно получить в рамках нашей машинной модели, показан на рис. 10.17. На этом рисунке указаны некоторые предположения о размещении данных: регистры R1, R2 и R3 хранят адреса начала массивов A, B и D; регистр R4 хранит константу c, а регистр R10 — значение $n - 1$, вычисляемое вне цикла. Такое вычисление, в основном, последовательное и занимает семь тактов; единственное имеющееся перекрытие — команды цикла и последней операции в итерации. □

```
   //  R1, R2, R3 = &A, &B, &D
   //  R4         = c
   //  R10        = n-1

L: LD  R5, 0(R1++)
   LD  R6, 0(R2++)
   MUL R7, R5, R6
   nop
   ADD R8, R7, R4
   nop
   ST  0(R3++), R8      BL R10, L
```

Рис. 10.17. Локально спланированный код примера 10.12

В общем случае добиться повышения степени использования аппаратного обеспечения можно путем развертки нескольких итераций цикла. Однако это приводит к увеличению размера кода, что, в свою очередь, может привести к отрицательному влиянию на общую производительность. Таким образом, следует искать компромисс — количество итераций, развертка которых даст наибольшее повышение эффективности без слишком сильного увеличения кода. Следующий пример иллюстрирует такой компромисс.

Пример 10.13. В то время как обнаружить параллелизм в пределах одной итерации очень сложно, между итерациями его сколько угодно. Развертка цикла помещает несколько итераций цикла в один большой базовый блок, после чего для планирования параллельного выполнения операций можно использовать простой алгоритм планирования списка. Если в нашем примере мы четырежды развернем цикл и применим алгоритм 10.7, то получим план, показанный на рис. 10.18 (для простоты мы пока что игнорируем детали распределения регистров). Цикл выполняется за 13 тактов, т.е. продолжительность выполнения одной итерации падает до 3,25 такта.

Цикл, развернутый k раз, требует как минимум $2k + 5$ тактов, достигая, таким образом, выполнения одной итерации за $2 + 5/k$ тактов. Следовательно, чем больше итераций будет развернуто, тем быстрее будет выполнен цикл. При $k \to \infty$

```
L:  LD
    LD
         LD
    MUL  LD
         MUL  LD
    ADD       LD
         ADD       LD
    ST        MUL  LD
         ST        MUL
              ADD
                   ADD
              ST
                   ST   BL (L)
```

Рис. 10.18. Развернутый код из примера 10.12

полностью развернутый цикл может выполнять в среднем одну итерацию за два такта. Однако чем больше итераций будет развернуто, тем большего размера код будет получен; так что развернуть все итерации цикла, определенно, не получится. Развертка 4 итераций дает код, состоящий из 13 команд, или 163% от оптимального значения; развертка 8 итераций дает 21 команду, или 131% от оптимального значения. Можно пойти и в обратном направлении; например, чтобы получить 110% от оптимального значения, следует развернуть 25 итераций, получая код с 55 командами. □

10.5.2 Программная конвейеризация циклов

Программная конвейеризация предоставляет удобный способ достичь оптимального использования ресурсов одновременно с компактным кодом. Проиллюстрируем лежащую в ее основе идею на нашем стандартном примере.

Пример 10.14. На рис. 10.19 показан пятикратно развернутый код из примера 10.12. (Как и ранее, вопрос распределения регистров остается за пределами нашего внимания.) В строке i показаны все операции, выполняемые на такте i; в столбце j показаны все команды итерации j. Заметим, что каждая итерация имеет один и тот же план выполнения относительно ее начала и что каждая итерация начинается через два такта после начала предыдущей. Легко видеть, что этот план удовлетворяет всем ограничениям, накладываемым ресурсами и зависимостями данных.

Мы видим, что на тактах 7 и 8 выполняются те же операции, что и на тактах 9 и 10. Во время тактов 7 и 8 выполняются операции из первых четырех итераций исходной программы; во время тактов 9 и 10 также выполняются операции из

Такт	$j = 1$	$j = 2$	$j = 3$	$j = 4$	$j = 5$
1	LD				
2	LD				
3	MUL	LD			
4		LD			
5		MUL	LD		
6	ADD		LD		
7			MUL	LD	
8	ST	ADD		LD	
9				MUL	LD
10		ST	ADD		LD
11					MUL
12			ST	ADD	
13					
14				ST	ADD
15					
16					ST

Рис. 10.19. Пятикратно развернутый код из примера 10.12

четырех итераций, но на этот раз — итераций со второй по пятую. План можно рассматривать как выполнение многооперационных команд — при этом работа цикла представляет собой завершение самой старой итерации и начало новой до тех пор, пока не будут выполнены все итерации.

Такое динамическое поведение можно сжато закодировать при помощи кода, показанного на рис. 10.20, если считать, что цикл содержит как минимум четыре итерации. Каждая строка на рисунке соответствует одной машинной команде. Строки 7 и 8 образуют двухтактный цикл, который выполняется $n - 3$ раза, где n — количество итераций в исходном цикле. □

Описанный выше метод называется *программной конвейеризацией* (software pipelining), поскольку он представляет собой программный аналог метода, используемого для планирования аппаратных конвейеров. План, выполняющий каждую итерацию в данном примере, можно рассматривать как восьмиэтапный конвейер. Новая итерация может начинаться в конвейере каждые два такта. В начале в конвейере находится только одна итерация. Когда первая итерация переходит на третий этап выполнения, с первого этапа начинается выполнение второй итерации.

На седьмом такте конвейер полностью заполнен первыми четырьмя итерациями. На этой стадии параллельно выполняются четыре соседние итерации. Когда наиболее старая итерация выполнена и покидает конвейер, начинается выпол-

```
 1      LD
 2      LD
 3      MUL  LD
 4           LD
 5           MUL  LD
 6      ADD       LD
 7  L:            MUL  LD
 8      ST   ADD       LD   BL (L)
 9                     MUL
10           ST   ADD
11
12                ST   ADD
13
14                     ST
```

Рис. 10.20. Программная конвейеризация кода из примера 10.12

нение новой итерации. Когда новых итераций нет, конвейер опустошается, и все находящиеся в нем итерации выполняются до конца. Последовательность команд, использующихся для заполнения конвейера (в нашем примере — команды с 1 по 6), называется *прологом*, строки 7 и 8 представляют *устойчивое состояние* (steady state), а последовательность команд, приводящих к опустошению конвейера (в нашем примере — команды с 9 по 14), называется *эпилогом*.

В нашем примере мы знаем, что цикл не может работать быстрее, чем итерация за два такта, поскольку машина не в состоянии выполнять более одного чтения за такт, а в каждой итерации их два. Рассмотренный программно конвейеризованный цикл выполняется за $2n + 6$ тактов, где n — количество итераций в исходном цикле. При $n \to \infty$скорость работы цикла стремится к одной итерации за два такта. Таким образом, в отличие от развертки циклов, программная конвейеризация потенциально способна получить оптимальный план при весьма компактной последовательности кода.

Заметим, что по отношению к отдельно взятой итерации этот план не является оптимальным. Сравнивая его с локально оптимизированным планом, показанным на рис. 10.17, мы видим излишнюю задержку перед операцией ADD. Эта стратегическая задержка необходима для того, чтобы выполнение каждой новой итерации могло начинаться спустя два такта после начала предыдущей без каких-либо конфликтов ресурсов. Если бы мы использовали локальный план, то во избежание конфликтов каждая новая итерация должна была бы начинаться через четыре такта после начала предыдущей и скорость работы программы упала бы вдвое. Этот пример иллюстрирует важный принцип конвейерного планирования: для оптимизации производительности следует очень аккуратно выбирать план. Локально

оптимизированный план, минимизирующий время выполнения одной итерации, может привести к неоптимальной производительности при конвейеризации.

10.5.3 Распределение регистров и генерация кода

Рассмотрим теперь вопросы распределения регистров для программно конвейеризованного цикла из примера 10.14.

Пример 10.15. В примере 10.14 результат операции умножения в первой итерации получается на третьем такте, а используется — на шестом. Между этими тактами, на пятом такте, генерируется новый результат операции умножения во второй итерации; это значение используется на восьмом такте. Результаты работы этих двух итераций должны храниться в разных регистрах для того, чтобы предотвратить их влияние друг на друга. Поскольку такое перекрытие временных интервалов существования результата осуществляется только для соседних итераций, достаточно использовать для хранения указанных значений два разных регистра: один — для четных итераций, другой — для нечетных. Поскольку код нечетных итераций отличается от кода четных, размер устойчивого состояния цикла удваивается. Такой код может использоваться для выполнения любого цикла с нечетным количеством итераций, не меньшим 5.

Для работы с циклами, состоящими менее чем из пяти итераций, и циклами с четным количеством итераций мы генерируем код, на уровне исходного текста эквивалентный показанному на рис. 10.21. Первый цикл конвейеризован, как видно из представленного на рис. 10.22 эквивалента программы на уровне машинных команд. Второй цикл на рис. 10.21 не требует оптимизации, так как в нем выполняется не более четырех итераций. □

```
if (N >= 5)
    N2 = 3 + 2 * floor((N-3)/2);
else
    N2 = 0;
for (i = 0; i < N2; i++)
    D[i] = A[i]* B[i] + c;
for (i = N2; i < N; i++)
    D[i] = A[i]* B[i] + c;
```

Рис. 10.21. Развертка цикла из примера 10.12 на уровне исходного текста

```
 1.     LD R5,0(R1++)
 2.     LD R6,0(R2++)
 3.     LD R5,0(R1++) MUL R7,R5,R6
 4.     LD R6,0(R2++)
 5.     LD R5,0(R1++) MUL R9,R5,R6
 6.     LD R6,0(R2++) ADD R8,R7,R4
 7. L:  LD R5,0(R1++) MUL R7,R5,R6
 8.     LD R6,0(R2++) ADD R8,R9,R4 ST 0(R3++),R8
 9.     LD R5,0(R1++) MUL R9,R5,R6
10.     LD R6,0(R2++) ADD R8,R7,R4 ST 0(R3++),R8 BL R10,L
11.                   MUL R7,R5,R6
12.                   ADD R8,R9,R4 ST 0(R3++),R8
13.
14.                   ADD R8,R7,R4 ST 0(R3++),R8
15.
16.                                ST 0(R3++),R8
```

Рис. 10.22. Код из примера 10.12 после программной конвейеризации и распределения регистров

10.5.4 Циклы с зависимыми итерациями

Программная конвейеризация применима и к циклам, у итераций которых имеются зависимости данных друг от друга. Такие циклы известны как *перекрестные* (do-across loops).

Пример 10.16. Код

```
for (i = 0; i < n; i++) {
    sum = sum + A[i];
    B[i] = A[i] * b;
}
```

содержит зависимости данных между соседними итерациями, поскольку при выполнении итерации для получения нового значения sum к старому значению sum прибавляется значение $A[i]$. Если машина обладает достаточной степенью параллелизма, то такое суммирование можно выполнить за время $O(\log n)$, но мы просто предположим, что все последовательные зависимости должны быть удовлетворены и что суммирование должно быть выполнено в исходном порядке. Поскольку в нашей модели машины выполнение команды ADD требует двух тактов, цикл не может выполняться быстрее, чем одна итерация за два такта. Ускорить выполнение не поможет никакое количество добавочных сумматоров или

умножителей. Производительность перекрестного цикла ограничена цепочкой зависимостей между итерациями.

На рис. 10.23, *а* показан локально оптимизированный план для каждой итерации, а на рис. 10.23, *б* — программно конвейеризованный код. Такой программно конвейеризованный цикл каждые два такта начинает выполнение новой итерации и, таким образом, работает с оптимальной скоростью. □

```
    // R1 = &A; R2 = &B
    // R3 = sum
    // R4 = b
    // R10 = n-1

L:  LD R5, 0(R1++)
    MUL R6, R5, R4
    ADD R3, R3, R4
    ST R6, 0(R2++)      BL R10, L
```

а) Локально оптимизированный план

```
    // R1 = &A; R2 = &B
    // R3 = sum
    // R4 = b
    // R10 = n-2

    LD R5, 0(R1++)
    MUL R6, R5, R4
L:  ADD R3, R3, R4    LD R5, 0(R1++)
    ST R6, 0(R2++)    MUL R6, R5, R4    BL R10, L
                      ADD R3, R3, R4
                      ST R6, 0(R2++)
```

б) Программно конвейеризованная версия

Рис. 10.23. Программная конвейеризация перекрестного цикла

10.5.5 Цели и ограничения программной конвейеризации

Основная цель программной конвейеризации — максимизация производительности долго выполняющихся циклов. Вторичная цель заключается в генерации кода относительно малого размера. Другими словами, программно конвейеризованный цикл должен иметь малое устойчивое состояние конвейера. Достичь

этого малого устойчивого состояния можно, если потребовать, чтобы относительные планы каждой итерации были одинаковы и чтобы итерации начинались через один и тот же постоянный интервал времени. Поскольку производительность цикла определяется как величина, обратная интервалу между запусками итераций, цель программной конвейеризации заключается в минимизации этого интервала.

План программной конвейеризации для графа зависимости данных $G = (N, E)$ может быть определен

1. интервалом между запусками итераций T;
2. относительным планом S, который для каждой операции указывает время ее выполнения относительно начала итерации, которой принадлежит эта команда.

Таким образом, операция n в i-й итерации, считая от нуля, выполняется в момент $i \times T + S(n)$. Подобно всем прочим задачам планирования, программная конвейеризация имеет два вида ограничений, связанных с ресурсами и зависимостями данных. Ниже мы рассмотрим каждый из этих типов ограничений.

Модульное резервирование ресурсов

Представим машинные ресурсы в виде $R = [r_1, r_2 \ldots]$, где r_i — количество единиц i-го вида ресурса. Если итерация цикла требует n_i единиц ресурса i, то средний интервал между запусками итераций конвейеризованного цикла составляет как минимум $\max_i (n_i/r_i)$ тактов. Программная конвейеризация требует, чтобы интервалы запуска между любыми соседними итерациями представляли собой одно и то же значение. Таким образом, интервал запуска должен состоять из как минимум $\max_i \lceil n_i/r_i \rceil$ тактов. Если $\max_i (n_i/r_i)$ меньше 1, то имеет смысл развернуть небольшое количество итераций.

Пример 10.17. Вернемся к нашему программно конвейеризованному циклу, показанному на рис. 10.20. Вспомним, что целевая машина может выполнить за один такт одну загрузку, одну арифметическую операцию, одно сохранение и одну операцию цикла. Поскольку цикл содержит две загрузки, две арифметические операции и одну операцию сохранения, минимальный интервал между запусками итераций, вычисленный на основании ограничений, связанных с ресурсами, равен двум тактам.

На рис. 10.24 показаны требования четырех последовательных итераций к ресурсам в разные моменты времени. Чем больше итераций запущено, тем выше требования к ресурсам; максимальные требования достигаются в устойчивом состоянии. Пусть RT — таблица резервирования ресурсов, представляющая запросы одной итерации, а RT_S представляет запросы в устойчивом состоянии. RT_S объединяет запросы четырех последовательных итераций, начавшихся в пределах

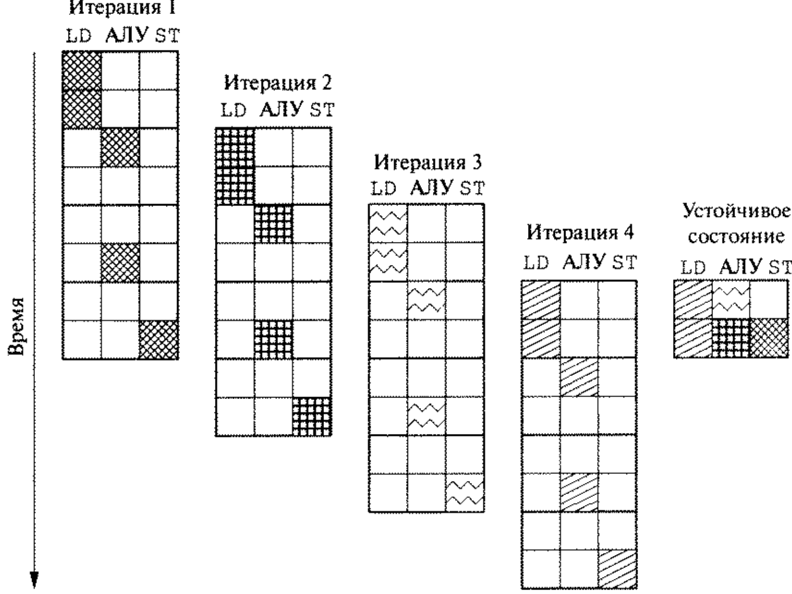


Рис. 10.24. Требования четырех последовательных итераций кода из примера 10.13 к ресурсам

T тактов. Запросы в строке 0 таблицы RT_S соответствуют сумме ресурсов, запрошенных в $RT[0]$, $RT[2]$, $RT[4]$ и $RT[6]$. Аналогично запросы в строке 1 соответствуют сумме ресурсов, запрошенных в $RT[1]$, $RT[3]$, $RT[5]$ и $RT[7]$, т.е. ресурсы, запрошенные в i-й строке в устойчивом состоянии, равны

$$RT_S[i] = \sum_{\{t|(t \bmod 2)=i\}} RT[t].$$

Будем говорить о таблице резервирования ресурсов, представляющей устойчивое состояние, как о *таблице модульного резервирования ресурсов* (modular resource-reservation table) конвейеризованного цикла.

Чтобы проверить наличие в плане программной конвейеризации конфликтов, связанных с ресурсами, можно просто проверить запросы в таблице модульного резервирования ресурсов. Само собой разумеется, если запросы в устойчивом состоянии могут быть удовлетворены, то тем более они могут быть удовлетворены в прологе и эпилоге — частях кода до и после устойчивого состояния цикла. □

В общем случае для данного интервала между запусками T и таблицы резервирования ресурсов итерацией RT конвейеризованный план на машине с вектором ресурсов R не имеет конфликтов, связанных с ресурсами, тогда и только тогда, когда $RT_S[i] \leqslant R$ для всех $i = 0, 1, \ldots, T-1$.

Ограничения, связанные с зависимостями через данные

Зависимости через данные при программной конвейеризации отличаются от зависимостей, с которыми мы сталкивались до настоящего времени, потому что они могут образовывать циклы. Операция может зависеть от результата той же операции из предыдущей итерации. Теперь помечать ребра зависимостей только лишь значениями задержек недостаточно; требуется также различать различные экземпляры одной и той же операции в разных итерациях. Мы будем помечать ребро зависимости $n_1 \rightarrow n_2$ меткой $\langle\delta, d\rangle$, если операция n_2 в итерации i должна быть задержана как минимум на d тактов после выполнения операции n_1 в итерации $i - \delta$. Пусть S – план, полученный путем программной конвейеризации – представляет собой функцию, областью определения которой является множество узлов графа зависимости данных, а областью значений – целые числа, и пусть T – интервал между запусками итераций. Тогда

$$(\delta \times T) + S(n_2) - S(n_1) \geqslant d.$$

Разность итераций δ должна быть неотрицательна. Более того, для данного цикла из ребер зависимости данных как минимум одно ребро имеет положительную разность итераций.

Пример 10.18. Рассмотрим следующий цикл в предположении, что значения p и q нам неизвестны:

```
for(i = 0; i < n; i++)
    *(p++) = *(q++) + c;
```

Мы должны считать, что любая пара `*(p++)` и `*(q++)` может обращаться к одной и той же ячейке памяти. Таким образом, все чтения и записи должны выполняться в том же порядке, что и в исходной программе. Считая, что целевая машина имеет те же характеристики, что и описанная в примере 10.12, мы получим для этого кода ребра зависимостей данных, показанные на рис. 10.25. Заметим, однако, что мы игнорируем команды управления циклом, которые состоят в вычислении и тестировании значения i либо в проверке, основанной на значениях R1 или R2. □

Разность итераций между связанными операциями может быть больше единицы, что видно из следующего примера:

```
for(i = 2; i < n; i++)
    A[i] = B[i] + A[i-2];
```

Здесь значение, записываемое на итерации i, используется двумя итерациями позже. Ребро зависимости между сохранением $A[i]$ и загрузкой $A[i-2]$, таким образом, имеет разность в две итерации.

```
// R1, R2 = q, p
//R3 = c

LD R4,0(R1++)      i1
   <0,1>
ADD R5,R4,R3       i2
   <0,2>
ST 0(R2++),R5      i3

<1,1>
```

Рис. 10.25. Граф зависимости данных к примеру 10.18

Наличие циклов зависимостей данных накладывает еще одно ограничение на скорость выполнения. Например, цикл зависимости данных на рис. 10.25 приводит к задержке в четыре такта между операциями загрузки из последовательных итераций, т.е. исходный цикл не может выполняться быстрее, чем одна итерация за четыре такта.

Интервал между запусками итераций конвейеризованного цикла не может быть меньше

$$\max_{c\text{—цикл в } G} \left\lceil \frac{\sum_{e\in c} d_e}{\sum_{e\in c} \delta_e} \right\rceil$$

тактов.

В результате интервал между запусками итераций каждого программно конвейеризованного цикла ограничен использованием ресурсов на каждой итерации. А именно, этот интервал должен быть не меньше, чем отношение количества необходимых единиц каждого ресурса и количества доступных единиц ресурса в целевой машине. Кроме того, если имеются циклы зависимостей данных, то интервал между запусками оказывается еще более ограниченным суммой задержек в цикле, деленной на сумму разностей итераций. Наибольшая из этих величин определяет нижнюю границу между запусками итераций.

10.5.6 Алгоритм программной конвейеризации

Цель программной конвейеризации состоит в том, чтобы найти план с наименьшим возможным интервалом между запусками итераций. Это NP-полная задача, которая может быть переформулирована как задача целочисленного линейного программирования. Мы должны показать, что если мы знаем, чему равен минимальный интервал, то алгоритм планирования может избежать конфликта ресурсов с использованием таблиц модульного резервирования ресурсов при раз-

мещении каждой операции. Но мы не знаем, чему равен минимальный интервал, пока не разработаем план. Как же разорвать этот круг?

Мы знаем, что интервал между запусками должен быть больше, чем граница, вычисленная из требований цикла к ресурсам и циклов зависимостей данных, как рассматривалось выше. Если мы можем найти план, отвечающий этой границе, то это — оптимальный план. Если же такой план не найден, можно повторить попытку поиска с бóльшим значением интервала между запусками, и так до тех пор, пока план не будет найден. Заметим, что если используется не исчерпывающий поиск, а лишь эвристика, то оптимальный план может так и не быть найден.

Можем ли мы найти план, близкий к нижней границе, зависит от свойств графа зависимости данных и архитектуры целевой машины. Можно легко найти оптимальный план в случае ацикличного графа зависимости данных, а каждая машинная команда требует для выполнения только одной единицы одного ресурса. Легко также найти близкий к нижней границе план, когда в наличии больше аппаратных ресурсов, чем может быть использовано графом зависимости данных с циклами. В таких случаях можно рекомендовать начать с нижней границы в качестве начального значения интервала между запусками итераций, а затем увеличивать его на один такт для каждой попытки построения плана. Другая возможность заключается в поиске интервала между запусками путем бинарного поиска. В качестве верхней границы можно использовать длину плана для одной итерации, полученную путем планирования списка.

10.5.7 Планирование ациклических графов зависимости данных

Для простоты мы пока что считаем, что программно конвейеризуемый цикл содержит только один базовый блок. Это условие будет ослаблено в разделе 10.5.11.

Алгоритм 10.19. Программная конвейеризация ациклического графа зависимости данных

Вход: вектор ресурсов целевой машины $R = [r_1, r_2, \ldots]$, где r_i — количество доступных единиц ресурса i-го вида и граф зависимости данных $G = (N, E)$. Каждая операция $n \in N$ помечается ее таблицей резервирования ресурсов RT_n; каждое ребро $e = n_1 \to n_2$ из E имеет метку $\langle \delta_e, d_e \rangle$, указывающую, что операция n_2 должна выполняться не ранее чем через d_e тактов после операции n_1 из δ_e-й предшествующей итерации.

Выход: программно конвейеризованный план S и интервал между запусками итераций T.

Метод: выполнить программу, приведенную на рис. 10.26. □

Алгоритм 10.19 программно конвейеризует ациклические графы зависимостей данных. Сначала алгоритм находит границу интервала между запусками итера-

```
main (){
    T_0 = \max_j \lceil \frac{\sum_{n,i} RT_n(i,j)}{r_j} \rceil;
    for (T = T_0, T_0 + 1, ..., пока не будут спланированы все узлы из N) {
        RT = пустая таблица резервирования ресурсов с T строками;
        for (каждый узел n ∈ N в приоритетном топологическом порядке) {
            s_0 = max_{e=p→n из E} (S(p) + d_e);
            for (s = s_0, s_0 + 1, ..., s_0 + T − 1)
                if (NodeScheduled (RT, T, n, s)) break;
            if (n не может быть спланировано в RT) break;
        }
    }
}

NodeScheduled (RT, T, n, s) {
    RT' = RT;
    for (каждая строка i из RT_n)
        RT' [(s + i) mod T] = RT' [(s + i) mod T] + RT_n [i];
    if (для всех i, RT' (i) ≤ R) {
        RT = RT';
        S (n) = s;
        return true;
    }
    else return false;
}
```

Рис. 10.26. Алгоритм программной конвейеризации ациклического графа

ций T_0 на основе требований к ресурсам операций в графе. Затем он пытается найти программно конвейеризованный план, начиная с T_0 в качестве целевого интервала между запусками итераций. Алгоритм повторяется с бóльшим значением интервала, если для текущего значения построить план не удается.

При каждой попытке алгоритм применяет подход из алгоритма планирования списка. Этот подход использует модульное резервирование ресурсов для отслеживания запросов ресурсов в устойчивом состоянии. Операции планируются в топологическом порядке, так что зависимости данных всегда можно удовлетворить соответствующими задержками операций. Для планирования операции мы сначала находим нижнюю границу s_0, соответствующую ограничениям зависимостей данных. Затем вызывается функция *NodeScheduled*, проверяющая возможные конфликты ресурсов в устойчивом состоянии. Если конфликт существует, алгоритм пытается спланировать операцию на следующий такт. Если выясняется, что опе-

рация вызывает конфликт для T последовательных тактов, то в силу модульной природы обнаружения конфликтов, связанных с ресурсами, последующие попытки будут гарантированно неуспешными, так что следует переходить к другому значению интервала между запусками итераций.

Эвристика, состоящая в планировании операций на как можно более ранние моменты времени, стремится минимизировать длину плана итерации. Однако планирование операции на как можно более ранний срок может привести к росту времени жизни некоторых переменных. Например, загрузка данных при этом может оказаться выполненной задолго до того, как потребуются загружаемые данные. Одна из простейших эвристик состоит в планировании графа зависимости данных в обратном направлении, поскольку обычно загрузок существенно больше, чем сохранений.

10.5.8 Планирование графов с циклическими зависимостями

Наличие циклов зависимостей существенно усложняет программную конвейеризацию. При планировании операций в ациклическом графе в топологическом порядке зависимости данных планируемых операций могут указать только нижнюю границу размещения каждой операции. В результате всегда можно удовлетворить ограничения, связанные с зависимостями данных, путем задержки операций. Но концепция "топологического порядка" не применима к циклическим графам. В действительности для данной пары операций в цикле размещение одной из них указывает как нижнюю, так и верхнюю границы для размещения второй.

Пусть n_1 и n_2 — две операции из цикла зависимостей, S — план, получаемый путем программной конвейеризации, а T — интервал между запусками итераций для данного плана. Ребро зависимости $n_1 \to n_2$ с меткой $\langle \delta_1, d_1 \rangle$ накладывает на $S(n_1)$ и $S(n_2)$ следующие ограничения:

$$(\delta_1 \times T) + S(n_2) - S(n_1) \geqslant d_1$$

Аналогично ребро зависимости $n_2 \to n_1$ с меткой $\langle \delta_2, d_2 \rangle$ накладывает ограничение

$$(\delta_2 \times T) + S(n_1) - S(n_2) \geqslant d_2$$

Таким образом,

$$S(n_1) + d_1 - (\delta_1 \times T) \leqslant S(n_2) \leqslant S(n_1) - d_2 + (\delta_2 \times T)$$

Сильно связанным компонентом (strongly connected component) графа называется множество узлов, в котором каждый узел компонента может быть достигнут

из любого другого узла компонента. Планирование одного узла в сильно связанном компоненте ограничивает время каждого другого узла компонента как снизу, так и сверху. Транзитивно, если существует путь p, ведущий из n_1 в n_2, то

$$S(n_2) - S(n_1) \geqslant \sum_{e \in p} (d_e - (\delta_e \times T)) \tag{10.1}$$

Заметим следующее.

- Сумма δ вдоль цикла должна быть положительной. Если бы она была равна 0 или отрицательна, то это означало бы, что либо операция в цикле должна предшествовать самой себе, либо все операции цикла должны выполняться одновременно, в один и тот же такт.
- План для операций внутри итерации одинаков для всех итераций; это требование, по сути, и означает "программную конвейеризацию". В результате сумма задержек (вторых компонентов меток ребер в графе зависимости данных) в цикле представляет собой нижнюю границу интервала между запусками итераций T.

Комбинируя эти два наблюдения, можно увидеть, что для любого допустимого интервала между запусками итераций T значение правой части уравнения (10.1) должно быть отрицательно или равно 0, если p представляет собой цикл. В результате наиболее строгие ограничения на размещение узлов получаются из *простых* путей, т.е. из путей, не содержащих циклы.

Таким образом, для каждого допустимого T вычисление транзитивного влияния зависимостей данных на каждую пару узлов эквивалентно поиску длины наибольшего простого пути от первого узла ко второму. Более того, поскольку циклы не могут увеличивать длину пути, для поиска наидлиннейших путей без требования "простоты пути" можно воспользоваться простым алгоритмом динамического программирования и быть уверенным, что полученные в результате длины будут также длинами наидлиннейших простых путей (см. упражнение 10.5.7).

Пример 10.20. На рис. 10.27 показан граф зависимости данных с четырьмя узлами: a, b, c и d. Возле каждого узла изображена его таблица резервирования ресурсов, а возле каждого ребра — разность итераций и задержка. Будем считать, что в этом примере целевая машина имеет по одной единице каждого ресурса. Поскольку имеется три использования первого ресурса и два — второго, интервал между запусками итераций не может быть меньше трех тактов. В данном графе имеется два сильно связанных компонента: первый из них тривиальный, состоящий из единственного узла a, а второй — из узлов b, c и d. Самый длинный цикл, $b - c - d - b$, имеет общую задержку, равную трем тактам в пределах одной итерации. Таким образом, нижняя граница интервала между запусками итераций, основанная на ограничениях цикла зависимости данных, также равна трем тактам.

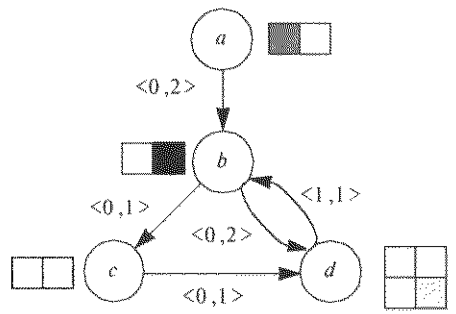


Рис. 10.27. Граф зависимости данных и требования к ресурсам из примера 10.20

Размещение любого из узлов b, c или d в плане накладывает ограничения на все остальные узлы компонента. Пусть T — интервал между запусками итераций. На рис. 10.28 показаны транзитивные зависимости. В части *а* показаны задержки и разности итераций δ для каждого из ребер. Задержка указана непосредственно, в то время как δ представлена путем "добавления" к задержке значения $-\delta T$.

	a	b	c	d
a		2		
b			1	2
c				1
d		$1-T$		

а) Исходные ребра

	a	b	c	d
a		2	3	4
b			1	2
c		$2-T$		1
d		$1-T$	$2-T$	

б) Длиннейший простой путь

	a	b	c	d
a		2	3	4
b			1	2
c		-1		1
d		-2	-1	

в) Длиннейший простой путь ($T = 3$)

	a	b	c	d
a		2	3	4
b			1	2
c		-2		1
d		-3	-2	

г) Длиннейший простой путь ($T = 4$)

Рис. 10.28. Транзитивные зависимости в примере 10.20

На рис. 10.28, *б* показана длина самого длинного пути между двумя узлами (если такой путь существует). Записями в этой таблице являются суммы выра-

жений из таблицы на рис. 10.28, *а* для каждого из ребер вдоль рассматриваемого пути. Далее, на рис. 10.28, *в* и *г* мы видим выражения из рис. 10.28, *б* с двумя подходящими значениями T, равными 3 и 4. Разность между планами для двух узлов $S(n_2) - S(n_1)$ должна быть не меньше, чем значение в ячейке (n_1, n_2) в таблицах *в* и *г* (в зависимости от выбранного значения T).

Например, рассмотрим запись $2 - T$ на рис. 10.28 для наидлиннейшего (простого) пути от c к b. Наидлиннейший простой путь от c к b — $c \to d \to b$. Общая задержка вдоль этого пути — два такта, а сумма значений δ — 1, которая представляет тот факт, что номер итерации должен увеличиться на 1. Поскольку T — продолжительность промежутка между двумя итерациями, узел b должен быть запланирован на такт, находящийся как минимум через $2 - T$ такта *после* такта, на который запланирован узел c. Но, поскольку T не меньше 3, это означает, что на самом деле b может быть спланирован за $T - 2$ такта *до* c или позже этого такта, но никак не ранее.

Заметим, что рассмотрение непростых путей от c к b не дает более строгих ограничений. Можно добавить к пути $c \to d \to b$ любое количество итераций цикла из узлов b и d. Если мы добавим k таких циклов, то получим длину пути $2 - T + k(3 - T)$, поскольку общая задержка вдоль пути равна 3, а сумма значений δ равна 1. Поскольку $T \geqslant 3$, эта длина никогда не превышает $2 - T$; таким образом, наиболее сильная нижняя граница такта b по отношению к такту c равна $2 - T$ — тому же значению, которое было получено при рассмотрении наидлиннейшего простого пути.

Например, из записей (b, c) и (c, b) мы видим, что

$$S(c) - S(b) \geqslant 1$$
$$S(b) - S(c) \geqslant 2 - T$$

Иначе говоря,

$$S(b) + 1 \leqslant S(c) \leqslant S(b) - 2 + T$$

Если $T = 3$, то

$$S(b) + 1 \leqslant S(c) \leqslant S(b) + 1$$

Это означает, что узел c должен быть запланирован на следующий после b такт. Если же $T = 4$, то

$$S(b) + 1 \leqslant S(c) \leqslant S(b) + 2,$$

так что узел c должен быть запланирован на один или два такта после b.

Информация о самом длинном пути позволяет нам легко вычислить корректное место для узла, исходя из зависимостей данных. Мы видим, что у нас нет выбора при $T = 3$, но при росте T количество возможных вариантов возрастает. □

Алгоритм 10.21. Программная конвейеризация

Вход: вектор ресурсов целевой машины $R = [r_1, r_2, \ldots]$, где r_i — количество доступных единиц ресурса i-го вида и граф зависимости данных $G = (N, E)$. Каждая операция $n \in N$ помечается ее таблицей резервирования ресурсов RT_n; каждое ребро $e = n_1 \to n_2$ из E имеет метку $\langle \delta_e, d_e \rangle$, указывающую, что операция n_2 должна выполняться не ранее чем через d_e тактов после операции n_1 из δ_e-й предшествующей итерации.

Выход: программно конвейеризованный план S и интервал между запусками итераций T.

Метод: выполнить программу, приведенную на рис. 10.29. □

Алгоритм 10.21 имеет высокоуровневую структуру, аналогичную структуре алгоритма 10.19, который способен работать только с ациклическими графами. Минимальный интервал между запусками итераций в данном случае ограничен не только требованиями к ресурсам, но и циклами зависимостей данных в графе. Планирование графа выполняется путем поочередного планирования его сильно связанных компонентов. Если рассматривать каждый сильно связанный компонент как отдельный модуль, то ребра между ними обязательно образуют ациклический граф. Если алгоритм 10.19 планирует узлы графа в топологическом порядке, то алгоритм 10.21 планирует в топологическом порядке сильно связанные компоненты. Как и ранее, если алгоритм не в состоянии спланировать все компоненты, то интервал между запусками итераций увеличивается и попытка планирования повторяется. Заметим, что в случае ациклического графа алгоритм 10.21 ведет себя точно так же, как и алгоритм 10.19.

Алгоритм 10.21 вычисляет два дополнительных множества ребер: E' — множество всех ребер, разность итераций которых равна 0, и E^* — ребра наидлиннейшего пути через все точки. Иначе говоря, для каждой пары узлов (n, p) существует ребро $e \in E^*$, с которым связано значение d_e, равное длине наидлиннейшего простого пути от p к n, что обеспечивает существование как минимум одного пути от p к n. E^* вычисляется для каждого значения интервала между запусками итераций T. Можно также выполнить это вычисление однократно с использованием символического значения T, а затем для каждой итерации подставлять вместо T конкретные значения, как это было сделано в примере 10.20.

Алгоритм 10.21 использует возвраты. Если он не в состоянии спланировать сильно связанный компонент, то он пытается спланировать весь компонент на такт позже. Эти попытки продолжаются до достижения T тактов. Возврат важен в связи с тем, что, как показано в примере 10.20, размещение первого узла сильно связанного компонента может полностью определять план для других узлов. Если такой план не согласуется с планом, уже разработанным к этому моменту, попытка составления плана оказывается неудачной.

main (){
$E' = \{e | e \in E,\ \delta_e = 0\}$;
$T_0 = \max\left(\max_j \left\lceil \frac{\sum_{n,i} RT_n(i,j)}{r_j} \right\rceil, \max_{c\text{—цикл в } G} \left\lceil \frac{\sum_{e \in c} d_e}{\sum_{e \in c} \delta_e} \right\rceil\right)$;
for ($T = T_0, T_0 + 1, \ldots$ или пока все сильно связанные компоненты
в G не будут спланированы) {
RT = пустая таблица резервирования с T строками;
E^* = *AllPairsLongestPath* (G, T);
for (каждый сильно связанный компонент C из G
в приоритетном топологическом порядке) {
for (все n из C)
$s_0 = \max_{e=p\to n \text{ из } E^*, \text{узел } p \text{ спланирован}} (S(p) + d_e)$;
first = некоторое n, такое, что $s_0(n)$ является минимумом;
$s_0 = s_0(\text{first})$;
for ($s = s_0$; $s < s_0 + T$; $s = s + 1$)
if (*SccScheduled* $(RT, T, C, first, s)$) **break**;
if (C не может быть спланирован в RT) **break**;
}
}
}

SccScheduled $(RT, T, C, first, s)$ {
$RT' = RT$;
if (not *NodeScheduled* $(RT', T, first, s)$) **return false**;
for (каждый остающийся n из с в приоритетном
топологическом порядке ребер из E') {
$s_l = \max_{e=n'\to n \text{ из } E^*, n' \in c, \text{узел } n' \text{ спланирован}} (S(n') + d_e - (\delta_c \times T))$;
$s_u = \min_{e=n'\to n \text{ из } E^*, n' \in c, \text{узел } n' \text{ спланирован}} (S(n') - d_e + (\delta_e \times T))$;
for ($s = s_l$; $s \leqslant \min(s_u, s_l + T - 1)$; $s = s + 1$)
if (*NodeScheduled* (RT', T, n, s)) **break**;
if (n не может быть спланирован в RT') **return false**;
}
$RT = RT'$;
return true;
}

Рис. 10.29. Алгоритм программной конвейеризации графа зависимости данных с циклами

Для планирования сильно связанного компонента алгоритм определяет самый ранний момент, когда каждый узел компонента может быть спланирован с удовлетворением всех транзитивных зависимостей данных из E^*. Затем в качестве пер-

вого планируемого узла *first* выбирается узел с наиболее ранним временем начала. Далее алгоритм использует функцию *SccScheduled*, которая пытается спланировать компонент с использованием наиболее раннего времени начала выполнения. Алгоритм делает не более T попыток с последовательно возрастающим временем начала. Если попытка оказывается неудачной, алгоритм пытается использовать другой интервал между запусками итераций.

Алгоритм *SccScheduled* напоминает алгоритм 10.19, но имеет три существенных отличия.

1. Цель *SccScheduled* состоит в планировании сильно связанного компонента в данный интервал времени s. Если узел *first* из сильно связанного компонента не может быть спланирован в интервале s, функция *SccScheduled* возвращает значение **false**. В таком случае функция *main* может, если это потребуется, вновь вызвать *SccScheduled* с более поздним интервалом времени.

2. Узлы сильно связанного компонента планируются в топологическом порядке, основанном на ребрах из E'. Поскольку разность итераций для всех ребер из E' равна 0, эти ребра не пересекают никакие границы итераций и не могут образовывать циклы. (Ребра, пересекающие границы итераций, известны как *циклонесущие* (loop carried).) Верхнюю границу размещения операций указывают только циклонесущие ребра. Итак, данный порядок планирования вкупе со стратегией по возможности наиболее раннего планирования каждой операции максимизируют диапазоны, в пределах которых могут планироваться последовательные операции.

3. Для сильно связанных компонентов зависимости указывают как нижнюю, так и верхнюю границы диапазона, в котором может планироваться узел. Функция *SccScheduled* вычисляет эти диапазоны и использует их для дальнейшего ограничения попыток планирования.

Пример 10.22. Применим алгоритм 10.21 к циклическому графу зависимости данных из примера 10.20. Сначала алгоритм вычисляет, что в данном примере граница интервала между запусками итераций равна трем тактам. Заметим, что достичь этой нижней границы невозможно. Когда интервал между запусками итераций T равен трем, транзитивные зависимости на рис. 10.28 требуют выполнения условия $S(d) - S(b) = 2$. Планирование узлов b и d на расстоянии двух тактов друг от друга приводит к конфликту в таблице модульного резервирования ресурсов длиной 3.

На рис. 10.30 показано поведение алгоритма 10.21 с графом из упомянутого примера. Сначала он пытается найти план для интервала между запусками итераций, равного 3. Попытка начинается с планирования узлов a и b в наиболее

Попытка	Интервал между запусками итераций	Узел	Диапазон	План	Модульное резервирование ресурсов
1	$T = 3$	a	$(0, \infty)$	0	
		b	$(2, \infty)$	2	
		c	$(3, 3)$	— —	
2	$T = 3$	a	$(0, \infty)$	0	
		b	$(2, \infty)$	3	
		c	$(4, 4)$	4	
		d	$(5, 5)$	— —	
3	$T = 3$	a	$(0, \infty)$	0	
		b	$(2, \infty)$	4	
		c	$(5, 5)$	5	
		d	$(6, 6)$	— —	
4	$T = 4$	a	$(0, \infty)$	0	
		b	$(2, \infty)$	2	
		c	$(3, 4)$	3	
		d	$(4, 5)$	— —	
5	$T = 4$	a	$(0, \infty)$	0	
		b	$(2, \infty)$	3	
		c	$(4, 5)$	5	
		d	$(5, 5)$	— —	
6	$T = 4$	a	$(0, \infty)$	0	
		b	$(2, \infty)$	4	
		c	$(5, 6)$	5	
		d	$(6, 7)$	6	

Рис. 10.30. Поведение алгоритма 10.21 с графом из примера 10.20

ранние допустимые моменты времени. Однако после размещения узла b в такте 2 узел c может быть помещен только в такт 3, а это приводит к конфликту с использованием ресурсов узлом a. И a, и c требуют себе первый ресурс в те такты, которые при делении на 3 дают остаток 0.

Алгоритм выполняет возврат и пытается выполнить планирование сильно связанного компонента $\{b, c, d\}$ на такт позже. В этот раз узел b размещается в такте 3, а узел c успешно размещается в такте 4. Однако узел d не может быть размещен в такте 5. И b, и d требуют себе второй ресурс в те такты, которые при делении

на 3 дают остаток 0. Заметим, что это просто совпадение, что и этот конфликт, и ранее рассматривавшийся проявляются в тактах, которые при делении на 3 дают остаток 0. С тем же успехом в других примерах конфликт мог бы проявиться и в тактах, дающих остаток 1 или 2.

Алгоритм продолжает работу, откладывая начало планирования сильно связанного компонента $\{b, c, d\}$ еще на один такт. Но, как уже говорилось, данный сильно связанный компонент не может быть спланирован с интервалом между запусками итераций, равным 3. Так что алгоритм прекращает попытки планирования с интервалом 3 и приступает к попыткам планирования с интервалом 4, и в конечном итоге находит оптимальный план во время шестой попытки. □

10.5.9 Усовершенствования алгоритма конвейеризации

Алгоритм 10.21 достаточно простой, хотя и неплохо работает на реальных целевых машинах. Важными элементами этого алгоритма являются следующие.

1. Использование таблицы модульного резервирования ресурсов для обнаружения конфликтов в устойчивом состоянии.
2. Необходимость вычисления транзитивных отношений зависимости для поиска диапазона планирования узла при наличии циклов зависимостей.
3. Возможность возврата, а также совместного перепланирования узлов *критических циклов* (циклов, которые определяют наибольшую нижнюю границу интервала между запусками итераций T), поскольку между ними нет временны́х зазоров.

Имеется ряд способов усовершенствования алгоритма 10.21. Например, в простом примере 10.22 алгоритм тратит некоторое время на то, чтобы убедиться, что интервал между запусками итераций, равный трем тактам, неприемлем. Можно сначала независимо спланировать сильно связанные компоненты для того, чтобы выяснить пригодность интервала между запусками итераций для каждого компонента.

Можно также изменить порядок планирования узлов. Порядок, использованный в алгоритме 10.21, имеет несколько недостатков. Во-первых, поскольку нетривиальные сильно связанные компоненты планируются сложнее тривиальных, их желательно планировать первыми. Во-вторых, некоторые регистры могут иметь неоправданно длительное время жизни. Желательно размещать определения поближе к соответствующим использованиям. Одна из возможностей заключается в том, чтобы начинать работу с сильно связанных компонентов с критическими циклами, а затем с двух сторон расширять план.

Есть ли альтернативы эвристикам

Задачу одновременного поиска оптимального плана и распределения регистров можно сформулировать в виде задачи целочисленного линейного программирования. При том что многие задачи целочисленного линейного программирования могут быть решены достаточно быстро, некоторые из них требуют для решения непомерного количества времени. Чтобы реально использовать целочисленное линейное программирование, необходимо иметь возможность прервать вычисления, если они не укладываются в некоторые предопределенные пределы.

Такой подход был эмпирически испытан на целевой машине (SGI R8000), и было обнаружено, что для большой части программ удавалось за приемлемое время найти оптимальное решение поставленной задачи. Оказалось также, что планы, полученные с применением эвристического подхода, достаточно близки к оптимальным. Таким образом, выяснилось, что использовать подход целочисленного линейного программирования не имеет особого смысла. Тем более что в связи с тем, что решение задачи линейного программирования может не быть получено за предопределенное время, в компиляторе все равно требуется иметь эвристический планировщик. Однако решения, полученные с его помощью, отличаются от оптимальных в столь незначительной степени, что при наличии эвристического планировщика просто нет смысла в реализации еще и планировщика на основе целочисленного линейного программирования.

10.5.10 Модульное расширение переменных

Скалярная переменная называется *приватизируемой* (privatizable) в цикле, если время ее жизни не выступает за пределы итерации. Другими словами, приватизируемая переменная не должна быть активна до входа в любую итерацию или после выхода из нее. Название таких переменных связано с тем, что различные процессоры, выполняя различные итерации цикла, могут иметь собственные частные копии переменной, таким образом, никак не влияя друг на друга.

Расширение переменной (variable expansion) означает преобразование по превращению приватизируемой скалярной переменной в массив, такой, что i-я итерация цикла читает и записывает i-й элемент этого массива. Такое преобразование устраняет ограничения антизависимости между чтениями в одной итерации и записями в последующих, как и выходные зависимости между записями в разных итерациях. Если все циклонесущие зависимости оказываются устраненными, то все итерации цикла могут выполняться параллельно.

Устранение циклонесущих зависимостей и, таким образом, устранение циклов в графе зависимости данных может существенно повысить эффективность программной конвейеризации. Как показано в примере 10.15, нам не требуется расширение приватизируемой переменной на количество итераций цикла. Одновременно может выполняться только небольшое количество итераций, а приватизируемые переменные могут быть одновременно активны даже в меньшем количестве итераций. Одна и та же память может использоваться для хранения переменных с неперекрывающимися временами жизни. Говоря конкретнее, если время жизни регистра равно l тактам, а интервал между запусками итераций равен T, то в любой одной точке могут быть активны только $q = \lceil l/T \rceil$ значений. Мы можем выделить для переменной q регистров, используя в i-й итерации $(i \bmod q)$-й регистр. Такое преобразование называется *модульным расширением переменной* (modular variable expansion).

Алгоритм 10.23. Программная конвейеризация с модульным расширением переменной

Вход: граф зависимости данных и описание ресурсов машины.

Выход: два цикла: один — программно конвейеризованный, а второй — неконвейеризованный.

Метод: выполнить следующие действия.

1. Устранить циклонесущие антизависимости и выходные зависимости, связанные с приватизируемыми переменными из графа зависимости данных.

2. Программно конвейеризовать получающийся в результате граф зависимостей с использованием алгоритма 10.21. Пусть T — интервал между запусками итераций, для которого найден план, а L — длина плана одной итерации.

3. Вычислить для получившегося плана q_v минимальное количество регистров, необходимых для каждой приватизируемой переменной v. Пусть $Q = \max_v q_v$.

4. Сгенерировать два цикла: программно конвейеризованный и неконвейеризованный. Программно конвейеризованный цикл содержит $\lceil L/T \rceil + Q - 1$ копий итераций, размещенных на расстоянии T тактов друг от друга. Его пролог состоит из $(\lceil L/T \rceil - 1)T$ команд, устойчивое состояние — из QT команд, а эпилог — из $L - T$ команд. Вставить команду цикла, которая выполняет ветвление из конца устойчивого состояния в его начало.

   Количество регистров, назначенных приватизируемой переменной v, равно

$$q'_v = \begin{cases} q_v, & \text{если } Q \bmod q_v = 0, \\ Q & \text{в противном случае.} \end{cases}$$

Переменная v в i-й итерации использует $(i \bmod q'_i)$-й из назначенных ей регистров.

Пусть n — переменная, представляющая количество итераций исходного цикла. Программно конвейеризованный цикл выполняется, если $n \geqslant \lceil L/T \rceil + Q - 1$. Количество выполнений вставленной команды цикла равно

$$n_1 = \left\lfloor \frac{n - \lceil L/T \rceil + 1}{Q} \right\rfloor$$

Таким образом, количество исходных итераций, выполняемых программно конвейеризованным циклом, равно

$$n_2 = \begin{cases} \lceil L/T \rceil - 1 + Qn_1, & \text{если } n \geqslant \lceil L/T \rceil + Q - 1, \\ 0 & \text{в противном случае.} \end{cases}$$

Количество итераций, выполняемых неконвейеризованным циклом, равно $n_3 = n - n_2$. □

Пример 10.24. В случае программно конвейеризованного цикла на рис. 10.22 $L = 8$, $T = 2$ и $Q = 2$. Программно конвейеризованный цикл содержит 7 копий итераций с прологом, устойчивым состоянием и эпилогом, состоящими соответственно из 6, 4 и 6 команд. Пусть n — количество итераций в исходном цикле. Программно конвейеризованный цикл выполняется, если $n \geqslant 5$; в этом случае команда цикла выполняется

$$\left\lfloor \frac{n-3}{2} \right\rfloor$$

раз, а программно конвейеризованный цикл отвечает за

$$3 + 2 \times \left\lfloor \frac{n-3}{2} \right\rfloor$$

итераций исходного цикла. □

Модульное расширение увеличивает размер устойчивого состояния в Q раз. Несмотря на это увеличение код, сгенерированный алгоритмом 10.23, остается достаточно компактным. В наихудшем случае программно конвейеризованный цикл будет содержать в три раза больше команд, чем спланировано для одной итерации. Грубо говоря, вместе с дополнительным циклом для обработки остающихся операций общий размер кода вырастает примерно в 4 раза по сравнению с первоначальным. Поскольку данный метод обычно применяется для небольших внутренних циклов, такое увеличение оказывается вполне приемлемым.

Алгоритм 10.23 минимизирует увеличение кода ценой использования большего количества регистров. Можно уменьшить количество используемых регистров,

генерируя менее компактный код. Минимальное количество используемых регистров q_v для каждой переменной v достигается при устойчивом состоянии из $T \times \text{LCM}_v q_v$ команд. Здесь LCM_v означает *наименьшее общее кратное* (least common multiple) всех q_v, где индекс v пробегает по всем приватизируемым переменным (т.е. нас интересует наименьшее целое число, которое кратно всем q_v). К сожалению, наименьшее общее кратное может оказаться достаточно большим даже при малых значениях q_v.

10.5.11 Условные инструкции

Если доступны предикатные команды, то можно преобразовать команды, зависящие от управления, в предикатные. Предикатные команды могут быть программно конвейеризованы так же, как и любые другие операции. Однако если в теле цикла содержится большое количество потока управления, зависящего от данных, то более подходящими могут оказаться методы планирования, описанные в разделе 10.4.

Если машина не оснащена предикатными командами, то для обработки небольших потоков управления, зависящих от данных, можно воспользоваться описываемой ниже концепцией *иерархического приведения* (hierarchical reduction). Подобно алгоритму 10.11 при иерархическом приведении управляющие конструкции в цикле планируются изнутри наружу, начиная с наиболее глубоко вложенных структур. При планировании каждая конструкция приводится к единому узлу, представляющему все ограничения, накладываемые на планирование ее компонентов со стороны других частей программы. Такой узел затем можно планировать так, как если бы это был простой узел в окружающей управляющей конструкции. Процесс планирования завершается, когда к единственному узлу приводится вся программа.

В случае условных инструкций с ветвями "then" и "else" каждая из ветвей планируется независимо.

1. Ограничения всей условной инструкции консервативно рассматриваются как объединение ограничений каждой из ветвей.

2. В качестве использования ресурсов берется максимальное из использований ресурсов в ветвях.

3. Ограничения предшествования представляют собой объединение соответствующих ограничений в каждой из ветвей, получаемое путем воображаемого выполнения обеих ветвей.

Такой узел затем планируется, как и любой другой. Генерируются два множества кода, соответствующие двум ветвям. Любой код, спланированный параллельно

с условной инструкцией, дублируется в обеих ветвях. Если перекрываются множественные условные инструкции, то для каждой комбинации выполняемых параллельно ветвей должен генерироваться отдельный код.

10.5.12 Аппаратная поддержка программной конвейеризации

Для минимизации размера программно конвейеризованного кода была предложена специализированная аппаратная поддержка. Примером может служить *блок смещающихся регистров* (rotating register file) в архитектуре Itanium. В этом блоке имеется *базовый регистр* (base register), который добавляется к номеру регистра, указанному в коде, для получения фактического регистра, к которому выполняется обращение. Различные итерации цикла могут использовать различные регистры, просто изменяя содержимое базового регистра на границе каждой итерации. Кроме того, в архитектуре Itanium имеется мощная поддержка предикатных команд. Предикаты могут использоваться не только для превращения зависимостей управления в зависимости данных, но и для избежания генерации пролога и эпилога. Тело программно конвейеризованного цикла содержит расширенный набор команд, выполняемых в прологе и эпилоге. Мы можем просто генерировать код для устойчивого состояния и использовать предикаты для получения эффекта пролога и эпилога.

При том что аппаратная поддержка повышает плотность программно конвейеризованного кода, следует отдавать себе отчет в цене такой поддержки. Поскольку программная конвейеризация — метод, предназначенный для компактных глубоко вложенных циклов, конвейеризованные циклы и так получаются небольшими. Специализированная поддержка программной конвейеризации обеспечивается, в основном, в машинах, предназначенных для выполнения большого количества программно конвейеризованных циклов и в ситуациях, когда очень важно минимизировать размер кода.

10.5.13 Упражнения к разделу 10.5

Упражнение 10.5.1. В примере 10.20 было показано, как установить относительные границы тактов, на которые планируются узлы b и c. Вычислите эти границы для каждой из пяти остальных пар узлов 1) для произвольного T; 2) для $T = 3$; 3) для $T = 4$.

Упражнение 10.5.2. На рис. 10.31 показано тело цикла. Адреса, такие как `a(R9)`, предназначены для указания ячеек памяти. Здесь a — константа, а `R9` — регистр, подсчитывающий итерации в цикле. Можно считать, что разные итерации обращаются к разным ячейкам памяти, поскольку `R9` имеет разные значения. Вос-

пользуйтесь моделью машины из примера 10.12 и спланируйте цикл на рис. 10.31 следующими способами.

а) Оставьте каждую итерацию как можно более компактной (т.е. добавьте по одной команде "нет операции" после каждой арифметической операции) и дважды разверните цикл. Спланируйте вторую итерацию так, чтобы она начиналась как можно в более ранний момент времени без нарушения ограничения, заключающегося в том, что в любой момент времени машина может выполнять одну загрузку, одно сохранение, одну арифметическую операцию и одно ветвление.

б) Повторите предыдущее задание, но с тройной разверткой цикла. Снова начинайте итерации в наиболее ранний момент времени, когда это позволяют сделать машинные ограничения.

! в) Постройте полностью конвейеризованный код с учетом машинных ограничений. В этом задании вы можете при необходимости вводить команды "нет операции", но вы должны каждые два такта начинать новую итерацию.

```
1) L:  LD  R1, a(R9)
2)     ST  b(R9), R1
3)     LD  R2, c(R9)
4)     ADD R3, R1, R2
5)     ST  c(R9), R3
6)     SUB R4, R1, R2
7)     ST  b(R9), R4
8)     BL  R9, L
```

Рис. 10.31. Машинный код к упражнению 10.5.2

Упражнение 10.5.3. Некоторый цикл требует 5 загрузок, 7 сохранений и 8 арифметических операций. Чему равен минимальный интервал между запусками итераций при программной конвейеризации этого цикла на машине, которая выполняет каждую операцию за один такт и обладает количеством ресурсов, достаточным для выполнения за один такт

а) 3 загрузок, 4 сохранений и 5 арифметических операций;

б) 3 загрузок, 3 сохранений и 3 арифметических операций.

! **Упражнение 10.5.4.** Используя модель машины из примера 10.12, найдите минимальный интервал между запусками итераций и единый план итераций для следующего цикла:

```
for (i = 1; i < n; i++) {
    A[i] = B[i-1] + 1;
    B[i] = A[i-1] + 2;
}
```

Не забывайте, что подсчет итераций выполняется при помощи автоматического увеличения значения регистра, так что никакие отдельные операции для этого не требуются.

! **Упражнение 10.5.5.** Докажите, что в частном случае, когда все операции требуют только по одной единице одного ресурса, алгоритм 10.19 всегда в состоянии построить оптимальный план, соответствующий нижней границе.

! **Упражнение 10.5.6.** Предположим, что у нас есть циклический граф зависимости данных с узлами a, b, c и d. Ребра от a к b и от c к d имеют метки $\langle 0, 1\rangle$, а ребра от b к c и от d к a — метки $\langle 1, 1\rangle$. Других ребер в графе нет.

а) Изобразите этот циклический граф зависимости данных.

б) Вычислите таблицу наидлиннейших простых путей между узлами.

в) Найдите длину наидлиннейших простых путей, если интервал между запусками итераций T равен 2.

г) Повторите предыдущее задание для значения $T = 3$.

д) Каковыми будут ограничения на относительные моменты планирования команд, представленных узлами a, b, c и d, в случае $T = 3$?

! **Упражнение 10.5.7.** Разработайте алгоритм со временем работы $O\left(n^3\right)$ для поиска длины самого длинного простого пути в графе с n узлами в предположении, что не существует циклов положительной длины. *Указание*: адаптируйте алгоритм Флойда (Floyd) для кратчайших путей (см., например, A. V. Aho and J. D. Ullman, *Foundations of Computer Science*, Computer Science Press, New York, 1992).

!! **Упражнение 10.5.8.** Предположим, что у нас есть машина с тремя типами команд, которые называются A, B и C. Все команды выполняются за один такт, и на каждом такте машина может выполнить по одной команде каждого типа. Предположим, что цикл состоит из шести команд, по две каждого типа. В таком случае можно выполнить цикл путем программной конвейеризации с интервалом между запусками итераций, равным двум. Однако некоторые последовательности шести команд требуют вставки задержки в один такт, а некоторые — в два такта. Сколько из 90 возможных последовательностей двух A, двух B и двух C не требуют задержек? Сколько требуют одной задержки? *Указание*: существует симметрия среди трех типов команд, так что две последовательности, которые могут

быть преобразованы одна в другую путем перестановки имен A, B и C, должны требовать одного и того же количества задержек. Например, последовательность $ABBCAC$ в этом смысле такая же, как и $BCCABA$.

10.6 Резюме к главе 10

- *Вопросы архитектуры.* Оптимизированное планирование кода позволяет использовать возможности современных компьютерных архитектур. Такие машины зачастую допускают конвейерное выполнение, когда одновременно несколько команд находятся на разных стадиях выполнения. Некоторые машины позволяют также одновременно начинать выполнение нескольких команд.

- *Зависимости через данные.* При планировании команд мы должны учитывать влияние команд на каждую ячейку памяти и на регистры. Истинные зависимости через данные проявляются, когда одна команда должна читать ячейку памяти после того, как другая запишет в нее информацию. Антизависимость заключается в записи после чтения, а зависимость через выход осуществляется, когда две команды записывают данные в одну и ту же ячейку памяти.

- *Устранение зависимостей.* Используя дополнительные ячейки для хранения данных, можно устранить антизависимости и зависимости через выход. Не могут быть устранены только истинные зависимости, которые должны учитываться при планировании кода.

- *Графы зависимости данных для базовых блоков.* Такие графы представляют временны́е ограничения между командами базового блока. Узлы графа соответствуют командам. Ребро от узла m к узлу n с меткой d означает, что команда m должна начаться не ранее чем через d тактов после начала команды n.

- *Приоритетный топологический порядок.* Графы зависимостей данных для базовых блоков всегда ацикличны, и обычно у такого графа имеется много вариантов топологических порядков. Для выбора предпочтительного топологического порядка данного графа может использоваться ряд эвристик, например, выбор, в первую очередь, узлов с наибольшими критическими путями.

- *Планирование списка.* Имея приоритетный топологический порядок для графа зависимости данных, мы можем рассматривать узлы этого графа

в указанном порядке. При этом планирование каждого узла в самый ранний из возможных тактов согласуется с временны́ми ограничениями, накладываемыми ребрами графа, планами всех ранее спланированных узлов и ограничениями, связанными с ресурсами машины.

- *Перемещение кода между блоками.* При определенных обстоятельствах оказывается возможным перемещение инструкций из блока, в котором они изначально размещены, в предшествующий или последующий блок. Преимущества такого перемещения в том, что в новом месте инструкция сможет выполняться параллельно с другими, что невозможно в ее исходном местоположении. Если новое и старое местоположения не связаны отношением доминирования, то вдоль некоторых путей может потребоваться вставка компенсирующего кода, чтобы гарантировать выполнение одной и той же последовательности команд независимо от потока управления.

- *Универсальные циклы.* Универсальные циклы не имеют зависимостей между итерациями, так что любые итерации могут выполняться параллельно.

- *Программная конвейеризация универсальных циклов.* Программная конвейеризация представляет собой метод использования возможностей машины по одновременному выполнению нескольких команд. Итерации цикла планируются таким образом, что они запускаются одна за другой через малые промежутки времени с возможным добавлением команд "нет операции" с целью избежать конфликтов, связанных с ресурсами машины. В результате код цикла может быть выполнен очень быстро и состоять из пролога, эпилога и небольшого внутреннего цикла.

- *Перекрестные циклы.* Большинство циклов имеют зависимости данных между итерациями. Такие циклы называются перекрестными.

- *Графы зависимостей данных для перекрестных циклов.* Для представления зависимостей между командами перекрестного цикла ребра должны помечаться парами значений: необходимой задержкой (как в случае графов, представляющих базовые блоки) и количеством итераций между двумя зависимыми командами.

- *Планирование списка для циклов.* Для планирования циклов следует для всех итераций выбрать один план, а также интервал между запусками итераций. Алгоритм включает выведение ограничений на относительные планы различных команд цикла путем поиска наидлиннейших ациклических путей между двумя узлами. Длины этих путей используют интервал между запусками итераций в качестве параметра, что позволяет определять нижнюю границу данного интервала.

10.7 Список литературы к главе 10

Желающим более глубоко ознакомиться с архитектурой и проектированием процессоров, мы рекомендуем книгу Хеннесси (Hennessy) и Паттерсона (Patterson) [5].

Концепция зависимости данных появилась в работах Кука (Kuck), Мураоки (Muraoka) и Чена (Chen) [6] и Лампорта (Lamport) [8] в контексте компиляции кода для многопроцессорных и векторных машин.

Планирование команд впервые было применено при планировании одноуровневого микрокода [2, 3, 11, 12]. Работа Фишера (Fisher) по уплотнению микрокода привела его к концепции VLIW-машины, в которой компилятор мог непосредственно управлять параллельным выполнением операций [3]. Гросс (Gross) и Хеннесси (Hennessy) [4] использовали планирование команд для обработки отложенных ветвлений в первых наборах команд MIPS RISC. Алгоритм из данной главы основан на более общем подходе Бернштейна (Bernstein) и Родега (Rodeh) [1] к планированию операций для машин с параллелизмом на уровне команд.

Основная идея, лежащая в основе программной конвейеризации, была изначально разработана Пателем (Patel) и Дэвидсоном (Davidson) [9] для планирования аппаратных конвейеров. Программная конвейеризация впервые использована Рау (Rau) и Глезером (Glaeser) [10] в компиляции для машины со специально разработанной аппаратной поддержкой программной конвейеризации. Описанный в главе алгоритм основан на работе Лама (Lam) [7], в которой специализированная аппаратная поддержка не предусматривалась.

1. Bernstein, D. and M. Rodeh, "Global instruction scheduling for superscalar machines", *Proc. ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation*, pp. 241–255.

2. Dasgupta, S., "The organization of microprogram stores", *Computing Surveys* **11**:1 (1979), pp. 39–65.

3. Fisher, J. A., "Trace scheduling: a technique for global microcode compaction", *IEEE Trans. on Computers* **C-30**:7 (1981), pp. 478–490.

4. Gross, T. R. and Hennessy, J. L., "Optimizing delayed branches", *Proc. 15th Annual Workshop on Microprogramming* (1982), pp. 114–120.

5. Hennessy, J. L. and D. A. Patterson, *Computer Architecture: A Quantitative Approach*, Third Edition, Morgan Kaufman, San Francisco, 2003.

6. Kuck, D., Y. Muraoka, and S. Chen, "On the number of operations simultaneously executable in Fortran-like programs and their resulting speedup", *IEEE Trans. on Computers* **C-21**:12 (1972), pp. 1293–1310.

7. Lam, M. S., "Software pipelining: an effective scheduling technique for VLIW machines", *Proc. ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation*, pp. 318–328.

8. Lamport, L., "The parallel execution of DO loops", *Comm. ACM* **17**:2 (1974), pp. 83–93.

9. Patel, J. H. and E. S. Davidson, "Improving the throughput of a pipeline by insertion of delays", *Proc. Third Annual Symposium on Computer Architecture* (1976), pp. 159–164.

10. Rau, B. R. and C. D. Glaeser, "Some scheduling techniques and an easily schedulable horizontal architecture for high performance scientific computing", *Proc. 14th Annual Workshop on Microprogramming* (1981), pp. 183–198.

11. Tokoro, M., E. Tamura, and T. Takizuka, "Optimization of microprograms", *IEEE Trans. on Computers* **C-30**:7 (1981), pp. 491–504.

12. Wood, G., "Global optimization of microprograms through modular control constructs", *Proc. 12th Annual Workshop in Microprogramming* (1979), pp. 1–6.

Глава 11

Оптимизация параллелизма и локальности

Из этой главы вы узнаете о том, каким образом компилятор может повысить степень параллельности вычислений и локальность численных программ с целью повышения их производительности при работе в многопроцессорных системах. Многие научные, инженерные и коммерческие приложения переполнены вычислениями, выполняемыми в цикле. Примерами могут служить метеорологические программы, программы для аэрогидродинамических расчетов, программы квантовой хромодинамики для изучения сильных взаимодействий в физике высоких энергий и многие другие.

Одним из способов ускорения вычислений является их параллельность. К сожалению, не так легко разработать программное обеспечение, которое будет использовать преимущества параллельной работы на нескольких машинах. Разделение вычислений на отдельные модули, которые могут выполняться параллельно на нескольких процессорах, – достаточно сложная задача; к тому же такое разделение само по себе не гарантирует ускорение вычислений. Следует также минимизировать межпроцессорное взаимодействие, поскольку связанные с ним накладные расходы легко могут сделать выполняющийся параллельно код более медленным, чем обычный последовательный.

Минимизация взаимодействия может рассматриваться как частный случай повышения *локальности данных* (data locality) программы. В общем случае мы говорим о хорошей локальности данных программы, если чаще всего процессор обращается к тем же данным, к которым он уже обращался в последнее время. Само собой разумеется, если процессор на работающей параллельно машине сталкивается с высокой локальностью, у него не возникает необходимости в частом общении с другими процессорами. Таким образом, параллельность и локальность должны рассматриваться вместе. Локальность данных важна для производительности отдельного процессора и сама по себе. Современные процессоры оснащены одним или несколькими уровнями кэшей в иерархии памяти, при этом обращение к основной памяти может занимать десятки тактов, в то время как обращение к кэшу выполняется за несколько тактов. Если программа имеет плохую локальность и часто сталкивается с промахами кэша, ее производительность будет снижена.

Еще одна причина, по которой параллельность и локальность рассматриваются вместе, в одной главе, заключается в том, что они используют одну и ту же теорию. Если мы знаем, как оптимизировать программу для повышения локальности данных, мы также знаем, как поднять степень ее параллелизма. В этой главе вы увидите, что программная модель, использовавшаяся нами при анализе потоков данных в главе 9, неадекватна в случае оптимизации параллелизма и локальности. Причина этого в том, что при работе с потоками данных считалось, что мы не различаем, каким именно образом достигнута данная инструкция; рассматривавшиеся в главе 9 методы использовали тот факт, что разные выполнения одной и той же инструкции неразличимы, например, при выполнении цикла. Для распараллеливания кода мы должны рассматривать зависимости между различными динамическими выполнениями одних и тех же инструкций, чтобы определить, могут ли они выполняться одновременно разными процессорами.

В этой главе внимание сосредоточено на методах оптимизации класса численных приложений, которые работают с массивами и обращаются к ним с использованием простых регулярных шаблонов. Говоря более конкретно, мы будем рассматривать программы, обращения к массивам в которых *аффинны* (affine) по отношению к индексам охватывающих циклов. Например, если i и j — индексные переменные охватывающих циклов, то $Z[i][j]$ и $Z[i][i+j]$ — аффинные обращения. Функция от одной или нескольких переменных $x_1, x_2, \ldots, x_n$ называется *аффинной*, если она может быть выражена как сумма константы и константных множителей, умноженных на переменные, т.е. как $c_0 + c_1x_1 + c_2x_2 + \cdots + c_nx_n$, где $c_0, c_1, \ldots, c_n$ — константы. Аффинные функции обычно известны как линейные, хотя, строго говоря, линейная функция не должна иметь член c_0.

Вот простой пример соответствующего цикла:

```
for(i = 0; i < 10; i++) {
    Z[i] = 0;
}
```

Поскольку итерации цикла записывают разные ячейки памяти, разные итерации могут одновременно выполняться разными процессорами. С другой стороны, если при этом выполнялась бы другая инструкция, скажем, `Z[j] = 1`, то надо было бы отслеживать, не может ли i быть равной j, и, если может, выяснять, в каком порядке следует выполнять экземпляры этих двух инструкций, которые работают с одним и тем же значением индекса массива.

Знать, какие итерации могут обращаться к одним и тем же ячейкам памяти, крайне важно. Это знание позволяет определить зависимости данных, которые следует учитывать при планировании кода как для однопроцессорных, так и для многопроцессорных систем. Наша цель заключается в том, чтобы найти план, который учитывает все зависимости данных, так что операции, обращающиеся к одной и той же ячейке памяти или строке кэша, выполняются по возможности

поближе друг к другу, причем в многопроцессорной системе — на одном и том же процессоре.

Представленная в этой главе теория основана на линейной алгебре и методах целочисленного программирования. Мы моделируем итерации в цикле глубиной вложенности n как n-мерный многогранник, границы которого определяются границами цикла в коде. Аффинные функции отображают каждую итерацию на массив ячеек памяти, к которым она обращается. Для определения наличия двух итераций, обращающихся к одной и той же ячейке памяти, можно использовать целочисленное линейное программирование.

Множество преобразований кода, рассматриваемое здесь, подразделяется на две категории: *аффинное разбиение* (affine partitioning) и *блокирование* (blocking). Аффинное разбиение разделяет многогранник итераций на компоненты, выполняемые либо различными машинами, либо последовательно. Блокирование, с другой стороны, создает иерархию итераций. Предположим, у нас имеется цикл, построчно сканирующий массив. Вместо этого можно разделить массив на блоки и посетить все элементы блока перед переходом к следующему блоку. Получающийся в результате код будет состоять из внешних циклов, обходящих блоки, и внутренних, сканирующих элементы в пределах каждого блока. Для определения как наилучшего аффинного разбиения, так и наилучшей схемы блоков используются методы линейной алгебры.

Мы начнем с обзора концепций оптимизации параллельных вычислений и локальности в разделе 11.1. Затем в разделе 11.2 теоретические концепции будут дополнены конкретным примером — умножением матриц, — который покажет, как *преобразование цикла* (loop transformation), изменяющее порядок вычисления в цикле, может повысить локальность и эффективность распараллеливания.

В разделах 11.3–11.6 представлена предварительная информация, необходимая для выполнения преобразований циклов. В разделе 11.3 показана модель отдельной итерации цикла, в разделе 11.4 — модель функций индексов массива, которые отображают каждую итерацию цикла на ячейки массива, к которым обращается эта итерация. В разделе 11.5 рассматривается, как с помощью алгоритмов линейной алгебры определить, какие итерации цикла обращаются к одной и той же ячейке цикла или строке кэша, а в разделе 11.6 — как найти все зависимости данных среди обращений к массиву в программе.

В оставшейся части главы эта информация применяется для выполнения оптимизации. В разделе 11.7 сначала рассматривается более простая задача поиска параллельности, не требующей синхронизации. Для получения наилучшего аффинного разбиения мы просто находим решение с ограничением, заключающимся в том, что операции с зависимостями данных назначаются одному и тому же процессору.

Однако имеется не так уж много программ, которые могут быть распараллелены без необходимости синхронизации. Поэтому в разделах 11.8–11.9.9 мы рас-

сматриваем общий случай поиска параллельности, требующей синхронизации. Мы вводим концепцию конвейеризации, показывающую, как найти аффинное разбиение, максимизирующее степень допустимой конвейеризации программы. Как выполнить оптимизацию локальности, будет рассмотрено в разделе 11.10. И наконец, мы рассмотрим, как аффинные преобразования могут пригодиться для оптимизации других видов параллелизма.

11.1 Фундаментальные концепции

В этом разделе вводятся фундаментальные концепции, связанные с оптимизацией параллелизма и локальности. Если операции могут выполняться параллельно, то они могут быть переупорядочены для других целей, например для повышения локальности. И наоборот, если зависимости данных в программе диктуют последовательное выполнение команд, то речь не идет ни о параллельности, ни о возможности переупорядочения команд для повышения локальности. Таким образом, анализ параллелизма одновременно находит возможности для повышающих локальность перемещений кода.

Для минимизации взаимосвязей в параллельном коде мы группируем все связанные операции и назначаем их одному процессору. Получающийся в результате код должен обладать локальностью данных. Один грубый подход для получения хорошей локальности данных в однопроцессорной системе заключается в том, чтобы процессор поочередно выполнял код, предназначенный для разных процессоров.

Этот вводный раздел мы начнем с обзора параллельных архитектур компьютеров. Затем мы рассмотрим базовые концепции распараллеливания и использование подобных соображений для оптимизации локальности. И наконец, мы неформально познакомимся с математическими концепциями, используемыми в данной главе.

11.1.1 Многопроцессорность

Наиболее популярной параллельной архитектурой является симметричная мультипроцессорность (symmetric multiprocessor — SMP). Высокопроизводительные персональные компьютеры зачастую оснащены двумя процессорами, а многие серверные машины имеют четыре, восемь, а иногда даже десятки процессоров. Еще большее распространение мультипроцессорные системы получили с возможностью компоновки нескольких высокопроизводительных процессоров в одной микросхеме.

Процессоры в симметричной мультипроцессорной системе используют одно адресное пространство. Для взаимодействия друг с другом они просто пишут информацию в ячейки памяти, которая затем считывается другим процессором.

Симметричная многопроцессорность названа так в связи с тем, что все процессоры могут обращаться ко всей памяти в системе с одинаковым временем доступа. На рис. 11.1 показана высокоуровневая архитектура многопроцессорной системы. Процессоры могут иметь свои кэши первого, второго, а иногда даже третьего уровней. Кэши наивысшего уровня подключаются к физической памяти обычно через общую шину.

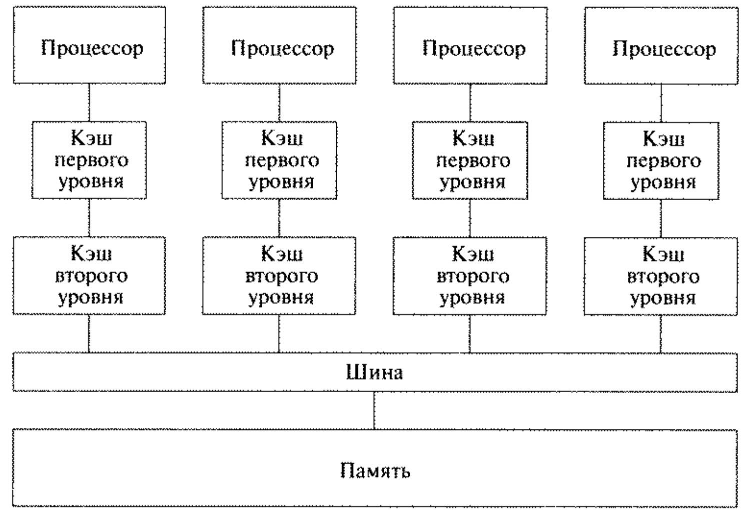


Рис. 11.1. Симметричная многопроцессорная архитектура

Симметричные многопроцессорные системы используют *протокол когерентной кэш-памяти* (coherent cache protocol) для сокрытия наличия кэшей от программиста. При использовании такого протокола несколько процессоров могут одновременно хранить копии одной и той же строки кэша[1] при условии, что они только считывают оттуда данные. Когда процессор должен записать данные в строку кэша, копии этой строки из всех других кэшей удаляются. Когда процессор запрашивает данные, отсутствующие в кэше, запрос идет к общей шине, и данные выбираются либо из памяти, либо из кэша другого процессора.

Время, затрачиваемое одним процессором на общение с другим, примерно вдвое превышает стоимость обращения к памяти. Данные, единицей которых служит строка кэша, сначала должны быть записаны из кэша первого процессора в память, а затем выбраны из памяти в кэш второго процессора. Вы можете подумать, что межпроцессорное взаимодействие — относительно дешевая операция, раз она всего лишь в два раза медленнее обращения к памяти. Но вы должны помнить, что обращения к памяти очень до́роги по сравнению с обращением к кэшу — они могут быть медленнее в сотни раз. Этот анализ подводит нас к сходству

[1]О кэше и строках кэша рассказывалось в разделе 7.4.

между параллельностью и локальностью. Для эффективной работы процессора — как самого по себе, так и в контексте многопроцессорной системы — требуется, чтобы большинство данных, с которыми он работает, находились в кэше.

В начале 2000-х годов проектирование симметричных многопроцессорных систем не выходило за пределы десятков процессоров, поскольку общая шина (или любой иной способ межпроцессорного взаимодействия) не могла обеспечить быструю работу при большем количестве процессоров. Чтобы обеспечить масштабируемость многопроцессорных систем, потребовалось введение еще одного уровня в иерархии памяти. Вместо памяти, доступной каждому процессору, в новой архитектуре память распределена для каждого процессора отдельно, так, чтобы каждый процессор мог быстро работать со своей локальной памятью, как показано на рис. 11.2. Таким образом, удаленная память образует очередной уровень иерархии; ее объем больше, но больше и время обращения к ней. Аналогично принципу проектирования иерархии памяти, который гласит, что быстрой памяти всегда меньше, чем медленной, можно сформулировать правило, что машины с быстрым межпроцессорным взаимодействием всегда имеют меньшее количество процессоров.

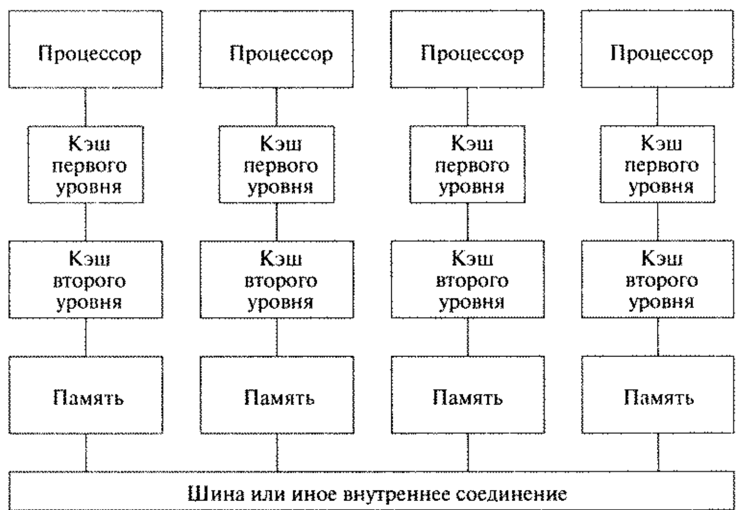


Рис. 11.2. Машины с распределенной памятью

Существует два варианта машин с распределенной памятью: машины с неравномерным доступом к памяти (nonuniform memory access — NUMA) и машины с передачей сообщений. Архитектура NUMA предоставляет программному обеспечению общее адресное пространство, позволяя процессорам взаимодействовать, записывая и считывая общие ячейки памяти. В машинах с передачей сообщений адресные пространства процессоров различны, и процессоры взаимодействуют путем передачи сообщений друг другу. Заметим, что хотя для машин

с общей памятью написание кода оказывается более простой задачей, программное обеспечение в любом случае должно обеспечивать высокую локальность для эффективной работы машины.

11.1.2 Параллелизм в приложениях

Для оценки работоспособности параллельного приложения мы будем использовать две метрики: *степень параллелизма* (parallelism coverage), представляющую собой процент вычислений, которые могут выполняться параллельно, и *зернистость параллелизма* (granularity of parallelism), представляющую собой количество вычислений, которые каждый процессор может выполнить без синхронизации или взаимодействия с другими процессорами. Одним из наиболее привлекательных приложений параллелизма являются циклы: цикл может состоять из многих итераций, и если они независимы одна от другой, то мы получаем неиссякаемый источник параллелизма.

Закон Амдаля

Важность степени параллелизма кратко выражается *законом Амдаля* (Amdahl), который гласит, что если f — доля распараллеленного кода и если распараллеленная версия выполняется на машине с p процессорами без накладных расходов на взаимодействие процессоров и организацию параллельных вычислений, то ускорение работы программы составляет

$$\frac{1}{(1-f)+(f/p)}$$

Например, если половина вычислений остается последовательной, то скорость программы нельзя повысить больше чем в два раза, независимо от количества имеющихся процессоров. При наличии четырех процессоров программа ускорится в 1,6 раза. Даже если степень параллелизма — 90%, то на четырех процессорах мы получим ускорение только в три раза, а при неограниченном количестве процессоров скорость работы программы возрастет в десять раз.

Зернистость параллелизма

В идеале все вычисления приложения могут быть разделены на много независимых крупных задач, поскольку в таком случае их можно просто назначить разным процессорам. Одним из таких примеров является проект SETI (Search for Extra-Terrestrial Intelligence — поиск внеземных цивилизаций), представляющий собой проект с использованием подключенных к Интернету домашних компьютеров для параллельного анализа различных частей данных, полученных с радиотелескопа. Каждое задание требует только небольшого количества входных данных и генерирует небольшие выходные данные, и может выполняться независимо от

всех остальных. В результате такие вычисления хорошо работают на множестве машин в Интернете, характеризующемся высокими задержками в линиях связи и низкой пропускной способностью.

Большинство приложений требуют большего взаимодействия между процессорами, даже если они и допускают разделение на крупные подзадачи. Рассмотрим, например, Web-сервер, ответственный за обслуживание большого количества, в основном, независимых запросов к базе данных. Такое приложение может быть запущено в многопроцессорной среде, в которой один поток реализует базу данных, а множество других обслуживает пользовательские запросы. Другими примерами могут служить разработка лекарств и аэродинамическое моделирование, в которых результаты для множества различных параметров могут вычисляться независимо. Иногда вычисления даже для одного множества параметров моделирования могут выполняться так долго, что их желательно ускорить при помощи распараллеливания. Если зернистость параллелизма снижается, требуются более мощная поддержка межпроцессорного взаимодействия и бóльшие усилия при программировании таких приложений.

Многие длительные научные и инженерные приложения с простыми управляющими структурами и большими множествами данных могут быть распараллелены более легко, чем упоминавшиеся ранее приложения. В этой главе мы, в первую очередь, сосредоточимся на методах, применимых к численным приложениям, в частности к программам, затрачивающим бóльшую часть времени на работу с данными в многомерных массивах.

11.1.3 Параллелизм на уровне циклов

Основным объектом распараллеливания служат циклы, в особенности в приложениях, использующих массивы. Долго работающие приложения, как правило, тяготеют к большим массивам данных, приводящим к циклам с множеством итераций, по одной для каждого элемента массива. При этом не такая уж редкость массивы, в которых итерации не зависят одна от другой. Можно разделить большое количество итераций в таких циклах между процессорами. Если количество вычислений, выполняемых в каждой итерации, примерно одинаковое, простое равномерное разделение итераций по процессорам обеспечит максимальную степень параллелизма. Исключительно простой пример 11.1 демонстрирует возможность использования преимуществ параллелизма на уровне цикла.

Пример 11.1. Цикл

```
for(i = 0; i < n; i++) {
    Z[i] = X[i] - Y[i];
    Z[i] = Z[i] * Z[i];
}
```

Параллелизм на уровне задач

Можно найти параллельность и вне итераций цикла. Например, можно назначить двум процессорам вызовы двух разных функций или выполнения двух независимых циклов. Такой параллелизм известен как *параллелизм на уровне задач*. Уровень задач не такой привлекательный источник параллелизма, как уровень цикла. Причина этого в том, что количество независимых задач в каждой программе постоянно и не изменяется с изменением размера данных, как в случае итераций в типичном цикле. Кроме того, в общем случае задачи имеют разные размеры, так что очень трудно все время загружать все процессоры системы.

вычисляет квадраты разностей между элементами векторов X и Y и сохраняет их в векторе Z. Цикл распараллеливаем, поскольку каждая его итерация обращается к различным множествам данных. Можно выполнить цикл на компьютере с M процессорами, присваивая каждому процессору уникальный идентификатор $p = 0, 1, \ldots, M - 1$ и выполняя на каждом процессоре один и тот же код

```
b = ceil(n/M);
for(i = b*p; i < min(n,b*(p+1)); i++) {
    Z[i] = X[i] - Y[i];
    Z[i] = Z[i] * Z[i];
}
```

Мы равномерно разделяем итерации цикла между процессорами; p-й процессор получает для выполнения p-ю "полосу" итераций. Заметим, что количество итераций может не быть кратно M, так что мы не можем гарантировать, что последний процессор не достигнет конца исходного цикла за минимальное количество операций. □

Параллельный код, приведенный в примере 11.1, представляет собой SPMD-программу (Single Program Multiple Data — одна программа, много данных). Один и тот же код выполняется всеми процессорами, но для каждого процессора этот код параметризован уникальным идентификатором процессора, так что различные процессоры могут выполнять различные действия. Обычно один процессор, известный как *ведущий* (master), выполняет все последовательные части вычислений. При достижении параллелизованной части кода ведущий процессор активирует все *ведомые* (slave) процессоры. Все вместе процессоры выполняют параллелизованную часть кода, после чего участвуют в *барьерной синхронизации* (barrier synchronization). Все операции, выполняемые процессором до входа

в барьер синхронизации, гарантированно завершаются до того, как любому другому процессору будет позволено покинуть этот барьер и выполнить операцию, находящуюся за ним.

Если параллелизуются только небольшие циклы наподобие приведенного в примере 11.1, то получающийся в результате код имеет невысокую степень параллелизма и относительно малую его зернистость. Предпочтительно параллелизовать внешние циклы программы, поскольку это существенно повышает зернистость параллелизма. Рассмотрим, например, приложение двумерного быстрого преобразования Фурье (БПФ), работающего с массивом данных размером $n \times n$. Такая программа выполняет n БПФ каждой строки данных, затем — n БПФ столбцов. Предпочтительнее назначить каждому из n независимых БПФ свой процессор, чем использовать несколько процессоров для выполнения одного БПФ. Такой код легче написать, степень параллельности алгоритма окажется равной 100%, и код будет иметь хорошую локальность, поскольку во время вычисления БПФ взаимодействие процессоров не потребуется.

Во многих приложениях нет параллелизуемых больших внутренних циклов. Однако зачастую во времени выполнения таких приложений доминирует время, затраченное на работу *ядер* (kernels), которые могут состоять из сотен строк кода с циклами разной степени вложенности. Иногда оказывается возможным реорганизовать вычисления в ядре и разбить его на малозависимые модули, сосредоточиваясь, в первую очередь, на локальности.

11.1.4 Локальность данных

Существует два несколько отличающихся понятия локальности данных, которые необходимы при рассмотрении параллелизма программ. *Временна́я* локальность означает использование некоторых данных несколько раз за короткий промежуток времени. *Пространственная* локальность означает, что за небольшой промежуток времени выполняется обращение к данным, находящимся рядом друг с другом. Важным видом пространственной локальности является совместное использование элементов данных из одной строки кэша. Причина такого выделения в том, что когда требуется один элемент из строки кэша, то в кэш загружается вся строка целиком, и, вероятно, все элементы строки будут находиться в кэше до тех пор, пока программа будет постоянно их использовать. Такая пространственная локальность приводит к минимизации промахов кэша, что, в свою очередь, существенно ускоряет работу программы.

Ядра часто могут быть написаны различными семантически эквивалентными способами, существенно отличающимися локальностью данных (и производительностью). В примере 11.2 показан альтернативный способ написания кода, эквивалентного коду из примера 11.1.

Пример 11.2. Так же, как и в примере 11.1, данный код находит квадраты разности элементов векторов X и Y:

```
for(i = 0; i < n; i++) {
    Z[i] = X[i] - Y[i];
}
for(i = 0; i < n; i++) {
    Z[i] = Z[i] * Z[i];
}
```

Первый цикл находит разности, второй — квадраты разностей. Код наподобие приведенного — не редкость в реальных программах, поскольку так можно оптимизировать программу для использования на *векторных машинах*, представляющих собой суперкомпьютеры с командами для выполнения простых арифметических операций над векторами за один раз. В примере 11.1 тела обоих циклов собраны в один.

Обе программы выполняют одни и те же вычисления. Какая же из них будет работать быстрее? Цикл в примере 11.1 имеет более высокую производительность в силу лучшей локальности данных. Каждая разность возводится в квадрат сразу же после вычисления; более того, разность может быть сохранена в регистре, возведена в квадрат и только затем полученное значение можно записать в ячейку памяти $Z[i]$ (в отличие от кода в данном примере, где первая запись $Z[i]$ выполняется существенно ранее его использования). Если размер массива превышает размер кэша, в нашем примере во втором цикле потребуется новая загрузка $Z[i]$ из памяти. Таким образом, этот код должен работать существенно медленнее, чем код из предыдущего примера. □

Пример 11.3. Предположим, что мы хотим установить все элементы построчно хранящегося в памяти массива Z (о способах хранения массивов мы уже говорили в разделе 6.4.3) равными 0. На рис. 11.3, *а* и *б* обнуление элементов выполняется соответственно по столбцам и по строкам. Для получения одного варианта кода из другого его достаточно просто транспонировать. С точки зрения пространственной локальности предпочтительно обнулять массив построчно, поскольку все слова в строке кэша обнуляются последовательно. В случае постолбцового обнуления, хотя каждая строка кэша повторно используется последовательными итерациями внешнего цикла, строки кэша перед повторным использованием будут сброшены, если размер столбца превышает размер кэша. Для получения наивысшей производительности мы распараллеливаем внешний цикл на рис. 11.3, *б* так же, как делали это в примере 11.1 и как показано на рис. 11.3, *в*. □

Два приведенных выше примера иллюстрируют несколько важных характеристик, относящихся к численным приложениям, работающим с массивами.

```
for (j = 0; j < n; j++)
    for (i = 0; i < n; i++)
        Z[i,j] = 0;
```

а) Постолбцовое обнуление массива

```
for (i = 0; i < n; i++)
    for (j = 0; j < n; j++)
        Z[i,j] = 0;
```

б) Построчное обнуление массива

```
b = ceil(n/M);
for (i = b*p; i < min(n,b*(p+1)); i++)
    for (j = 0; j < n; j++)
        Z[i,j] = 0;
```

в) Распараллеленное построчное обнуление массива

Рис. 11.3. Последовательный и параллельный коды обнуления элементов массива

- Код для работы с массивом часто содержит много распараллеливаемых циклов.
- Если цикл распараллеливаем, то его итерации могут выполняться в произвольном порядке. Они могут быть переупорядочены, чтобы получить существенное повышение локальности данных.
- При создании больших модулей параллельных вычислений, независимых друг от друга, их последовательное выполнение обычно имеет тенденцию к высокой локальности данных.

11.1.5 Введение в теорию аффинных преобразований

Написать корректную и эффективную последовательную программу — дело непростое, но написать корректную и эффективную параллельную программу — дело куда более сложное. Уровень сложности возрастает с уменьшением зернистости параллелизма. Как мы видели выше, для получения высокой производительности программисты должны особое внимание уделять локальности данных. Исключительно трудна также задача преобразования существующей последовательной программы в параллельную. Очень сложно обнаружить все зависимости в программе, в особенности если эта программа не знакома, как свои пять пальцев. Еще сложнее отладить параллельную программу, что связано с недетерминистическим характером ошибок. В идеальном случае распараллеливающий

компилятор автоматически транслирует обычные последовательные программы в эффективные параллельные, оптимизируя при этом их локальность. К сожалению, компиляторы без высокоуровневого знания о приложении могут только сохранить семантику исходного алгоритма, который может плохо поддаваться распараллеливанию. Кроме того, выбор реализации алгоритма программистом может дополнительно ограничить параллелизм программы.

Успешность распараллеливания и оптимизации локальности была продемонстрирована для множества численных приложений на языке Fortran, которые выполняют аффинный доступ к массивам. Отсутствие в языке указателей и соответствующей арифметики облегчает анализ программ. Заметим, что не все приложения используют аффинные обращения к массивам. Многие численные приложения работают с разреженными матрицами, обращение к которым выполняется косвенно, через элементы другого массива. В этой главе мы остановимся на распараллеливании и оптимизации ядер, состоящих, в основном, из десятков строк.

Как продемонстрировано в примерах 11.2 и 11.3, параллелизм и оптимизация локальности требуют выделения отдельных экземпляров циклов и выяснения их взаимоотношения друг с другом. Эта ситуация существенно отличается от анализа потоков данных, при котором мы объединяли в одно целое информацию, связанную с отдельными экземплярами.

В задаче оптимизации циклов с обращениями к массивам мы используем три вида пространств. Каждое пространство может рассматриваться как точки одномерной или многомерной структуры.

1. *Пространство итераций* представляет собой множество динамически выполняемых в процессе вычислений экземпляров итераций, т.е. множество комбинаций значений, которые принимают индексы циклов.

2. *Пространство данных* представляет собой множество элементов массива, к которым выполняется обращение.

3. *Пространство процессоров* представляет собой множество процессоров в системе. Обычно этим процессорам назначены целочисленные номера или векторы для того, чтобы отличать один процессор от другого.

В качестве входных данных мы получаем последовательный порядок, в котором выполняются итерации, и аффинные функции обращения к массивам (например, $X[i, j+1]$), которые указывают порядок обращения экземпляров из пространства итераций к элементам пространства данных.

Выходные данные оптимизации также представляют собой аффинные функции, определяющие, что и когда должен делать каждый процессор. Для указания, что должен делать каждый процессор, мы используем аффинную функцию, назначающую процессорам экземпляры из исходного пространства итераций. Для

определения, когда он должен это делать, используется аффинная функция, отображающая экземпляры из пространства итераций в новое упорядочение. План порождается путем анализа зависимостей данных и шаблонов использования в функциях обращения к массивам.

В приведенном ниже примере демонстрируются три описанные пространства — итераций, данных и процессоров. В нем также неформально описываются важные концепции и вопросы, которые должны быть решены при использовании данных пространств для распараллеливания кода. Все эти концепции будут детально рассмотрены в последующих разделах.

Пример 11.4. На рис. 11.4 проиллюстрированы различные пространства и их взаимоотношения для программы

```
float Z[100];
for(i = 0; i , 10; i++)
    Z[i+10] = Z[i];
```


Рис. 11.4. Пространства итераций, данных и процессоров к примеру 11.4

Три пространства и отображения между ними выглядят следующим образом.

1. *Пространство итераций.* Пространство итераций представляет собой множество итераций, идентификаторы которых задаются значениями, хранящимися в индексных переменных. *Вложение циклов* (loop nest) глубиной d (т.е. d вложенных циклов) имеет d индексных переменных и, таким образом, моделируется d-мерным пространством. Пространство итераций ограничено нижней и верхней границами индексов циклов. В нашем примере

цикл определяет одномерное пространство из десяти итераций, помеченных значениями индекса цикла $i = 0, 1, \ldots, 9$.

2. *Пространство данных.* Пространство данных задается непосредственно объявлениями массивов. В нашем примере элементы массива проиндексированы значениями $a = 0, 1, \ldots, 99$. Несмотря на то что любой массив представляет собой линейный участок в адресном пространстве программы, мы рассматриваем n-мерные массивы как n-мерные пространства и считаем, что отдельные индексы находятся в их границах. Массив в нашем примере одномерный.

3. *Пространство процессоров.* Будем считать, что в целевой системе имеется неограниченное количество виртуальных процессоров. Процессоры организованы в многомерное пространство, по одному измерению для каждого цикла вложения, которое мы хотим распараллелить. Если после распараллеливания оказывается, что физических процессоров у нас меньше, чем виртуальных, мы разделяем виртуальные процессоры на одинаковые блоки и назначаем каждый блок физическому процессору. В данном примере необходимо десять процессоров, по одному для каждой итерации цикла. На рис. 11.4 предполагается, что процессоры организованы в одномерном пространстве и пронумерованы от 0 до 9, так что процессор i выполняет i-ю итерацию. Если у нас, скажем, всего пять физических процессоров, то мы можем назначить итерации 0 и 1 процессору 0, итерации 2 и 3 — процессору 1 и т.д. Поскольку итерации независимы, не имеет значения, как именно будет выполнено их распределение по процессорам, важно лишь, что каждому из пяти процессоров будут назначены две итерации.

4. *Аффинная индексная функция.* Каждое обращение к массиву в коде определяет отображение итерации из пространства итераций на элемент массива в пространстве данных. Функция обращений является аффинной, если она включает только умножения индексных переменных на константы, а также суммирование полученных произведений и прибавление константы. Обе функции обращения в нашем примере — i и $i + 10$ — аффинные. Зная функцию обращения, мы можем говорить о *размерности* данных, к которым выполняется обращение. В нашем случае, поскольку каждая функция содержит только одну переменную цикла, пространство элементов массива, к которым выполняется обращение, одномерно.

5. *Аффинное разбиение.* Мы распараллеливаем цикл, используя аффинную функцию для назначения итераций из пространства итераций процессорам из пространства процессоров. В нашем примере мы просто назначаем итерацию i процессору i. При помощи аффинной функции можно также

указать новый порядок выполнения. Если мы хотим выполнять наш цикл последовательно, но в обратном порядке, достаточно просто определить упорядочивающую функцию как аффинное выражение $10 - i$. Таким образом, первой будет выполнена итерация 9, и т.д.

6. *Область данных, к которым выполняется обращение.* Для поиска наилучшего аффинного разбиения желательно знать область данных, к которым выполняется обращение. Можно получить эту область данных путем объединения информации о пространстве итераций с индексной функцией массива. В нашем случае обращение к массиву $Z[i+10]$ затрагивает область $\{a \mid 10 \leqslant a < 20\}$, а обращение $Z[i]$ – область $\{a \mid 0 \leqslant a < 10\}$.

7. *Зависимости данных.* Для определения того, можно ли распараллелить данный цикл, следует выяснить, пересекают ли зависимости данных границы каждой итерации. В нашем примере мы сначала рассматриваем зависимости записей. Поскольку функция обращения $Z[i+10]$ отображает различные итерации на различные элементы массива, здесь нет зависимостей, которые относятся к порядку, в котором разные итерации записывают значения в массив. Но нет ли здесь зависимостей между обращениями для чтения и обращениями для записи? Поскольку записываются только элементы $Z[10], Z[11], \ldots, Z[19]$ (обращение к $Z[i+10]$), а считываются только $Z[0], Z[1], \ldots, Z[9]$ (обращение к $Z[i]$), нет никаких зависимостей, связанных с относительным порядком чтения и записи. Таким образом, данный цикл можно распараллелить, т.е. каждая итерация цикла не зависит от всех остальных итераций, и эти итерации могут быть выполнены параллельно, в любом выбранном порядке. Заметим, однако, что если внести небольшие изменения, например установить верхний предел для индекса i равным 10 или более, то в таком случае возникнут зависимости, поскольку некоторые элементы массива Z записываются в одной итерации, а затем считываются десятью итерациями позже. В таком случае полное распараллеливание цикла невозможно, и нам надо хорошо подумать о том, как распределить итерации среди процессоров и в каком порядке их выполнять. □

Формулировка задачи в терминах многомерных пространств и аффинных отображений между ними позволяет применить стандартный математический аппарат для решения задачи распараллеливания и оптимизации локальности в общем виде. Например, область данных, к которым выполняется обращение, можно найти путем устранения переменных с использованием алгоритма Фурье–Моцкина (Fourier–Motzkin elimination algorithm). Зависимости данных эквивалентны задаче целочисленного линейного программирования. Наконец, поиск аффинного разбиения соответствует решению множества линейных ограничений. Не

беспокойтесь, если вам ничего не говорят упомянутые здесь названия — все они будут пояснены позже, начиная с раздела 11.3.

11.2 Пример посерьезнее: умножение матриц

Множество методов, используемых параллельными компиляторами, можно рассмотреть на одном большом примере. В этом разделе мы изучим хорошо известный алгоритм умножения матриц, чтобы показать, насколько нетривиальна задача оптимизации даже такого простого и легко распараллеливаемого алгоритма. Мы увидим, как переписывание кода может повысить локальность кода, т.е. процессоры будут иметь возможность выполнять работу с существенно меньшим взаимодействием (с глобальной памятью или друг с другом — в зависимости от используемой архитектуры), чем в случае выбора обычной программы, непосредственно реализующей алгоритм. Мы также обсудим, как может повысить производительность такой программы, как матричное умножение, существование строк кэша, хранящих несколько последовательных элементов данных.

11.2.1 Алгоритм умножения матриц

На рис. 11.5 показана типичная программа умножения матриц.[2] Она принимает две матрицы, X и Y, размером $n \times n$ и выдает их произведение в качестве матрицы Z. Вспомним, что Z_{ij} — элемент матрицы Z в строке i и столбце j — должен быть равен $\sum_{k=1}^{n} X_{ik}Y_{kj}$.

```
for (i = 0; i < n; i++)
    for (j = 0; j < n; j++) {
        Z[i,j] = 0.0;
        for (k = 0; k < n; k++)
            Z[i,j] = Z[i,j] + X[i,k]*Y[k,j];
    }
```

Рис. 11.5. Базовый алгоритм умножения матриц

Код на рис. 11.5 генерирует n^2 результатов, каждый из которых представляет собой скалярное произведение одной строки и одного столбца из двух операндов-матриц. Очевидно, что вычисления каждого элемента Z независимы и могут быть выполнены параллельно.

Чем больше значение n, тем большее количество раз алгоритм обращается к каждому элементу. Всего в трех матрицах используются $3n^2$ ячеек памяти, но

[2]В программах этой главы мы, в основном, используем синтаксис C, но при обращении к многомерным массивам (что является центральным вопросом этой главы) для простоты чтения используется запись в стиле Fortran — $Z[i,j]$ вместо $Z[i][j]$.

алгоритм выполняет n^3 операций, состоящих в умножении элемента X на элемент Y и прибавлении полученного произведения к элементу Z. Таким образом, данный алгоритм выполняет интенсивные вычисления и обращения к памяти, которые в принципе, не должны быть его узким местом.

Последовательное выполнение умножения матриц

Сначала рассмотрим, как ведет себя программа при последовательной работе на единственном процессоре. Наиболее глубоко вложенный цикл считывает и записывает один и тот же элемент Z, используя при этом строку X и столбец Y. $Z[i, j]$ можно легко хранить в регистре без обращений к памяти. Без потери общности будем считать, что матрицы располагаются в памяти построчно и что в одной строке кэша может храниться c элементов массива.

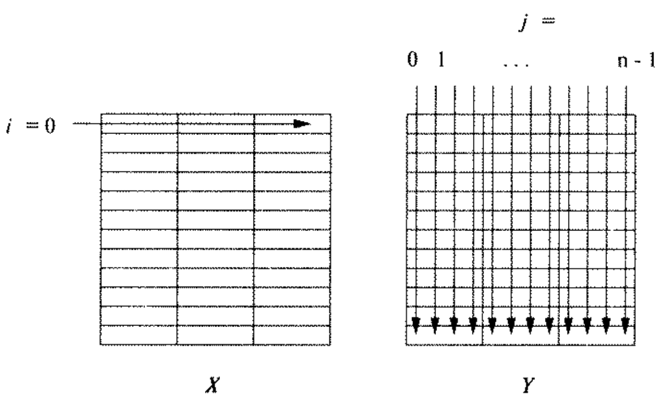


Рис. 11.6. Шаблон обращения к данным при умножении матриц

На рис. 11.6 показан шаблон обращения к матрицам при выполнении одной итерации внешнего цикла, представленной на рис. 11.5. Конкретно на рисунке показана первая итерация, с $i = 0$. Каждый раз при переходе от одного элемента первой строки X к следующему мы посещаем каждый элемент одного столбца Y. На рис. 11.6 представлено предположительное размещение матриц в строках кэша. Каждый небольшой прямоугольник на рисунке представляет строку кэша, в которой хранятся четыре элемента массива (т.е. на рисунке представлен случай $c = 4$ и $n = 12$).

Обращение к X обеспечивается небольшой нагрузкой на кэш. Одна строка X распределена среди n/c строк кэша. В предположении, что все они помещаются в кэше, можно считать, что для фиксированного значения индекса i наблюдается n/c промахов кэша, а общее количество промахов для всех элементов X составляет n^2/c — это минимально возможное значение (для удобства мы считаем, что n делится на c).

Однако при использовании одной строки X алгоритм умножения матриц обращается ко всем элементам Y столбец за столбцом. Иначе говоря, при $j = 0$ внутренний цикл загружает в кэш весь первый столбец Y. Заметим, что элементы этого столбца хранятся в n разных строках кэша. Если кэш достаточно большой (или n достаточно мало́) для хранения n строк и кэш больше никем не используется, то столбец для $j = 0$ останется в кэше, когда нам потребуется второй столбец Y. В этом случае промахов кэша не будет до тех пор, пока не будет достигнуто значение $j = c$, когда потребуется внести в кэш полностью новое множество строк Y. Таким образом, для завершения первой итерации внешнего цикла (для которой $i = 0$) потребуется от n^2/c до n^2 промахов, в зависимости от того, смогут ли столбцы строк кэша оставаться в нем от одной итерации второго цикла до другой.

По завершении внешнего цикла при i, равном 1, 2 и так далее, может возникнуть большое количество дополнительных промахов кэша при чтении Y (а может, их не будет вовсе). Если кэш достаточно велик для того, чтобы все n^2/c строк кэша, в которых хранится матрица Y, могли располагаться в кэше, то новых промахов не будет. Таким образом, общее количество промахов кэша в этом случае составит $2n^2/c$, одна половина — для матрицы X, а другая — для матрицы Y. Но если кэш может хранить только один столбец Y, но не всю матрицу целиком, то потребуется полная загрузка всей матрицы Y для каждой итерации внешнего цикла. Таким образом, общее количество промахов кэша окажется равным $n^2/c + n^3/c$: первый член — для матрицы X, а второй — для матрицы Y. Хуже того, если в кэше нельзя хранить столбец матрицы Y полностью, то у нас будет n^2 промахов на итерацию внешнего цикла, а общее количество промахов составит $n^2/c + n^3$.

Построчное распараллеливание

Давайте теперь посмотрим, как можно использовать несколько процессоров (скажем, p) для того, чтобы ускорить выполнение программы на рис. 11.5. Очевидный подход заключается в том, чтобы назначить разным процессорам вычисление различных строк Z. Каждый процессор при этом отвечает за вычисление n/p соседних строк (для удобства будем считать, что n делится на p). При таком разделении труда каждому процессору потребуется доступ к n/p строкам матриц X и Z и обращение ко всей матрице Y. Один процессор будет вычислять n^2/p элементов матрицы Z, выполняя n^3/p операций умножения и сложения (в этом разделе пара операций — умножения и сложения — рассматривается нами как единая операция).

Таким образом, хотя время вычислений и уменьшается пропорционально p, стоимость взаимодействия растет пропорционально p, т.е. каждый из p процессоров должен прочесть n^2/p элементов X и все n^2 элементов Y. Общее количество строк кэша, которые должны быть загружены в кэши p процессоров, как минимум

равно $n^2/c + pn^2/c$; эти два члена соответствуют загрузке X и копий Y. Если p приближается к n, то время вычисления становится равным $O\left(n^2\right)$, а стоимость взаимодействия — $O\left(n^3\right)$. Таким образом, узким местом этой системы становится шина, по которой пересылаются данные между памятью и кэшами процессоров. Итак, при предложенной схеме данных использование большого количества процессоров может не ускорить, а напротив, замедлить вычисления.

11.2.2 Оптимизации

Алгоритм умножения матриц на рис. 11.5 демонстрирует, что, хотя алгоритм может многократно использовать одни и те же данные, локальность данных при этом может оставлять желать лучшего. Многократное использование данных дает преимущества при работе с кэшем только в том случае, если многократное использование выполняется в ближайшее время, до того, как данные будут удалены из кэша. В нашем же случае n^2 операций умножения и сложения используют одни и те же элементы данных матрицы Y в слишком разные моменты времени, так что результирующая локальность оказывается низкой. Кроме того, в многопроцессорной системе многократное использование дает положительный результат только в том случае, когда данные используются одним и тем же процессором. При рассмотрении реализации параллельных вычислений в разделе 11.2.1 мы видели, что элементы Y должны были использоваться каждым из процессоров. Таким образом, многократное использование Y не означает локальности данных.

Изменение схемы данных

Один из способов повышения локальности программы заключается в изменении схемы используемых ею структур данных. Например, постолбцовое сохранение Y могло бы повысить степень повторного использования строк кэша для матрицы Y. Применимость такого подхода ограничена в силу того, что одна и та же матрица обычно используется в разных операциях. Если Y будет играть роль X в другом матричном умножении, то производительность этого умножения упадет от постолбцового размещения матрицы, поскольку для получения более высокой производительности первая матрица-сомножитель должна храниться в памяти построчно.

Разбиение на блоки

Иногда для повышения локальности данных можно изменить порядок выполнения команд. Однако метод перемены циклов не повышает эффективности программы матричного умножения. Предположим, что программа написана так, чтобы считать матрицу Z не по строкам, а по столбцам, т.е. j-цикл становится внешним, а i-цикл — внутренним. В предположении, что матрицы хранятся в памяти, как и ранее, построчно, мы найдем, что теперь у матрицы Y наблюдается

лучшая временна́я и пространственная локальность, но это происходит за счет матрицы X, у которой эти показатели становятся хуже.

Другим способом переупорядочения итераций в циклах, который может привести к повышению локальности программы, является *разбиение на блоки*, или *блокирование* (blocking). Вместо вычисления результата построчно или постолбцово мы делим матрицу на подматрицы, или *блоки*, как показано на рис. 11.7, и переупорядочиваем операции так, чтобы весь блок использовался в течение короткого промежутка времени. Обычно блоки представляют собой квадраты со стороной длиной B. Если B является делителем n, то все блоки являются квадратами; если не является, то блоки у нижней и правой границы будут меньшего размера.

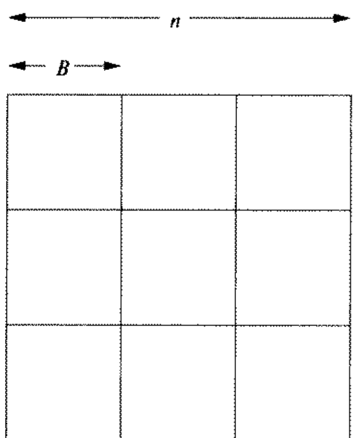


Рис. 11.7. Матрица, разделенная на блоки размером B

На рис. 11.8 показана версия базового алгоритма умножения матриц, в котором все три матрицы разделены на квадратные блоки размером B. Как и на рис. 11.5, считается, что матрица Z инициализирована нулевыми значениями. Мы считаем, что n кратно B; если это не так, то следует изменить строки 4–6 так, чтобы верхний предел был равен $\min(ii + B, n)$.

Три внешних цикла в строках 1–3 используют индексы ii, jj и kk, которые всегда увеличиваются на B и, таким образом, маркируют левую или верхнюю границу некоторого блока. При фиксированных значениях индексов ii, jj и kk строки 4–7 обеспечивают внесение в блок с верхним левым углом $Z[ii, jj]$ полного вклада от блоков с верхними левыми углами $X[ii, kk]$ и $Y[kk, jj]$.

При корректном выборе размера блоков B можно существенно снизить количество промахов кэша по сравнению с базовым алгоритмом, если матрицы X, Y и Z не помещаются в кэше. Значение B следует выбирать таким, чтобы можно было разместить в кэше по одному блоку из каждой матрицы. В силу выбранного

```
1)  for (ii = 0; ii < n; ii = ii+B)
2)      for (jj = 0; jj < n; jj = jj+B)
3)          for (kk = 0; kk < n; kk = kk+B)
4)              for (i = ii; i < ii+B; i++)
5)                  for (j = jj; j < jj+B; j++)
6)                      for (k = kk; k < kk+B; k++)
7)                          Z[i,j] = Z[i,j] + X[i,k]*Y[k,j];
```

Рис. 11.8. Матричное умножение с разбивкой на блоки

Другой взгляд на блочное умножение матриц

Можно представить, что матрицы X, Y и Z на рис. 11.8 представляют собой не матрицы размером $n \times n$ чисел с плавающей точкой, а матрицы размером $(n/B) \times (n/B)$, элементами которых являются матрицы размером $B \times B$ чисел с плавающей точкой. Строки 1–3 на рис. 11.8 в таком случае аналогичны трем циклам базового алгоритма на рис. 11.5, но для размеров матриц не n, а n/B. Строки 4–7 можно рассматривать как реализацию единой операции умножения и сложения на рис. 11.5. Заметим, что единый шаг умножения представляет собой умножение матриц, которое использует базовый алгоритм на рис. 11.5 для чисел с плавающей точкой, являющихся элементами умножаемых матриц. Матричное суммирование представляет собой простое поэлементное сложение чисел с плавающей точкой.

порядка циклов каждый блок Z требуется в кэше только один раз, так что (как и в анализе базового алгоритма в разделе 11.2.1) мы не учитываем промахи кэша, связанные с Z.

Для загрузки блоков X и Y в кэш требуется B^2/c промахов (вспомним, что c — количество элементов в строке кэша). Для фиксированных блоков X и Y в строках 4–7 выполняется B^3 операций умножения и сложения (напомним, что эти два действия рассматриваются как единая операция). Поскольку всего умножение матриц требует выполнения n^3 операций, количество загрузок пар блоков в кэш оказывается равным n^3/B^3. Поскольку каждый раз при загрузке пары блоков мы сталкиваемся с $2B^2/c$ промахами кэша, общее их количество равно $2n^3/Bc$.

Интересно сравнить полученное значение $2n^3/Bc$ с оценкой из раздела 11.2.1. В нем говорилось, что если вся матрица может быть размещена в кэше, то достаточно $O\left(n^2/c\right)$ промахов. Однако в таком случае мы можем выбрать $B = n$, т.е. рассматривать каждую матрицу как единый блок. В этом случае мы получим ту же оценку количества промахов — $O\left(n^2/c\right)$. С другой стороны, если матрицы полностью в кэш не помещаются, то потребуется $O\left(n^3/c\right)$ или даже $O\left(n^3\right)$ про-

махов кэша. В этом случае использование разбиения на блоки даст существенный выигрыш, в особенности если учесть, что B может быть достаточно большим (например, при B, равном 200, можно разместить все три блока 8-байтовых чисел в кэше размером 1 Мбайт).

Метод разбиения на блоки может быть применен на каждом уровне иерархии памяти. Например, мы можем захотеть оптимизировать использование регистров путем хранения операндов умножения матриц размером 2×2 в регистрах. При этом мы можем выбирать блоки все большего и большего размера для различных уровней кэша и физической памяти.

Мы можем распределить блоки между процессорами для минимизации пересылки данных. Эксперименты показывают, что такая оптимизация может повысить производительность однопроцессорной системы в три раза, а ускорение в многопроцессорной системе близко к линейной зависимости от количества используемых процессоров.

11.2.3 Интерференция кэша

К сожалению, к вопросу об использовании кэшей следует добавить еще несколько слов. Большинство кэшей не полностью ассоциативны (см. раздел 7.4.2). Если в кэше с прямым отображением n кратно размеру кэша, то все элементы одной строки массива размером $n \times n$ будут конкурировать за одно и то же место в кэше. В этом случае загрузка второго элемента столбца будет приводить к выгрузке из кэша строки с первым элементом, несмотря на то, что объема кэша достаточно для одновременного хранения обеих строк. Такая ситуация называется *интерференцией кэша* (cache interference).

Существуют различные способы преодоления этой проблемы. Первый из них состоит в такой перестановке данных, чтобы данные, к которым выполняется обращение, находились в последовательных ячейках памяти. Второй способ предполагает вставку массива размером $n \times n$ в больший массив размером $m \times n$, где m выбирается таким образом, чтобы уменьшить интерференцию. Третий способ состоит в том, что в некоторых случаях можно выбрать такой размер блока, который гарантирует отсутствие интерференции.

11.2.4 Упражнения к разделу 11.2

Упражнение 11.2.1. Алгоритм умножения матриц с разделением на блоки, представленный на рис. 11.8, не инициализирует матрицу Z нулевыми значениями, как это делает код на рис. 11.5. Добавьте в код на рис. 11.8 шаги, необходимые для корректной инициализации массива Z нулями.

11.3 Пространства итераций

В простых задачах наподобие матричного умножения применение описанных методов реализуется достаточно просто. Но в общем случае, хотя описанные методы и применимы, делается это гораздо менее интуитивно. Если же воспользоваться линейной алгеброй, все становится существенно проще даже в общем случае.

Как говорилось в разделе 11.1.5, в нашей модели имеется три типа пространств — пространство итераций, пространство данных и пространство процессоров. Сейчас мы начнем наше рассмотрение с пространства итераций. Пространство итераций вложения циклов определяется, как все комбинации индексных переменных вложения.

Зачастую пространство итераций прямоугольное, как в примере умножения матриц на рис. 11.5. Здесь каждый вложенный цикл имеет нижнюю границу 0 и верхнюю границу $n-1$. Однако в более сложных, но при этом достаточно реалистических вложениях циклов верхние и/или нижние границы индексов циклов могут зависеть от значений индексов охватывающих циклов. Вскоре мы познакомимся с примером такой ситуации.

11.3.1 Построение пространств итераций вложений циклов

Сначала опишем вид вложений циклов, с которыми можно работать при помощи рассматриваемых методов. Каждый цикл имеет один индекс, который предполагается возрастающим на 1 на каждой итерации цикла. Это предположение не приводит к потере общности, поскольку при увеличении на целое число $c > 1$ мы всегда можем заменить индекс i величиной $ci + a$ с некоторой положительной или отрицательной константой a и увеличивать i в цикле на 1. Границы цикла должны записываться как аффинные выражения от индексов внешних циклов.

Пример 11.5. Рассмотрим цикл, в котором на каждой итерации цикла значение i увеличивается на 3:

```
for(i = 2; i <= 100; i = i + 3)
    Z[i] = 0;
```

Это приводит к тому, что значение 0 присваивается элементам $Z[2], Z[5], Z[8], \ldots, Z[98]$. Тот же результат можно получить при помощи цикла

```
for(j = 0; i <= 32; j++)
    Z[3*j+2] = 0;
```

Иначе говоря, мы просто заменяем i на $3j+2$. Нижний предел $i = 2$ превращается в $j = 0$ (простым решением уравнения $3j + 2 = 2$), а верхний предел $i \leqslant 100$

становится пределом $j \leqslant 32$ (из $3j + 2 \leqslant 100$ получаем $j \leqslant 32{,}67$ и округляем полученное значение до 32, так как j — целое число). □

Обычно циклы for используются во вложениях циклов. Циклы while и repeat могут быть заменены циклом for, если существует индекс цикла и его нижняя и верхняя границы, как, например, в цикле на рис. 11.9, *а*, который вполне заменим циклом `for(i=0; i<100; i++)`.

```
i = 0;
while (i<100) {
    <Некоторые инструкции с участием i>
    i = i+1;
}
```

а) Цикл while с очевидными границами

```
i = 0;
while (1) {
    <Некоторые инструкции>
    i = i+1;
}
```

б) Неясно, когда и почему завершается данный цикл

```
i = 0;
while (i<n) {
    <Некоторые инструкции, не включающие i или n>
    i = i+1;
}
```

в) Значение n неизвестно, так что неизвестно и когда завершается данный цикл

Рис. 11.9. Некоторые циклы while

Однако некоторые циклы while и repeat не имеют очевидных границ. Например, цикл на рис. 11.9, *б* может завершиться, а может и не завершиться, причем нет способа указать, какое условие, связанное с i в невидимом теле цикла, вызывает его завершение. На рис. 11.9, *в* показан еще один проблемный случай. Переменная n может быть, например, параметром функции. Мы знаем, что цикл выполняется n раз, но во время компиляции мы не знаем, чему равно n, и можем ожидать, что в случае нескольких выполнений этого цикла всякий раз будет выполняться иное количество итераций. В случаях, подобных представленным на рис. 11.9, *б* и *в*, следует полагать, что верхняя граница цикла равна бесконечности.

Вложенность циклов глубиной d может быть смоделирована при помощи d-мерного пространства. Измерения упорядочены, k-е измерение представляет k-й вложенный цикл, считая от самого внешнего цикла в глубину. Точка $(x_1, x_2, \ldots, x_d)$ в этом пространстве представляет значения всех индексов циклов; значение индекса внешнего цикла равно x_1, второго цикла — x_2 и т.д. Значение индекса наиболее глубоко вложенного цикла равно x_d.

Однако не все точки в этом пространстве представляют комбинации индексов, которые в действительности встречаются в процессе выполнения вложенности циклов. Представляя собой аффинную функцию от индексов внешних циклов, каждая верхняя и нижняя граница цикла определяет неравенство, разделяющее пространство итераций на два полупространства: то, в котором имеются итерации цикла (*положительное* полупространство), и то, в котором его нет (*отрицательное* полупространство). Конъюнкция (логическое И) всех линейных неравенств представляет пересечение положительных полупространств и определяет выпуклый многогранник, который мы называем *пространством итераций* вложения циклов. *Выпуклый многогранник* обладает тем свойством, что если две точки принадлежат многограннику, то и все точки на линии между ними также принадлежат этому многограннику. Все итерации цикла представлены точками с целочисленными координатами, находящимися в описанном граничными неравенствами многограннике. И наоборот, все целочисленные точки внутри полиэдра представляют итерации вложенности циклов в некоторый момент времени.

Пример 11.6. Рассмотрим двумерную вложенность циклов, показанную на рис. 11.10. Эта двумерная вложенность может быть смоделирована двумерным многоугольником, показанным на рис. 11.11. Две оси представляют значения индексов циклов i и j. Индекс i может принимать любое целочисленное значение между 0 и 5; индекс j может принимать целочисленные значения, такие, что $i \leqslant j \leqslant 7$. □

```
for(i = 0; i <= 5; i++)
    for(j = i; j <= 7; j++)
        Z[j,i] = 0;
```

Рис. 11.10. Двумерная вложенность циклов

11.3.2 Порядок выполнения вложенности циклов

Последовательное выполнение вложенности циклов сканирует итерации в их пространстве в возрастающем лексикографическом порядке. Вектор $\mathbf{i} = [i_0, i_1, \ldots, i_n]$ *лексикографически меньше* вектора $\mathbf{i}' = [i'_0, i'_1, \ldots, i'_{n'}]$, что записывается как $\mathbf{i} \prec \mathbf{i}'$, тогда и только тогда, когда существует $m < \min(n, n')$, такое, что

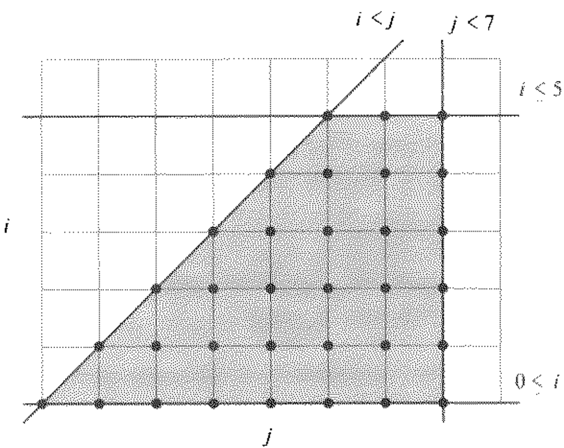


Рис. 11.11. Пространство итераций к примеру 11.6

> **Пространства итераций и обращения к массивам**
>
> В коде на рис. 11.10 пространство итераций представляет собой часть массива Z, к которому в коде выполняется обращение. Такой вид обращения, когда индексами массива являются индексы циклов в том же порядке, весьма распространен. Однако не следует путать пространство итераций, измерениями которого являются индексы циклов, и пространство данных. Если бы на рис. 11.10 вместо $Z[j,i]$ было обращение наподобие $Z[2i, i+j]$, различие этих двух пространств стало бы очевидным.

$[i_0, i_1, \ldots, i_m] = [i'_0, i'_1, \ldots, i'_m]$ и $i_{m+1} < i'_{m+1}$. Заметим, что случай $m = 0$ не только допустим, но и весьма распространен.

Пример 11.7. Рассматривая i как внешний цикл, итерации из вложенности циклов в примере 11.6 выполняются в порядке, показанном на рис. 11.12. □

$[0,0]$,	$[0,1]$,	$[0,2]$,	$[0,3]$,	$[0,4]$,	$[0,5]$,	$[0,6]$,	$[0,7]$
	$[1,1]$,	$[1,2]$,	$[1,3]$,	$[1,4]$,	$[1,5]$,	$[1,6]$,	$[1,7]$
		$[2,2]$,	$[2,3]$,	$[2,4]$,	$[2,5]$,	$[2,6]$,	$[2,7]$
			$[3,3]$,	$[3,4]$,	$[3,5]$,	$[3,6]$,	$[3,7]$
				$[4,4]$,	$[4,5]$,	$[4,6]$,	$[4,7]$
					$[5,5]$,	$[5,6]$,	$[5,7]$

Рис. 11.12. Порядок выполнения итераций вложенности циклов на рис. 11.10

11.3.3 Матричная запись неравенств

Математически итерации цикла глубиной d могут быть представлены как

$$\left\{\mathbf{i} \in Z^d \mid \mathbf{Bi} + \mathbf{b} \geqslant \mathbf{0}\right\} \tag{11.1}$$

Здесь

1. Z, как принято в математике, представляет множество целых чисел — положительных, отрицательных и нуля;
2. **B** представляет собой целочисленную матрицу размером $d \times d$;
3. **b** — целочисленный вектор длиной d;
4. **0** — вектор длиной d, все элементы которого нулевые.

Пример 11.8. Неравенства из примера 11.6 можно записать так, как показано на рис. 11.13. Здесь диапазон i описывается неравенствами $i \geqslant 0$ и $i \leqslant 5$; диапазон j описывается неравенствами $j \geqslant i$ и $j \leqslant 7$. Мы должны привести каждое неравенство к виду $ui + vj + w \geqslant 0$. Тогда $[u, v]$ становится строкой матрицы **B** в неравенстве (11.1), а w — соответствующим компонентом вектора **b**. Например, $i \geqslant 0$ соответствует указанному виду, если $u = 1$, $v = 0$ и $w = 0$. Это неравенство представлено первой строкой матрицы **B** и верхним элементом вектора **b** на рис. 11.13.

$$\begin{bmatrix} 1 & 0 \\ -1 & 0 \\ -1 & 1 \\ 0 & -1 \end{bmatrix} \begin{bmatrix} i \\ j \end{bmatrix} + \begin{bmatrix} 0 \\ 5 \\ 0 \\ 7 \end{bmatrix} \geqslant \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

Рис. 11.13. Неравенство, определяющее пространство итераций

В качестве другого примера неравенство $i \leqslant 5$ можно эквивалентно записать как $(-1)\,i + (0)\,j + 5 \geqslant 0$, что представлено второй строкой **B** и **b** на рис. 11.13. Точно так же неравенство $j \geqslant i$ превращается в $(-1)\,i + (1)\,j + 0 \geqslant 0$ и представлено третьей строкой; а неравенство $j \leqslant 7$ становится $(0)\,i + (-1)\,j + 7 \geqslant 0$ и представлено последней строкой матрицы и вектора. □

11.3.4 Добавление символьных констант

Иногда приходится оптимизировать вложенность циклов, в которой участвуют некоторые переменные, являющиеся инвариантами для всех циклов во вложенности. Мы называем такие переменные *символьными константами* (symbolic

Работа с неравенствами

Преобразовывать неравенства, как в примере 11.8, можно так же, как равенства, т.е. выполняя прибавление или вычитание с обеих сторон или умножая обе стороны на константу. Следует только помнить одно правило — при умножении обеих сторон неравенства на отрицательное число необходимо изменить направление неравенства. Так, $i \leqslant 5$, умноженное на -1, становится неравенством $-i \geqslant -5$. Добавляя 5 к обеим сторонам неравенства, мы получаем $-i + 5 \geqslant 0$, что соответствует второй строке на рис. 11.13.

constants), но при описании границ пространства итераций должны рассматривать их как переменные и создавать записи для них в векторе индексов циклов, т.е. в векторе **i** в общем виде (11.1).

Пример 11.9. Рассмотрим простой цикл

```
for(i = 0; i <= n; i++) {
    ...
}
```

Этот цикл определяет одномерное пространство итераций с индексом i, ограниченное неравенствами $i \geqslant 0$ и $i \leqslant n$. Поскольку n — символьная константа, мы должны включить ее в качестве переменной, что даст нам вектор индексов циклов $[i, n]$. В матричном виде указанное пространство итераций определяется как

$$\left\{ i \in Z \;\middle|\; \begin{bmatrix} -1 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} i \\ n \end{bmatrix} \geqslant \begin{bmatrix} 0 \\ 0 \end{bmatrix} \right\}$$

Заметим, что хотя вектор массива индексов двумерен, только первое его измерение, представляющее i, является частью выхода — множества точек, лежащих в пространстве итераций. □

11.3.5 Управление порядком выполнения

Линейные неравенства, выделенные из нижней и верхней границ тела цикла, определяют множество итераций в выпуклом многограннике. Как таковое представление не определяет порядка выполнения итераций в пределах пространства итераций. Исходная программа накладывает на выполнение итераций последовательный порядок, являющийся лексикографическим порядком по отношению к переменным индексов циклов, упорядоченным от внешних циклов к внутренним. Однако итерации в пространстве могут быть выполнены в произвольном

порядке, лишь бы учитывались их зависимости данных (т.е. порядок, в котором выполняются записи и чтения любого элемента массива в пределах вложенности циклов, остается неизменным).

Задача выбора такого упорядочения, которое отвечает зависимостям данных и оптимизирует локальность данных и параллелизм, сложна и будет рассмотрена позже, начиная с раздела 11.7. Здесь же мы считаем, что корректный порядок, отвечающий всем требованиям, задан, и покажем, как сгенерировать код, обеспечивающий реализацию этого порядка. Начнем с рассмотрения альтернативы порядку из примера 11.6.

Пример 11.10. Зависимостей между итерациями в примере 11.6 нет. Следовательно, мы можем выполнять итерации в произвольном порядке, как последовательно, так и параллельно. Поскольку итерация $[i, j]$ обращается к элементу $Z[j, i]$, исходная программа обходит массив в порядке, показанном на рис. 11.14, *а*. Для повышения пространственной локальности предпочтительнее обход соседних слов, например, в порядке, показанном на рис. 11.14, *б*.

Этот шаблон обращений получится, если выполнять итерации в порядке, показанном на рис. 11.14, *в*. Иначе говоря, пространство итераций на рис. 11.11 сканируется не по горизонтали, а по вертикали, так что j становится индексом внешнего цикла. Код, выполняющий итерации в указанном порядке, имеет вид

```
for(j = 0; j <= 7; j++)
    for(i = 0; i <= min(5,j); i++)
      Z[j,i] = 0;
```

□

Как же сгенерировать границы циклов, которые сканируют пространство в лексикографическом порядке переменных для данного выпуклого многогранника и упорядочения индексных переменных? В приведенном выше примере ограничение $i \leqslant j$ рассматривается как нижняя граница индекса j во внутреннем цикле исходной программы, но как верхняя граница для индекса i (опять же, во внутреннем цикле) в преобразованной программе. Границы внешнего цикла, выраженные в виде линейной комбинации обычных и символьных констант, определяют диапазон всех возможных значений, которые может принимать его индексная переменная. Границы переменных внутренних циклов выражаются как линейные комбинации переменных внешних циклов, символьных и обычных констант. Они определяют диапазон значений переменной для каждой комбинации значений переменных внешних циклов.

Проекция

Говоря геометрически, найти границы индекса внешнего цикла во вложенности циклов глубиной 2 можно путем *проекции* выпуклого многогранника, представляющего пространство итераций, на внешнее измерение пространства. Про-

$$
\begin{array}{cccccccc}
Z[0,0], & Z[1,0], & Z[2,0], & Z[3,0], & Z[4,0], & Z[5,0], & Z[6,0], & Z[7,0] \\
 & Z[1,1], & Z[2,1], & Z[3,1], & Z[4,1], & Z[5,1], & Z[6,1], & Z[7,1] \\
 & & Z[2,3], & Z[3,2], & Z[4,2], & Z[5,2], & Z[6,2], & Z[7,2] \\
 & & & Z[3,3], & Z[4,3], & Z[5,3], & Z[6,3], & Z[7,3] \\
 & & & & Z[4,4], & Z[5,4], & Z[6,4], & Z[7,4] \\
 & & & & & Z[5,5], & Z[6,5], & Z[7,5]
\end{array}
$$

а) Исходный порядок обращений

$$
\begin{array}{llllll}
Z[0,0] \\
Z[1,0], & Z[1,1] \\
Z[2,0], & Z[2,1], & Z[2,2] \\
Z[3,0], & Z[3,1], & Z[3,2], & Z[3,3] \\
Z[4,0], & Z[4,1], & Z[4,2], & Z[4,3], & Z[4,4] \\
Z[5,0], & Z[5,1], & Z[5,2], & Z[5,3], & Z[5,4], & Z[5,5] \\
Z[6,0], & Z[6,1], & Z[6,2], & Z[6,3], & Z[6,4], & Z[6,5] \\
Z[7,0], & Z[7,1], & Z[7,2], & Z[7,3], & Z[7,4], & Z[7,5]
\end{array}
$$

б) Предпочтительный порядок обращений

$$
\begin{array}{llllll}
[0,0] \\
[0,1], & [1,1] \\
[0,2], & [1,2], & [2,2] \\
[0,3], & [1,3], & [2,3], & [3,3] \\
[0,4], & [1,4], & [2,4], & [3,4], & [4,4] \\
[0,5], & [1,5], & [2,5], & [3,5], & [4,5], & [5,5] \\
[0,6], & [1,6], & [2,6], & [3,6], & [4,6], & [5,6] \\
[0,7], & [1,7], & [2,7], & [3,7], & [4,7], & [5,7]
\end{array}
$$

в) Предпочтительный порядок итераций

Рис. 11.14. Переупорядочение обращений к элементам массива и итераций во вложенности циклов

екция многогранника на пространство меньшей размерности интуитивно представляет собой тень, отбрасываемую объектом на это пространство. Проекция двумерного пространства итераций на рис. 11.11 на ось i представляет собой вертикальный отрезок от 0 до 5, а проекция на ось j — горизонтальный отрезок от 0 до 7. При проекции трехмерного объекта вдоль оси z на двумерную плос-

кость xy мы устраняем переменную z, теряем высоту отдельных точек и получаем двумерный отпечаток объекта на плоскости xy.

Генерация границ циклов — всего лишь одно из множества применений проекций. Формально проекция может быть определена следующим образом. Пусть S — n-мерный многогранник. Проекция S на первые m его измерений представляет собой множество точек $(x_1, x_2, \ldots, x_m)$, таких, что для некоторых $x_{m+1}, x_{m+2}, \ldots, x_n$ вектор $[x_1, x_2, \ldots, x_n]$ принадлежит S. Вычислить проекцию можно при помощи алгоритма *исключения Фурье–Моцкина*.

Алгоритм 11.11. Исключение Фурье–Моцкина

Вход: многогранник S с переменными $x_1, x_2, \ldots, x_n$, т.е. S представляет собой множество линейных ограничений, включающих переменные x_i. Одна из переменных, x_m, является исключаемой.

Выход: многогранник S' с переменными $x_1, \ldots, x_{m-1}, x_{m+1}, \ldots, x_n$ (т.е. со всеми переменными S за исключением x_m), представляющий собой проекцию S на измерения, отличные от m.

Метод: пусть C — все ограничения в S, включающие x_m. Выполняем следующее.

1. Для каждой пары нижних и верхних границ x_m в C, таких как

$$L \leqslant c_1 x_m$$
$$c_2 x_m \leqslant U$$

   создаем новое ограничение

$$c_2 L \leqslant c_1 U$$

   Заметим, что c_1 и c_2 представляют собой целые числа, в то время как L и U могут быть выражениями с переменными, отличными от x_m.

2. Если целые c_1 и c_2 имеют общий делитель, делим на него обе стороны неравенства.

3. Если новые неравенства являются несовместимыми, то решения для S не существует, т.е. многогранники S и S' являются пустыми множествами.

4. S' представляет собой множество ограничений $S-C$ плюс все ограничения, сгенерированные на шаге 2.

Заметим кстати, что, если x_m имеет u нижних границ и v верхних, исключение x_m приведет к uv неравенствам, но не более. □

Ограничения, добавляемые на первом шаге алгоритма 11.11, соответствуют импликации ограничений C на остальные переменные системы. Следовательно, в S' существует решение тогда и только тогда, когда существует как минимум одно

соответствующее решение в S. Диапазон x_m для данного решения S' может быть получен путем замены всех переменных, кроме x_m, их фактическими значениями в ограничениях C.

Пример 11.12. Рассмотрим неравенства, определяющие пространство итераций на рис. 11.11. Предположим, что мы хотим применить исключение Фурье–Моцкина для получения проекции двумерного пространства на измерение j, избавляясь от измерения i. Для i существует одна нижняя граница, $0 \leqslant i$, и две верхние, $i \leqslant j$ и $i \leqslant 5$. Это приводит нас к генерации двух ограничений: $0 \leqslant j$ и $0 \leqslant 5$. Последнее неравенство тривиально и его можно игнорировать. Первое же дает нижнюю границу j, а исходная верхняя граница $j \leqslant 7$ дает новую верхнюю границу. □

Генерация границ циклов

Теперь, когда мы определили исключение Фурье–Моцкина, получение алгоритма для генерации границ цикла для сканирования выпуклого многоугольника (алгоритм 11.13) оказывается совсем простым делом. Мы вычисляем границы цикла в порядке от внутреннего к внешним циклам. Все неравенства, включающие индексные переменные внутренних циклов, записываются как нижние или верхние границы этих переменных. Затем мы исключаем измерение, представляющее наиболее глубоко вложенный цикл, и получаем многогранник с размерностью, на единицу меньшей исходной. Мы повторяем эти действия для всех индексных переменных.

Алгоритм 11.13. Вычисление границ для заданного порядка переменных

Вход: выпуклый многогранник S с переменными $v_1, \ldots, v_n$.

Выход: множества нижних границ L_i и верхних границ U_i для каждой переменной v_i, выраженные только с использованием переменных v_j, где $j < i$.

Метод: алгоритм, приведенный на рис. 11.15. □

Пример 11.14. Применим алгоритм 11.13 для генерации границ циклов, сканирующих пространство итераций на рис. 11.11 по вертикали. Упорядочение переменных — j, i. Алгоритм генерирует следующие границы:

$$\begin{aligned} L_i &: \ 0 \\ U_i &: \ 5, j \\ L_j &: \ 0 \\ U_j &: \ 7 \end{aligned}$$

Мы должны обеспечить выполнение всех ограничений, так что граница i представляет собой $\min(5, j)$. Избыточностей в этом примере нет. □

```
Sₙ = S; /* Используем алгоритм 11.11 для поиска границ */
for ( i = n; i ⩾ 1; i − − ) {
    L_{v_i} = все нижние границы v_i в S_i;
    U_{v_i} = все верхние границы v_i в S_i;
    S_{i−1} = ограничения, которые возвращаются при применении
              алгоритма 11.11 для исключения v_i из ограничений S_i;
}
/* Устранение избыточностей */
S' = ∅;
for ( i = 1; i ⩽ n; i + + ) {
    Удаляем из L_{v_i} и U_{v_i} все границы, вытекающие из S';
    Добавляем остальные ограничения на v_i из L_{v_i} и U_{v_i} в S';
}
```

Рис. 11.15. Код для выражения границ переменных для заданного порядка переменных

11.3.6 Изменение осей

Заметим, что сканирования пространства итераций по горизонтали и вертикали, как уже говорилось, являются всего лишь двумя из гораздо большего количества способов обхода пространства. Имеется множество других возможных способов; например, можно обойти пространство итераций из примера 11.6 по диагонали, как рассматривается в примере 11.15.

Пример 11.15. Пространство итераций, показанное на рис. 11.11, может быть сканировано по диагонали, как показано на рис. 11.16. Разность между координатами j и i на каждой диагонали постоянна и составляет в данном примере от 0 до 7. Таким образом, мы определяем новую переменную $k = j - i$ и сканируем пространство итераций в лексикографическом порядке по отношению к k и j. Подставляя $i = j - k$ в неравенства, мы получим

$$\begin{array}{rcl} 0 \leqslant & j - k & \leqslant 5 \\ j - k \leqslant & j & \leqslant 7 \end{array}$$

Чтобы создать границы циклов для описанного выше порядка, можно применить алгоритм 11.13 к полученному множеству неравенств с упорядочением переменных k, j.

$$\begin{array}{ll} L_j: & k \\ U_j: & 5 + k, 7 \\ L_k: & 0 \\ U_k: & 7 \end{array}$$

$$
\begin{array}{llllll}
[0,0], & [1,1], & [2,2], & [3,3], & [4,4], & [5,5] \\
[0,1], & [1,2], & [2,3], & [3,4], & [4,5], & [5,6] \\
[0,2], & [1,3], & [2,4], & [3,5], & [4,6], & [5,7] \\
[0,3], & [1,4], & [2,5], & [3,6], & [4,7] & \\
[0,4], & [1,5], & [2,6], & [3,7] & & \\
[0,5], & [1,6], & [2,7] & & & \\
[0,6], & [1,7] & & & & \\
[0,7] & & & & &
\end{array}
$$

Рис. 11.16. Подиагональное упорядочение пространства итераций, показанного на рис. 11.11

На основании этих неравенств мы генерируем следующий код, заменяя i на $j - k$ в обращении к массиву:

```
for(k = 0; k <= 7; k++)
    for(j = k; j <= min(5+k,7); j++)
        Z[j,j-k] = 0;
```

□

В общем случае можно изменить оси многогранника путем создания новых индексных переменных циклов, которые представляют аффинные комбинации исходных переменных, и определения упорядочения этих переменных. Сложная проблема возникает при выборе корректных осей, удовлетворяющих зависимостям данных при достижении поставленной цели — параллельности и локальности. Эта проблема будет рассматриваться начиная с раздела 11.7. Пока же мы выяснили, что если оси выбраны, то, как показал пример 11.15, сгенерировать требуемый код достаточно просто.

Имеется масса порядков обхода итераций, с которыми описанный метод справиться не в состоянии. Например, мы можем пожелать сначала обойти все нечетные строки пространства итераций, а затем перейти к четным. Или начать со средины пространства и идти к его краям. Однако для приложений с аффинными функциями обращения к данным описанные здесь методы охватывают большинство желательных упорядочений итераций.

11.3.7 Упражнения к разделу 11.3

Упражнение 11.3.1. Преобразуйте каждый из следующих циклов в такой вид, в котором на каждой итерации индексная переменная увеличивается на 1.

а) `for(i = 10; i < 50; i = i + 7) X[i,i+1] = 0;`

б) `for(i = -3; i <= 10; i = i + 2) X[i] = X[i+1];`

в) `for(i = 50; i >= 10; i--) X[i] = 0;`

Упражнение 11.3.2. Изобразите или опишите пространства итераций для каждой из следующих вложенностей циклов:

а) вложенность циклов на рис. 11.17, *а*;

б) вложенность циклов на рис. 11.17, *б*;

в) вложенность циклов на рис. 11.17, *в*.

```
for (i = 1; i < 30; i++)
    for (j = i+2; j < 40-i; j++)
        X[i,j] = 0;
```

а) Вложенность циклов к упражнению 11.3.2, *а*

```
for (i = 10; i <= 1000; i++)
    for (j = i; j < i+10; j++)
        X[i,j] = 0;
```

б) Вложенность циклов к упражнению 11.3.2, *б*

```
for (i = 0; i < 100; i++)
    for (j = 0; j < 100+i; j++)
        for (k = i+j; k < 100-i-j; k++)
            X[i,j,k] = 0;
```

в) Вложенность циклов к упражнению 11.3.2, *в*

Рис. 11.17. Вложенности циклов к упражнению 11.3.2

Упражнение 11.3.3. Запишите ограничения для каждой из вложенностей циклов на рис. 11.17 в виде (11.1), т.е. укажите значения векторов **i** и **b** и матрицы **B**.

Упражнение 11.3.4. Обратите каждый из порядков вложенности циклов на рис. 11.17.

Упражнение 11.3.5. Воспользуйтесь алгоритмом исключения Фурье–Моцкина для исключения i из каждого из множеств ограничений, полученных в упражнении 11.3.3.

Упражнение 11.3.6. Воспользуйтесь алгоритмом исключения Фурье–Моцкина для исключения j из каждого из множеств ограничений, полученных в упражнении 11.3.3.

Упражнение 11.3.7. Для каждой из вложенностей циклов на рис. 11.17 перепишите код так, чтобы ось i была заменена главной диагональю, т.е. осью, направление которой определяется равенством $i = j$. Новая ось должна соответствовать внешнему циклу.

Упражнение 11.3.8. Повторите упражнение 11.3.7 для следующих замен осей:

а) замените i на $i + j$, т.е. направление оси задается линиями, для которых сумма $i + j$ представляет собой константу;

б) замените j на $i - 2j$; новая ось соответствует внешнему циклу.

! **Упражнение 11.3.9.** Пусть A, B и C – целочисленные константы, причем $C > 1$ и $B > A$. Перепишите цикл

```
for(i = A; i <= B; i = i + C)
    Z[i] = 0;
```

так, чтобы прирост переменной цикла был равен 1, а начальное ее значение было равно 0, т.е. представьте цикл в виде

```
for(j = 0; j <= D; j++)
    Z[E*j+F] = 0;
```

для некоторых D, E и F. Выразите D, E и F через A, B и C.

Упражнение 11.3.10. Запишите для приведенной ниже обобщенной вложенности циклов ограничения, определяющие пространство итераций, в матричном виде, т.е. в виде $\mathbf{Bi} + \mathbf{b} = 0$.

```
for(i = 0; i <= A; i++)
    for(j = B*i+C; j < D*i+E; j++)
```

Здесь A, B, C, D и E – константы.

Упражнение 11.3.11. Повторите упражнение 11.3.10 для обобщенной вложенности циклов с символьными целочисленными константами m и n:

```
for(i = 0; i <= m; i++)
    for(j = A*i+B; j < C*i+n; j++)
```

Как и ранее, A, B и C – определенные целочисленные константы. В векторе неизвестных должны участвовать только i, j, m и n. Вспомните также, что выходными являются только переменные i и j.

11.4 Аффинные индексы массивов

В этой главе рассматривается класс аффинных обращений к массивам, в котором каждый индекс массива представляет собой аффинное выражение от индексов циклов и символических констант. Аффинные функции предоставляют собой лаконичное отображение пространства итераций на пространство данных, упрощающее определение того, какие итерации отображаются на одни и те же данные или одни и те же строки кэша.

Аффинные функции обращения к данным могут быть представлены в матричном виде, так же, как и аффинные верхняя и нижняя границы цикла. Применение матричной записи позволяет применить стандартные методы линейной алгебры для получения разнообразной информации, например о размерности данных, к которым выполняется обращение, или итерациях, которые обращаются к одним и тем же данным.

11.4.1 Аффинные обращения к данным

Мы говорим, что обращение к массиву в цикле аффинное, если

1. границы цикла представляются аффинными выражениями от переменных охватывающих циклов и символьных констант;

2. индекс каждого измерения массива также представим в виде аффинного выражения от переменных окружающих циклов и символьных констант.

Пример 11.16. Предположим, что i и j – индексные переменные циклов, ограниченных аффинными выражениями. Примерами аффинных обращений к массивам могут служить $Z[i]$, $Z[i+j+1]$, $Z[0]$, $Z[i,i]$ и $Z[2i+1, 3*j-10]$. Еще одним примером может служить $Z[3*n, n-j]$, где n – символическая константа вложения циклов. Однако ни $Z[i*j]$, ни $Z[n*j]$ аффинными обращениями не являются. □

Каждое аффинное обращение к массиву можно описать двумя матрицами и двумя векторами. Первая пара "матрица – вектор" (**B** и **b**) описывает пространство итераций для данных обращений, как в неравенстве (11.1). Вторая пара, которую мы обозначим как **F** и **f**, представляет функции от индексных переменных циклов, которые возвращают индекс (или индексы) массива, используемые в качестве различных измерений при обращениях к массиву.

Формально мы представляем обращения к массиву во вложенности циклов, которая использует вектор индексных переменных **i**, в виде четверки $\mathcal{F} = \langle \mathbf{F}, \mathbf{f}, \mathbf{B}, \mathbf{b} \rangle$. Она отображает вектор **i** в границах

$$\mathbf{Bi} + \mathbf{b} \geqslant \mathbf{0}$$

на элемент массива

$$\mathbf{Fi} + \mathbf{f}$$

Пример 11.17. На рис. 11.18 показаны некоторые распространенные обращения к массивам, выраженные в матричной форме. Циклами индексов являются i и j, образующие вектор **i**. X, Y и Z — одно-, двух- и трехмерные массивы соответственно.

Обращение	Аффинное выражение
`X[i-1]`	$\begin{bmatrix} 1 & 0 \end{bmatrix} \begin{bmatrix} i \\ j \end{bmatrix} + \begin{bmatrix} -1 \end{bmatrix}$
`Y[i,j]`	$\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} i \\ j \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \end{bmatrix}$
`Y[j,j+1]`	$\begin{bmatrix} 0 & 1 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} i \\ j \end{bmatrix} + \begin{bmatrix} 0 \\ 1 \end{bmatrix}$
`Y[1,2]`	$\begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} i \\ j \end{bmatrix} + \begin{bmatrix} 1 \\ 2 \end{bmatrix}$
`Z[1,i,2*i+j]`	$\begin{bmatrix} 0 & 0 \\ 1 & 0 \\ 2 & 1 \end{bmatrix} \begin{bmatrix} i \\ j \end{bmatrix} + \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}$

Рис. 11.18. Некоторые обращения к массивам и их матричное представление

Первое обращение, $X[i-1]$, представлено с помощью матрицы **F** размером 1×2 и вектора **f** длиной 1. Заметим, что при выполнении матричного умножения и суммирования с вектором **f** мы получаем единственную функцию $i - 1$, которая представляет собой формулу для обращения к одномерному массиву X. Третье обращение, $Y[j, j+1]$, после матричных умножения и сложения дает пару функций $(j, j+1)$, которые являются индексами двух размерностей при обращении к массиву.

Наконец, взглянем на четвертое обращение — $Y[1,2]$. Это константное обращение, так что ничего удивительного, что **F** представляет собой нулевую матрицу. Соответственно, вектор индексов циклов **i** в функции обращения не появляется.□

11.4.2 Аффинное и неаффинное обращения на практике

В численных программах встречаются некоторые распространенные шаблоны обращения к данным, не являющиеся аффинными. Важным примером служат

программы, работающие с разреженными матрицами. Один из методов хранения разреженных матриц заключается в хранении ненулевых элементов в векторе, а вспомогательный массив индексов используется для информации о том, где начинаются строки и какие столбцы содержат ненулевые значения. Для работы с такими данными используется косвенное обращение к массивам. Обращение такого вида, такое как $X[Y[i]]$, является неаффинным обращением к массиву X. Если разреженность регулярная, как в ленточной матрице, в которой ненулевые элементы располагаются вокруг диагонали, то для обращения к ним могут использоваться плотные массивы, представляющие подобласти с ненулевыми элементами. В этом случае обращение может быть аффинным.

Другим распространенным примером неаффинного доступа являются линеаризованные массивы. Программисты иногда используют линейные массивы для хранения логически многомерных объектов. Одной из причин такого решения может оказаться неизвестность различных измерений массива во время компиляции. Обращение, которое выглядит как $Z[i,j]$, может быть выражено в виде $Z[i*n+j]$, в котором участвует квадратичная функция (напомним, что символьные константы рассматриваются как переменные). Линейное обращение можно преобразовать в многомерное, если каждое обращение может быть разложено на отдельные измерения с гарантией, что ни один из компонентов не выйдет за границы массива. Наконец, заметим, что для преобразования некоторых неаффинных обращений в аффинные может использоваться анализ переменных индукции, как показано в примере 11.18.

Пример 11.18. Код

```
j = n;
for (i = 0; i <= n; i++) {
    Z[j] = 0;
    j = j+2;
}
```

может быть переписан как

```
j = n;
for (i = 0; i <= n; i++) {
    Z[n+2*i] = 0;
}
```

Это делает обращение к массиву Z аффинным. □

11.4.3 Упражнения к разделу 11.4

Упражнение 11.4.1. Для каждого из приведенных обращений к массиву приведите вектор **f** и матрицу **F**, описывающие эти обращения. Считается, что вектор индексов **i** имеет вид $i, j, \ldots$ и что все индексы циклов имеют аффинные границы.

а) $X[2*i+3, 2*j-i]$.

б) $Y[i-j, j-k, k-i]$.

в) $Z[3, 2*j, k-2*i+1]$.

11.5 Повторное использование данных

Из функций обращения к массивам мы получаем два вида информации, полезной при решении задачи оптимизации локальности и распараллеливания.

1. *Повторное использование данных* (data reuse). При оптимизации локальности необходимо определить множества итераций, которые обращаются к одним и тем же данным или одной и той же строке кэша.

2. *Зависимости данных.* Для корректности преобразований распараллеливания и оптимизации локальности следует определить *все* зависимости данных в коде. Вспомним, что два (не обязательно различных) обращения связаны зависимостями через данные (для краткости — зависимостями данных), если экземпляры обращений могут обращаться к одной и той же ячейке памяти и при этом как минимум одно из них представляет собой запись в память.

Во многих случаях, когда мы выявляем итерации, использующие одни и те же данные, между ними обнаруживаются и зависимости данных.

При наличии зависимостей данных очевидно, что повторно используются одни и те же данные. Например, в матричном умножении один и тот же элемент выходного массива записывается $O(n)$ раз. Операции записи должны выполняться в исходном порядке;[3] о повторном обращении к данным в этом случае можно говорить потому, что мы можем выделить регистр для хранения одного элемента выходного массива в процессе вычисления этого элемента.

Однако не все повторные использования могут быть задействованы в оптимизации локальности; ниже приведен пример, иллюстрирующий это утверждение.

Пример 11.19. Рассмотрим цикл

```
for (i = 0; i < n; i++)
    Z[7*i+3] = Z[3*i+5];
```

[3] Здесь имеется один тонкий момент. Из-за коммутативности сложения мы должны получить один и тот же результат независимо от порядка суммирования. Однако это слишком частный случай. В общем случае определение того, что вычисление представляет собой последовательность арифметических шагов, за которыми следует запись результата, слишком сложна для компилятора. Мы не можем полагаться на применение алгебраических правил для безопасного переупорядочения кода.

Мы видим, что на каждой итерации цикл выполняет запись в разные места, так что между различными операциями записи нет зависимостей данных по записи и повторных использований данных. Цикл считывает элементы 5, 8, 11, 14, 17, ... и записывает элементы 3, 10, 17, 24, ... Таким образом, чтение и запись обращаются к одним и тем же элементам 17, 38, 59 и т.д., т.е. целые числа вида $17 + 21j$, $j = 0, 1, 2, \ldots$ могут быть записаны как в виде $7i_1 + 3$, так и в виде $3i_2 + 5$ для некоторых целых i_1 и i_2. Однако такое повторное использование встречается крайне редко и воспользоваться им очень сложно, если вообще возможно. □

Анализ зависимостей данных отличается от анализа повторного использования тем, что одно из обращений при исследовании зависимостей данных должно быть обращением для записи. Очень важно, чтобы поиск зависимостей данных был корректным и точным. Чтобы быть корректным, он должен найти все зависимости и не должен найти ложные зависимости, которые могут привести к излишнему последовательному выполнению.

При повторном использовании данных все, что нам надо, — найти места, где сосредоточивается большинство повторных использований, которыми можно воспользоваться для повышения эффективности программы. Эта задача существенно проще, так что мы рассмотрим ее в этом разделе, а с зависимостями данных разберемся в следующем. Мы упростим анализ повторного использования, игнорируя границы циклов, так как они редко влияют на повторное использование. Воспользоваться повторным использованием можно при помощи аффинного разбиения среди экземпляров обращений к одному и тому же массиву и обращений с одной и той же *матрицей коэффициентов* (которую мы обычно обозначаем как $\mathbf{F}$ в аффинной индексной функции). Как было показано выше, шаблоны обращений наподобие $7i+3$ и $3i+5$ не представляют интереса с точки зрения повторного использования.

11.5.1 Типы повторных использований

Начнем с примера 11.20, который демонстрирует различные типы повторных использований. Далее мы должны четко отличать обращение как команду программы, например `x = Z[i,j]`, от многократного выполнения этой команды, как, например, при выполнении вложенности циклов. Чтобы подчеркнуть это отличие, говоря об инструкции самой по себе, мы будем говорить о *статическом обращении* (static access), в то время как различные итерации инструкции при выполнении вложенности циклов будем называть *динамическим обращением* (dynamic access).

Повторные использования можно классифицировать как *собственные* (self) и *групповые* (group). Если итерации повторно используют одни и те же данные из одного и того же статического обращения, мы говорим о собственном по-

вторном использовании. Если же они поступают от разных обращений, то мы говорим о групповом повторном использовании. Повторное использование является *временны́м*, если обращение происходит к одной и той же ячейке памяти, и *пространственным*, если обращение выполняется к одной строке кэша.

Пример 11.20. Рассмотрим следующую вложенность циклов:

```
float Z[n];
for (i = 0; i < n; i++)
    for (j = 0; j < n; j++)
        Z[j+1] = (Z[j] + Z[j+1] + Z[j+2])/3;
```

Обращения $Z[j]$, $Z[j+1]$ и $Z[j+2]$ представляют собственные пространственные повторные использования, поскольку последовательные итерации одного и того же доступа обращаются к последовательным элементам массива. Скорее всего, последовательные элементы располагаются в одной и той же строке кэша. Кроме того, все они обладают собственно-временны́м повторным использованием, поскольку при каждой итерации внешнего цикла выполняется обращение к одним и тем же элементам. Они также имеют одинаковую матрицу коэффициентов, а значит, обладают групповым повторным использованием. Между различными обращениями имеется групповое повторное использование, как временно́е, так и пространственное. Хотя всего в этом коде $4n^2$ обращений, если суметь воспользоваться повторными использованиями, нам потребуется только около n/c загрузок строк в кэш, где c — количество слов в строке кэша. Множитель n удаляется благодаря собственно-пространственному повторному использованию, множитель 4 — из-за группового повторного использования, а множитель c связан с пространственной локальностью. □

Далее мы рассмотрим возможность применения линейной алгебры для получения информации о повторном использовании из аффинных обращений к массиву. Нас интересует не только поиск потенциальной экономии, но и то, какие именно итерации повторно используют данные, чтобы их можно было переупорядочить и разместить поближе друг к другу и чтобы в полной мере воспользоваться преимуществами, предоставляемыми повторными использованиями.

11.5.2 Собственные повторные использования

Если суметь воспользоваться собственными повторными использованиями, можно получить существенную экономию обращений к памяти. Если данные, к которым выполняется статическое обращение, имеют k измерений, а доступ вложен в цикл глубиной d, где $d > k$, то одни и те же данные могут быть повторно использованы n^{d-k} раз, где n — количество итераций в каждом цикле. Например, если цикл с глубиной вложенности 3 обращается к одному столбцу массива, то потенциально множитель, определяющий экономию обращений,

равен n^2. Оказывается, что размерность обращения соответствует концепции *ранга* матрицы коэффициентов обращения, и найти итерации, которые обращаются к одним и тем же ячейкам, можно путем поиска *нуль-пространства* (null space) матрицы, как поясняется далее.

Ранг матрицы

Ранг матрицы **F** равен наибольшему количеству линейно независимых столбцов (или строк). Множество векторов называется *линейно независимым*, если ни один из них не может быть записан как линейная комбинация конечного количества других векторов из этого множества.

Пример 11.21. Рассмотрим матрицу

$$\begin{bmatrix} 1 & 2 & 3 \\ 5 & 7 & 9 \\ 4 & 5 & 6 \\ 2 & 1 & 0 \end{bmatrix}$$

Вторая строка матрицы представляет собой сумму первой и третьей строк, а четвертая — разность третьей и удвоенной первой строк. Однако первая и третья строки линейно независимы — одна из них не кратна другой. Таким образом, ранг матрицы равен 2.

Тот же вывод можно сделать и при рассмотрении столбцов. Третий столбец представляет собой разность между удвоенным вторым и первым столбцами. С другой стороны, любые два столбца линейно независимы. Таким образом, мы делаем вывод о том, что ранг матрицы равен 2. □

Пример 11.22. Рассмотрим обращения к массивам на рис. 11.18. Первое обращение, $X[i-1]$, имеет размерность 1, так как ранг матрицы $\begin{bmatrix} 1 & 0 \end{bmatrix}$ равен 1 (единственная строка является линейно независимой).

Второе обращение, $Y[i,j]$, имеет размерность 2. Причина этого в том, что матрица

$$\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

имеет две независимые строки (и, конечно, два независимых столбца). Третье обращение, $Y[j,j+1]$, — одномерное, так как матрица

$$\begin{bmatrix} 0 & 1 \\ 0 & 1 \end{bmatrix}$$

имеет ранг 1. Заметим, что строки матрицы идентичны, так что только одна из них линейно независима. Интуитивно в большой квадратной матрице Y обращения

выполняются только к элементам вдоль одномерной линии, лежащей над главной диагональю.

Четвертое обращение, $Y[1,2]$, имеет размерность 0, так как матрица, состоящая только из нулей, имеет нулевой ранг. Заметим, что для такой матрицы невозможно найти ненулевую линейную сумму даже из одной строки. И наконец, последнее обращение, $Z[i, i, 2 * i + j]$, имеет размерность 2. В матрице этого обращения

$$\begin{bmatrix} 0 & 0 \\ 1 & 0 \\ 2 & 1 \end{bmatrix}$$

последние две строки линейно независимы — ни одна из них не кратна другой. Однако первая строка представляет собой линейную "сумму" двух других с коэффициентами 0. □

Нуль-пространство матрицы

Обращение во вложенности циклов глубиной d с рангом r обращается к $O(n^r)$ элементам данных в $O(n^d)$ итерациях, так что в среднем $O(n^{d-r})$ итераций должны обращаться к одному и тому же элементу массива. Какие же итерации обращаются к одним и тем же данным? Предположим, что обращение в данной вложенности циклов представлено матрично-векторной комбинацией $\mathbf{F}$ и $\mathbf{f}$. Пусть $\mathbf{i}$ и $\mathbf{i}'$ — две итерации, которые обращаются к одному и тому же элементу. Тогда $\mathbf{Fi} + \mathbf{f} = \mathbf{Fi}' + \mathbf{f}$. Переставляя члены, получим

$$\mathbf{F}(\mathbf{i} - \mathbf{i}') = \mathbf{0}$$

В линейной алгебре имеется хорошо известная концепция, которая говорит о том, когда $\mathbf{i}$ и $\mathbf{i}'$ удовлетворяют приведенному уравнению. Множество всех решений уравнения $\mathbf{Fv} = \mathbf{0}$ называется *нуль-пространством* $\mathbf{F}$. Таким образом, две итерации обращаются к одному и тому же элементу массива, если разность их индексных векторов принадлежит нуль-пространству $\mathbf{F}$.

Легко увидеть, что нулевой вектор $\mathbf{v} = \mathbf{0}$ всегда удовлетворяет уравнению $\mathbf{Fv} = \mathbf{0}$. Иначе говоря, две итерации гарантированно обращаются к одному и тому же элементу массива, если их разность равна $\mathbf{0}$, т.е. если это одна и та же итерация. Далее, очевидно, что нуль-пространство является истинным векторным пространством, т.е. если $\mathbf{Fv}_1 = \mathbf{0}$ и $\mathbf{Fv}_2 = \mathbf{0}$, то $\mathbf{F}(\mathbf{v}_1 + \mathbf{v}_2) = \mathbf{0}$ и $\mathbf{F}(c\mathbf{v}_1) = \mathbf{0}$.

Если матрица $\mathbf{F}$ имеет *полный ранг*, т.е. ее ранг равен d, то нуль-пространство $\mathbf{F}$ состоит из единственного нулевого вектора. В таком случае все итерации во вложенности циклов обращаются к разным данным. В общем случае размерность нуль-пространства равна $d - r$. Если $d > r$, то для каждого элемента имеется $(d - r)$-мерное пространство итераций, обращающихся к этому элементу.

Нуль-пространство может быть представлено его базисными векторами. k-мерное нуль-пространство представляется k независимыми векторами; любой вектор, который может быть выражен в виде линейной комбинации базисных векторов, принадлежит нуль-пространству.

Пример 11.23. Вернемся к матрице из примера 11.21:

$$\begin{bmatrix} 1 & 2 & 3 \\ 5 & 7 & 9 \\ 4 & 5 & 6 \\ 2 & 1 & 0 \end{bmatrix}$$

Мы выяснили, что ранг этой матрицы равен 2; таким образом, размерность нуль-пространства равна $3 - 2 = 1$. Чтобы найти базис нуль-пространства, который в данном случае должен быть простым ненулевым вектором длиной 3, можем считать, что это вектор $[x, y, z]$, и попытаться найти решение уравнения

$$\begin{bmatrix} 1 & 2 & 3 \\ 5 & 7 & 9 \\ 4 & 5 & 6 \\ 2 & 1 & 0 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}$$

Если умножить первые две строки на вектор неизвестных, можно получить два уравнения:

$$x + 2y + 3z = 0$$
$$5x + 7y + 9z = 0$$

Можно записать и уравнения, полученные из третьей и четвертой строк, но поскольку в матрице не имеется трех линейно независимых строк, мы знаем, что новые уравнения не дадут новых ограничений на переменные x, y и z. Например, уравнение, которое мы получим из третьей строки — $4x + 5y + 6z = 0$, — может быть получено путем вычитания первого уравнения из второго.

Мы должны исключить как можно большее количество переменных из полученных уравнений. Начнем с первого уравнения и выразим из него x: $x = -2y - 3z$. Затем подставим x во второе уравнение и получим $-3y = 6z$, или $y = -2z$. Поскольку $x = -2y - 3z$, а $y = -2z$, мы получаем $x = z$. Таким образом, вектор $[x, y, z]$ на самом деле представляет собой вектор $[z, -2z, z]$. Мы можем взять любое ненулевое значение z для того, чтобы получить единственный базисный вектор нуль-пространства. Например, выбрав $z = 1$, мы получим базисный вектор нуль-пространства, равный $[1, -2, 1]$. □

Пример 11.24. Ранги, размерности нуль-пространств и сами нуль-пространства для всех обращений из примера 11.17 показаны на рис. 11.19. Заметьте, что сумма ранга и размерности нуль-пространства во всех случаях равна глубине вложения циклов, 2. Поскольку обращения $Y[i,j]$ и $Z[1,i,2*i+j]$ имеют ранг 2, все итерации обращаются к разным элементам.

Обращение	Аффинное выражение	Ранг	Размерность нуль-пространства	Базис нуль-пространства
`X[i-1]`	$\begin{bmatrix}1 & 0\end{bmatrix}\begin{bmatrix}i\\j\end{bmatrix}+\begin{bmatrix}-1\end{bmatrix}$	1	1	$\begin{bmatrix}0\\1\end{bmatrix}$
`Y[i,j]`	$\begin{bmatrix}1 & 0\\0 & 1\end{bmatrix}\begin{bmatrix}i\\j\end{bmatrix}+\begin{bmatrix}0\\0\end{bmatrix}$	2	0	
`Y[j,j+1]`	$\begin{bmatrix}0 & 1\\0 & 1\end{bmatrix}\begin{bmatrix}i\\j\end{bmatrix}+\begin{bmatrix}0\\1\end{bmatrix}$	1	1	$\begin{bmatrix}1\\0\end{bmatrix}$
`Y[1,2]`	$\begin{bmatrix}0 & 0\\0 & 0\end{bmatrix}\begin{bmatrix}i\\j\end{bmatrix}+\begin{bmatrix}1\\2\end{bmatrix}$	0	2	$\begin{bmatrix}1\\0\end{bmatrix},\begin{bmatrix}0\\1\end{bmatrix}$
`Z[1,i,2*i+j]`	$\begin{bmatrix}0 & 0\\1 & 0\\2 & 1\end{bmatrix}\begin{bmatrix}i\\j\end{bmatrix}+\begin{bmatrix}1\\0\\0\end{bmatrix}$	2	0	

Рис. 11.19. Ранг и размерность нуль-пространств аффинных обращений

Оба обращения, $X[i-1]$ и $Y[j,j+1]$, имеют матрицы ранга 1, так что к одним и тем же элементам обращаются $O(n)$ итераций. В первом случае к одному и тому же элементу обращается целая строка в пространстве итераций. Другими словами, итерации, отличающиеся только размерностью j, обращаются к одному и тому же элементу, что кратко представлено базисом нуль-пространства $[0,1]$. В случае $Y[j,j+1]$ к одному элементу обращается столбец пространства итераций; этот факт представлен базисом нуль-пространства $[1,0]$.

Наконец, в случае $Y[1,2]$ к одному и тому же элементу обращаются все итерации. Нуль-пространство в данном случае имеет два базисных вектора — $[0,1]$ и $[1,0]$, — которые означают, что все пары итераций обращаются к одному и тому же элементу массива. □

11.5.3 Собственное пространственное повторное использование

Анализ пространственного повторного использования зависит от схемы размещения данных в матрице. Матрицы в C располагаются построчно, а в Fortran — постолбцово. Другими словами, элементы массивов $X[i, j]$ и $X[i, j + 1]$ являются соседними в языке программирования C, а в языке программирования Fortran соседними будут элементы $X[i, j]$ и $X[i + 1, j]$. Без потери общности в оставшейся части главы будем считать, что используется построчная схема размещения массивов из C.

В качестве первого приближения мы рассмотрим два элемента массива, находящихся в одной строке кэша тогда и только тогда, когда они находятся в одной строке двумерного массива. В общем случае, если массив имеет размерность d, то элементы такого массива располагаются в одной строке кэша, если они отличаются один от другого только значением индекса последнего измерения. Поскольку для типичного массива и кэша в одной строке кэша располагается множество элементов массива, наблюдается существенное ускорение при построчном доступе, несмотря на то что, строго говоря, время от времени приходится ожидать загрузки новой строки кэша.

Выявление и использование преимуществ собственного пространственного повторного использования состоит в отбрасывании последней строки из матрицы коэффициентов **F**. Если ранг получившейся *усеченной* матрицы меньше глубины вложенности циклов, то обеспечить пространственную локальность можно путем такого преобразования кода, что в наиболее глубоко вложенном цикле изменяется только последняя координата массива.

Пример 11.25. Рассмотрим последнее обращение на рис. 11.19 — $Z[1, i, 2 * i + j]$. Если удалить последнюю строку, можно получить усеченную матрицу

$$\begin{bmatrix} 0 & 0 \\ 1 & 0 \end{bmatrix}$$

Очевидно, что ранг данной матрицы — 1, так что, поскольку глубина вложенности циклов равна 2, имеется возможность обеспечить пространственное повторное использование. В нашем случае, поскольку индексной переменной внутреннего цикла является j, внутренний цикл посещает соседние элементы построчно хранящегося массива Z. Если сделать индексом внутреннего цикла переменную i, пространственное повторное использование получено не будет, так как при изменении i будут изменяться и вторая, и третья координаты массива. □

Общее правило определения наличия собственного пространственного повторного использования выглядит следующим образом. Как обычно, мы полагаем, что индексы цикла соответствуют столбцам матрицы коэффициентов в порядке,

в котором первым следует самый внешний цикл, а последним — наиболее глубоко вложенный. Тогда для наличия пространственного повторного использования в нуль-пространстве усеченной матрицы должен иметься вектор $[0, 0, \ldots, 0, 1]$. Причина этого в том, что если такой вектор есть в нуль-пространстве, то при фиксации всех индексов циклов, кроме наиболее глубоко вложенного, у всех динамических обращений в пределах выполнения внутреннего цикла будет изменяться только последний индекс массива. Если массив хранится построчно, то все эти элементы будут близки друг к другу и, возможно, будут находиться в одной строке кэша.

Пример 11.26. Заметим, что в усеченной матрице из примера 11.25 имеется транспонированный вектор-столбец $[0, 1]$. Таким образом, как упоминалось ранее, можно ожидать, что, если индексом вложенного цикла является j, будет достигнута пространственная локальность. С другой стороны, если изменить порядок циклов, так что внутренним циклом станет цикл i, то матрица коэффициентов примет вид

$$\begin{bmatrix} 0 & 0 \\ 0 & 1 \end{bmatrix}$$

Теперь $[0, 1]$ не входит в нуль-пространство этой матрицы. Нуль-пространство теперь генерируется базисным вектором $[1, 0]$. Таким образом, как и предполагалось в примере 11.25, нельзя ожидать пространственной локальности, если внутренним циклом является цикл i.

Мы, однако, должны заметить, что проверки наличия $[0, 0, \ldots, 0, 1]$ в нуль-пространстве не вполне достаточно для гарантии пространственной локальности. Предположим, например, что обращение имеет вид не $Z[1, i, 2 * i + j]$, а $Z[1, i, 2 * i + 50 * j]$. Тогда в процессе выполнения внутреннего цикла обращение будет только к каждому пятидесятому элементу и повторного использования строк кэша не будет, если они недостаточно длинны, чтобы хранить более 50 элементов. □

11.5.4 Групповое повторное использование

Групповое повторное использование вычисляется только среди обращений с одной и той же матрицей коэффициентов. Если даны два динамических обращения, $\mathbf{F}\mathbf{i}_1 + \mathbf{f}_1$ и $\mathbf{F}\mathbf{i}_2 + \mathbf{f}_2$, повторное использование одних и тех же данных требует выполнения условия

$$\mathbf{F}\mathbf{i}_1 + \mathbf{f}_1 = \mathbf{F}\mathbf{i}_2 + \mathbf{f}_2$$

или

$$\mathbf{F}(\mathbf{i}_1 - \mathbf{i}_2) = (\mathbf{f}_2 - \mathbf{f}_1).$$

Предположим, что решением этого уравнения является вектор **v**. Тогда, если **w** — любой вектор из нуль-пространства **F**, то вектор $\mathbf{v} + \mathbf{w}$ также является решением, и такие векторы охватывают все решения уравнения.

Пример 11.27. Вложенность циклов глубиной 2

```
for(i = 1; i <= n; i++)
    for(j = 1; j <= n; j++)
        Z[i,j] = Z[i-1,j];
```

содержит два обращения к массиву Z, а именно — $Z[i,j]$ и $Z[i-1,j]$. Оба эти обращения характеризуются матрицей коэффициентов

$$\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix},$$

как и обращение $Y[i,j]$ на рис. 11.19. Данная матрица имеет ранг 2, так что собственного временнóго повторного использования в данном случае нет.

Однако каждое обращение проявляет собственное пространственное повторное использование. Как говорилось в разделе 11.5.3, если мы удалим нижнюю строку матрицы, то останемся только с верхней строкой $[1,0]$, имеющей ранг 1. Поскольку в нуль-пространство этой усеченной матрицы входит вектор $[1,0]$, ожидается наличие пространственного повторного использования. Так как каждое увеличение индекса внутреннего цикла j увеличивает второй индекс массива на 1, мы обращаемся к соседним элементам массива и максимально используем каждую строку кэша.

Хотя у каждого из обращений отсутствует собственное временнóе повторное использование, заметим, что две ссылки, $Z[i,j]$ и $Z[i-1,j]$, обращаются почти к одинаковым множествам элементов. Иначе говоря, мы имеем дело с групповым временны́м повторным использованием, поскольку данные, считанные обращением $Z[i-1,j]$, те же, что и записываемые обращением $Z[i,j]$, за исключением случая $i = 1$. Этот простой шаблон применяется ко всему пространству итераций и может быть использован для повышения локальности данных в коде. Формально, без учета границ цикла, две ссылки, $Z[i,j]$ и $Z[i-1,j]$, обращаются к одним и тем же данным в итерациях (i_1, j_1) и (i_2, j_2) при условии

$$\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} i_1 \\ j_1 \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} i_2 \\ j_2 \end{bmatrix} + \begin{bmatrix} -1 \\ 0 \end{bmatrix}$$

Переписывая данное уравнение, получаем

$$\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} i_1 - i_2 \\ j_1 - j_2 \end{bmatrix} = \begin{bmatrix} -1 \\ 0 \end{bmatrix}$$

Другими словами, $j_1 = j_2$ и $i_2 = i_1 + 1$.

Заметим, что повторное использование наблюдается вдоль i-оси пространства итераций, т.е. итерация (i_2, j_2) встречается через n итераций (внутреннего цикла) после итерации (i_1, j_1). Таким образом, перед повторным использованием записанных данных проходит много времени. Эти данные могут к тому времени оказаться удаленными из кэша. Если кэш хранит две последовательные строки матрицы Z, то обращение $Z[i-1, j]$ не приводит к промаху кэша и общее количество промахов во всей вложенности циклов составляет n^2/c, где c — количество элементов в строке кэша. В противном случае количество промахов удваивается, так как оба статических обращения требуют новой строки кэша для каждых c динамических обращений. □

Пример 11.28. Предположим, что во вложенности циклов глубиной 3 с индексами i, j и k (от внешнего цикла к внутреннему) имеются два обращения:

$$A[i, j, i+j] \text{ и } A[i+1, j-1, i+j]$$

Тогда два обращения, $\mathbf{i}_1 = [i_1, j_1, k_1]$ и $\mathbf{i}_2 = [i_2, j_2, k_2]$, повторно используют один и тот же элемент при условии

$$\begin{bmatrix}1 & 0 & 0\\0 & 1 & 0\\1 & 1 & 0\end{bmatrix}\begin{bmatrix}i_1\\j_1\\k_1\end{bmatrix} + \begin{bmatrix}0\\0\\0\end{bmatrix} = \begin{bmatrix}1 & 0 & 0\\0 & 1 & 0\\1 & 1 & 0\end{bmatrix}\begin{bmatrix}i_2\\j_2\\k_2\end{bmatrix} + \begin{bmatrix}1\\-1\\0\end{bmatrix}$$

Решением этого уравнения относительно вектора $\mathbf{v} = [i_1 - i_2, j_1 - j_2, k_1 - k_2]$ является вектор $\mathbf{v} = [1, -1, 0]$, т.е. $i_1 = i_2 + 1$, $j_1 = j_2 - 1$ и $k_1 = k_2$.[4] Нуль-пространство матрицы

$$\mathbf{F} = \begin{bmatrix}1 & 0 & 0\\0 & 1 & 0\\1 & 1 & 0\end{bmatrix}$$

генерируется базисным вектором $[0, 0, 1]$, т.е. третий индекс, k, может быть любым. Таким образом, решением приведенного выше уравнения является любой вектор $[1, -1, m]$ для некоторого m. Иначе говоря, динамическое обращение к $A[i, j, i+j]$ во вложенности циклов с индексами i, j и k повторно используется не только другим динамическим обращением $A[i, j, i+j]$ с теми же индексами i и j и другим k, но и динамическим обращением $A[i+1, j-1, i+j]$ с индексами $i+1$, $j-1$ и любым k. □

[4]Интересно заметить, что хотя в данном случае решение имеется, его не будет при замене одного из третьих компонентов $i+j$ на $i+j+1$, т.е. в данном примере оба обращения работают с теми элементами массивов, которые лежат в двумерном подпространстве S, определяемом как "третий компонент равен сумме первых двух". Если же заменить $i+j$ на $i+j+1$, то ни один из элементов второго обращения не будет лежать в S, и повторное использование будет отсутствовать как таковое.

Хотя здесь мы этого и не делаем, мы можем провести аналогичные рассуждения и для группового пространственного повторного использования. Как и при рассмотрении собственного пространственного повторного использования, мы просто убираем из рассмотрения последнее измерение.

Величина повторного использования различна для разных категорий. Собственное временнóе повторное использование дает наибольшие преимущества: k-мерное нуль-пространство приводит к повторному использованию одних и тех же данных $O\left(n^k\right)$ раз. Величина собственного пространственного повторного использования ограничена длиной строки кэша. И наконец, величина группового повторного использования ограничена количеством обращений в группе.

11.5.5 Упражнения к разделу 11.5

Упражнение 11.5.1. Вычислите ранг каждой из матриц на рис. 11.20. Укажите для них максимальное множество линейно независимых столбцов и максимальное множество линейно независимых строк.

$$\begin{bmatrix} 0 & 1 & 5 \\ 1 & 2 & 6 \\ 2 & 3 & 7 \\ 3 & 4 & 8 \end{bmatrix} \qquad \begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 12 & 15 \\ 3 & 2 & 2 & 3 \end{bmatrix} \qquad \begin{bmatrix} 0 & 0 & 1 \\ 0 & 1 & 1 \\ 1 & 1 & 1 \\ 5 & 6 & 3 \end{bmatrix}$$

а) б) в)

Рис. 11.20. Вычислите ранги и нуль-пространства этих матриц

Упражнение 11.5.2. Найдите базис нуль-пространства для каждой из матриц, представленных на рис. 11.20.

Упражнение 11.5.3. Предположим, что пространство итераций имеет размерности (переменные) i, j и k. Для каждого из приведенных далее обращений опишите подпространства, которые обращаются к одному элементу массива:

а) $A[i, j, i + j]$;

б) $A[i, i + 1, i + 2]$;

! в) $A[i, i, j + k]$;

! г) $A[i - j, j - k, k - i]$.

! **Упражнение 11.5.4.** Предположим, что к построчно хранящемуся в памяти массиву A выполняются обращения из следующей вложенности циклов:

```
for(i = 0; i <= 100; i++)
    for(j = 0; j <= 100; j++)
        for(k = 0; k <= 100; k++)
            <Некоторое обращение к A>
```

Для каждого из приведенных далее обращений укажите, можно ли переписать циклы так, чтобы обращение к A демонстрировало собственное пространственное повторное использование, т.е. последовательно использовались строки кэша целиком. Покажите, как именно следует переписать циклы, если это возможно. *Примечание*: переписывание циклов может включать как их перестановку местами, так и введение новых индексов циклов. Однако изменить схему размещения данных массива вы не можете: массив не может стать постолбцовым. *Еще одно примечание*: в общем случае переупорядочение индексов может быть допустимым и недопустимым, в зависимости от критериев, которые будут рассматриваться в следующем разделе. Однако в данном случае, когда каждое обращение состоит просто в присваивании нулевого значения, вам не надо беспокоиться о влиянии переупорядочения циклов на программу с точки зрения семантики.

а) `A[i+1,i+k,j] = 0`

!! б) `A[j+k,i,i] = 0`

в) `A[i,j,k,i+j+k] = 0`

!! г) `A[i,j-k,i+j,i+k] = 0`

Упражнение 11.5.5. В разделе 11.5.3 говорилось, что пространственная локальность достигается в том случае, когда в наиболее глубоко вложенном цикле изменяется только последняя координата массива, к которому выполняется обращение. Однако это утверждение зависит от нашего предположения, что массив хранится построчно. Как будет звучать данное утверждение при постолбцовом хранении массива в памяти?

! **Упражнение 11.5.6.** В примере 11.28 мы видели, что наличие повторного использования между двумя схожими обращениями сильно зависит от конкретных выражений для координат массива. Обобщите это наблюдение для определения того, для каких функций $f(i,j)$ существует повторное использование между обращениями $A[i,j,i+j]$ и $A[i+1,j-1,f(i,j)]$.

! **Упражнение 11.5.7.** В примере 11.27 мы предположили, что если строки матрицы Z будут такими большими, что не будут помещаться в кэше, то количество промахов кэша станет больше необходимого. Если матрица Z обладает указанным свойством, как можно переписать вложенность циклов для обеспечения группового пространственного повторного использования?

11.6 Анализ зависимости данных в массивах

Распараллеливание и оптимизация локальности часто меняют порядок операций, выполняемых исходной программой. Как и в случае любой оптимизации, изменение порядка операций не должно влиять на выход программы. Поскольку в общем случае невозможно точно определить, что именно делает программа, оптимизация кода обычно использует простые, консервативные тесты, которые позволяют гарантировать, что выход программы остается неизменным: мы убеждаемся, что операции с любой ячейкой памяти выполняются в одном и том же порядке как в исходной, так и в оптимизированной программах. В этом разделе мы сосредоточимся на работе с массивами, т.е. рассматриваемыми ячейками памяти являются элементы массивов.

Два обращения — не важно, для чтения или для записи — являются *независимыми* (т.е. могут быть переупорядочены), если они обращаются к разным ячейкам памяти. Кроме того, операции чтения не изменяют состояние памяти, а потому также являются независимыми. Как и в разделе 11.5, мы говорим, что два обращения *зависимы через данные* (data dependent), если они обращаются к одной и той же ячейке памяти и хотя бы одно из них является операцией записи. Чтобы гарантировать, что модифицированная программа делает то же, что и исходная, относительный порядок выполнения каждой пары зависимых через данные операций в новой программе должен быть сохранен.

Вспомним из раздела 10.2.1, что существует три вида зависимостей через данные.

1. *Истинная зависимость.* За записью следует чтение из той же ячейки памяти.
2. *Антизависимость.* За чтением следует запись в ту же ячейку памяти.
3. *Зависимость через выход.* Выполняются две записи в одну и ту же ячейку памяти.

Ранее зависимости данных определялись для динамических обращений. Мы говорим о том, что статическое обращение зависит от другого статического обращения, если существует динамический экземпляр первого обращения, который зависит от некоторого экземпляра второго обращения.[5]

Легко видеть, как зависимости данных могут использоваться при распараллеливании. Например, если между обращениями в цикле не обнаружено никаких зависимостей, можно легко назначить каждую итерацию отдельному процессору.

[5]Вспомним о разнице между статическим и динамическим обращениями. Статическое обращение — это ссылка на массив в определенном месте программы, в то время как динамическое представляет собой одно выполнение статического обращения.

В разделе 11.7 рассматривается, как можно систематически использовать такую информацию при распараллеливании.

11.6.1 Определение зависимостей данных доступов к массивам

Рассмотрим два статических обращения к одному и тому же массиву, возможно, в разных циклах. Первое представлено функцией доступа и границами $\mathcal{F} = \langle \mathbf{F}, \mathbf{f}, \mathbf{B}, \mathbf{b} \rangle$ и находится во вложенности циклов глубиной d; второе представлено функцией доступа и границами $\mathcal{F}' = \langle \mathbf{F}', \mathbf{f}', \mathbf{B}', \mathbf{b}' \rangle$ и находится во вложенности циклов глубиной d'. Эти обращения зависимы через данные, если

1. хотя бы одно из них — обращение для записи;
2. существуют векторы $\mathbf{i} \in Z^d$ и $\mathbf{i}' \in Z^{d'}$, такие, что
   а) $\mathbf{Bi} \geqslant \mathbf{0}$;
   б) $\mathbf{B}'i' \geqslant \mathbf{0}$;
   в) $\mathbf{Fi} + \mathbf{f} = \mathbf{F}'i' + \mathbf{f}'$.

Поскольку статическое обращение обычно объединяет множество динамических обращений, имеет смысл задаться вопросом, не могут ли эти динамические обращения ссылаться на одно место в памяти? Для поиска зависимостей между экземплярами одного и того же статического обращения мы полагаем $\mathcal{F} = \mathcal{F}'$ и дополняем приведенное выше определение дополнительным ограничением $\mathbf{i} \neq \mathbf{i}'$, чтобы избежать тривиального решения.

Пример 11.29. Рассмотрим вложенность циклов единичной глубины:

```
for(i = 1; i <= 10; i++) {
    Z[i] = Z[i-1];
}
```

В этом цикле находятся два обращения — $Z[i-1]$ и $Z[i]$. Первое представляет собой чтение, а второе — запись. Чтобы найти все зависимости данных в программе, мы должны проверить наличие зависимостей как между разными обращениями для записи, так и между обращениями для записи и обращениями для чтения.

1. *Зависимости данных между* $Z[i-1]$ и $Z[i]$. За исключением первой итерации, каждая итерация считывает значение, записанное в предыдущей итерации. Математически зависимость существует, так как существуют такие целые числа i и i', что

$$1 \leqslant i \leqslant 10, 1 \leqslant i' \leqslant 10 \text{ и } i - 1 = i'$$

Существует девять решений приведенной системы ограничений: ($i = 2$, $i' = 1$), ($i = 3, i' = 2$) и т.д.

2. *Зависимости данных между обращением* $Z[i]$ *и им же.* Легко видеть, что различные итерации цикла выполняют запись в разные места памяти, т.е. зависимостей данных между экземплярами обращения для записи $Z[i]$ нет. Математически зависимости нет, так как не существует i и i', удовлетворяющих условиям

$$1 \leqslant i \leqslant 10, 1 \leqslant i' \leqslant 10, i = i' \text{ и } i \neq i'$$

Заметим, что третье условие, $i = i'$, вытекает из требования, чтобы $Z[i]$ и $Z[i']$ обращались к одному и тому же месту в памяти. Оно противоречит четвертому условию — $i \neq i'$, — которое вытекает из требования нетривиальности зависимости между разными динамическими обращениями.

Нет необходимости рассматривать зависимости данных между обращением для чтения $Z[i-1]$ и им же самим, поскольку любые два обращения для чтения независимы. □

11.6.2 Целочисленное линейное программирование

Зависимости данных требуют решения вопроса о существовании целых чисел, удовлетворяющих системам, состоящим из уравнений и неравенств. Уравнения вытекают из матричного представления обращений, а неравенства — из границ циклов. Уравнения могут быть выражены в виде неравенств: равенство $x = y$ можно заменить двумя неравенствами — $x \geqslant y$ и $y \geqslant x$.

Таким образом, зависимости данных могут быть выражены как поиск целочисленных решений, удовлетворяющих множеству линейных неравенств, т.е. как решение хорошо известной задачи *целочисленного линейного программирования*. Известно, что целочисленное линейное программирование является NP-полной задачей. Полиномиальный алгоритм для ее решения неизвестен, но имеется ряд эвристик для задач с многими переменными, причем во многих случаях они достаточно быстрые. К сожалению, такие стандартные эвристики непригодны для анализа зависимостей данных, где проблема заключается в решении большого количества небольших и простых задач целочисленного линейного программирования, а не в решении одной большой сложной задачи.

Алгоритм анализа зависимостей данных состоит из трех частей.

1. Поиск наибольшего общего делителя (НОД) для проверки существования целочисленного решения уравнений с применением теории линейных диофантовых уравнений. Если целочисленных решений не существует, нет и зависимостей данных. В противном случае мы используем уравнения

для подстановки части переменных, чтобы получить более простые неравенства.

2. Использование набора простых эвристик для работы с большим количеством типичных неравенств.

3. В редких случаях, когда эвристики не срабатывают, используется решение задачи целочисленного линейного программирования с применением метода ветвлений и границ на основе исключения Фурье–Моцкина.

11.6.3 НОД

Первая подзадача состоит в проверке существования целочисленного решения уравнений. Уравнения с условием, что их решения представляют собой целые числа, известны как *диофантовы уравнения*. В приведенном ниже примере показано, откуда берется условие целочисленности решений, и демонстрируется, что, несмотря на то что в наших примерах рассматривается по одной вложенности циклов, понятие зависимости данных может применяться и к разным циклам.

Пример 11.30. Рассмотрим следующий фрагмент кода:

```
for(i = 1; i < 10; i++) {
    Z[2*i] = ...;
}
for(j = 1; j < 10; j++) {
    Z[2*j+1] = ...;
}
```

Обращение $Z[2*i]$ выполняет запись только в четные элементы массива Z, а $Z[2*j+1]$ — в нечетные. Понятно, что зависимости данных между этими обращениями нет[6], какими бы ни были границы циклов. Можно выполнить итерации второго цикла до первого, можно чередовать итерации обоих циклов. Этот пример не настолько искусственен, как можно подумать. Примером может служить работа с массивом комплексных чисел, в котором действительные и мнимые части хранятся друг за другом.

Доказать отсутствие зависимости данных в приведенном примере можно следующим образом. Предположим, что существуют такие целые числа i и j, что $Z[2*i]$ и $Z[2*j+1]$ обращаются к одному и тому же элементу массива Z. Мы получаем диофантово уравнение

$$2i = 2j + 1$$

[6]Если, конечно, таковая не появляется из-за выражений, результаты которых записываются в массив Z и которые в исходном тексте показаны троеточиями. — *Прим. ред.*

Алгоритм Евклида

Алгоритм Евклида для поиска $\gcd(a, b)$ работает следующим образом. Будем считать, что a и b — положительные целые числа и $a \geqslant b$. Заметим, что НОД отрицательных чисел или НОД отрицательного и положительного числа тот же, что и НОД их абсолютных значений, так что мы можем считать, что все целые числа положительны.

Если $a = b$, то $\gcd(a, b) = a$. Если же $a > b$, то пусть c — остаток от деления a/b. Если $c = 0$, то b является делителем a, так что $\gcd(a, b) = b$. В противном случае вычисляем $\gcd(b, c)$; результат вычисления будет равен $\gcd(a, b)$.

Чтобы найти $\gcd(a_1, a_2, \ldots, a_n)$ при $n > 2$, воспользуемся алгоритмом Евклида для вычисления $\gcd(a_1, a_2) = c$, а затем рекурсивно вычислим $\gcd(c, a_3, a_4, \ldots, a_n)$.

Не существует целых чисел i и j, которые удовлетворяли бы данному уравнению. Доказательство этого состоит в том, что каким бы ни было целое число i, значение $2i$ всегда четно. Аналогично, каким бы ни было целое число j, значение $2j + 1$ всегда нечетно. Но никакое число не может быть одновременно и четным, и нечетным. Следовательно, уравнение не имеет целочисленных решений, а значит, нет и зависимостей между обращениями для записи и для чтения. □

Чтобы выяснить, существует ли решение линейного диофантового уравнения, нам нужна концепция *наибольшего общего делителя*, НОД (greatest common divisor — GCD), двух или более целых чисел. НОД целых чисел $a_1, a_2, \ldots, a_n$, обозначаемый как $\gcd(a_1, a_2, \ldots, a_n)$, представляет собой наибольшее целое число, на которое без остатка делятся все указанные числа. НОД может быть эффективно вычислен при помощи широко известного алгоритма Евклида.

Пример 11.31. $\gcd(24, 36, 54) = 6$, так как $24/6$, $36/6$ и $54/6$ имеют остаток от деления, равный 0, причем любое число, большее 6, будет давать по крайней мере один остаток при делении на него 24, 36 и 54. Например, 24 и 36 делятся нацело на 12, но при делении 54 на 12 получается остаток 6. □

Важность НОД заключается в следующей теореме.

Теорема 11.32. Линейное диофантово уравнение

$$a_1x_1 + a_2x_2 + \cdots + a_nx_n = c$$

имеет целочисленное решение $x_1, x_2, \ldots, x_n$ тогда и только тогда, когда $\gcd(a_1, a_2, \ldots a_n)$ является делителем c. □

Пример 11.33. В примере 11.30 мы видели, что линейное диофантово уравнение не имеет решения. Это уравнение можно записать как

$$2i - 2j = 1$$

Поскольку $\gcd(2, -2) = 2$, а 2 не является делителем 1, указанное уравнение решений не имеет.

В качестве еще одного примера рассмотрим уравнение

$$24x + 36y + 54z = 30$$

Поскольку $\gcd(24, 36, 54) = 6$, а $30/6 = 5$, решение приведенного уравнения в целых числах существует. Одним из решений является $x = -1$, $y = 0$ и $z = 1$, но, помимо него, имеется бесконечное множество иных решений. □

Первый шаг к решению задачи анализа зависимостей данных состоит в использовании для решения данных уравнений стандартного метода, такого как метод исключения Гаусса. Каждый раз при построении линейного уравнения применяется теорема 11.32 для выяснения, существует ли его решение в целых числах. Если решение существует, то оно используется для снижения количества переменных в неравенствах.

Пример 11.34. Рассмотрим два уравнения:

$$x - 2y + z = 0$$
$$3x + 2y + z = 5$$

Если взглянуть на каждое уравнение по отдельности, то они имеют решения. В первом уравнении $\gcd(1, -2, 1) = 1$ является делителем 0, а во втором $\gcd(3, 2, 1) = 1$ является делителем 5. Но если использовать первое уравнение, вывести из него $z = 2y - x$ и подставить во второе уравнение, то можно получить $2x + 4y = 5$. Но это диофантово уравнение не имеет целочисленных решений, поскольку $\gcd(2, 4) = 2$ не является делителем 5. □

11.6.4 Эвристики для решения задачи целочисленного линейного программирования

Задача анализа зависимостей данных требует решения большого количества простых задач целочисленного линейного программирования. Здесь мы рассмотрим некоторые методы работы с простыми неравенствами.

Проверка независимых переменных

Многие задачи целочисленного линейного программирования, возникающие при анализе зависимостей данных, состоят из неравенств с единственным неизвестным. Такие задачи могут быть решены простой проверкой существования целых чисел между константными верхними и нижними границами.

Пример 11.35. Рассмотрим вложенный цикл

```
for(i = 0; i <= 10; i++)
    for(j = 0; j <= 10; j++)
        Z[i,j] = Z[j+10,i+11];
```

Чтобы выяснить, имеется ли зависимость между $Z[i,j]$ и $Z[j+10,i+11]$, мы проверяем, существуют ли целые числа i, j, i' и j', такие, что

$$\begin{gathered} 0 \leqslant i, j, i', j' \leqslant 10 \\ i = j' + 10 \\ j = i' + 11 \end{gathered}$$

Проверка НОД, примененная к данным двум уравнениям, позволяет определить, что данные уравнения *могут* иметь целочисленные решения. Эти решения можно выразить как

$$i = t_1, j = t_2, i' = t_2 - 11, j' = t_1 - 10$$

для произвольных t_1 и t_2. Подстановка переменных t_1 и t_2 в линейные неравенства дает

$$\begin{aligned} 0 \leqslant & \quad t_1 & \leqslant 10 \\ 0 \leqslant & \quad t_2 & \leqslant 10 \\ 0 \leqslant & \; t_2 - 11 & \leqslant 10 \\ 0 \leqslant & \; t_1 - 10 & \leqslant 10 \end{aligned}$$

Объединяя нижние границы из двух последних неравенств с верхними границами из первых двух, мы выводим

$$\begin{gathered} 10 \leqslant t_1 \leqslant 10 \\ 11 \leqslant t_2 \leqslant 10 \end{gathered}$$

Поскольку нижняя граница t_2 оказывается больше верхней, можно сделать вывод, что целочисленного решения исходной системы не существует, а значит, нет и зависимостей данных. Этот пример показывает, что даже при наличии нескольких уравнений с несколькими переменными можно прийти к неравенствам, в которых используется по одной переменной. □

Проверка ацикличности

Еще одна простая эвристика заключается в выяснении, не существует ли переменной, ограниченной снизу или сверху константой. При некоторых условиях можно безопасно заменить переменную константой; упрощенные неравенства имеют решение тогда и только тогда, когда решение имеют исходные неравенства. В частности, предположим, что каждая нижняя граница для v_i имеет вид

$$c_0 \leqslant c_i v_i \text{ для некоторого } c_i > 0,$$

при этом верхние границы для v_i имеют вид

$$c_i v_i \leqslant c_0 + c_1 v_1 + \cdots + c_{i-1} v_{i-1} + c_{i+1} v_{i+1} + \cdots + c_n v_n,$$

где c_i неотрицательно. Тогда можно заменить переменную v_i ее наименьшим возможным целым значением. Если такой нижней границы нет, то мы просто заменяем v_i на $-\infty$. Аналогично, если все ограничения, включающие v_i, могут быть записаны в представленном выше виде, но с обратными направлениями неравенств, то мы можем заменить v_i наибольшим возможным положительным целым числом или ∞, если верхней границы не существует. Этот шаг можно повторять, чтобы упростить неравенства, а в некоторых случаях даже получить решение.

Пример 11.36. Рассмотрим следующие неравенства:

$$\begin{aligned} 1 \leqslant v_1, v_2 &\leqslant 10 \\ 0 \leqslant \quad v_3 &\leqslant 4 \\ v_2 \leqslant \quad v_1 & \\ v_1 &\leqslant v_3 + 4 \end{aligned}$$

Переменная v_1 ограничена снизу переменной v_2, а сверху — значением $v_3 + 4$. Однако v_2 ограничена снизу константой 1, а v_3 ограничена сверху константой 4. Таким образом, заменяя в неравенствах v_2 на 1, а v_3 на 4, мы получим систему

$$\begin{aligned} 1 \leqslant v_1 &\leqslant 10 \\ 1 \leqslant v_1 & \\ v_1 &\leqslant 8 \end{aligned}$$

Она легко решается методом проверки независимых переменных. □

Проверка вычетов циклов

Теперь рассмотрим ситуацию, когда переменная ограничена сверху и снизу другими переменными. В анализе зависимостей данных весьма распространены

ограничения вида $v_i \leqslant v_j + c$, которые могут быть решены при помощи упрощенной *проверки вычета цикла* (loop-residue test) Шостака (Shostak). Множество таких ограничений может быть представлено ориентированным графом, узлы которого помечены переменными. Для каждого ограничения $v_i \leqslant v_j + c$ имеется ребро из v_i в v_j, помеченное c.

Определим *вес* пути как сумму меток всех ребер, составляющих путь. Каждый путь в графе представляет собой комбинацию ограничений в системе, т.е. если существует путь от v до v' с весом c, то можно заключить, что $v \leqslant v' + c$. Наличие в графе цикла с весом c представляет ограничения $v \leqslant v + c$ для каждого узла v в цикле. Если в графе можно найти цикл с отрицательным весом, это означает, что можно получить невозможное ограничение $v < v$. В таком случае мы заключаем, что система неразрешима, а значит, зависимостей данных нет.

В проверку можно также включить ограничения вида $c \leqslant v$ и $v \leqslant c$ для переменной v и константы c. Мы вводим в систему неравенств новую фиктивную переменную v_0, которая добавляется к каждой константе, представляющей нижнюю или верхнюю границу. Конечно, переменная v_0 должна иметь значение 0, но поскольку нас интересуют только циклы, фактические значения переменных не имеют значения. Для константных границ мы заменяем

$$
\begin{aligned}
v \leqslant c \text{ на } v \leqslant v_0 + c,\\
c \leqslant v \text{ на } v_0 \leqslant v - c.
\end{aligned}
$$

Пример 11.37. Рассмотрим неравенства

$$
\begin{array}{rcl}
1 \leqslant & v_1, v_2 & \leqslant 10\\
0 \leqslant & v_3 & \leqslant 4\\
v_2 \leqslant & v_1 & \\
 & 2v_1 & \leqslant 2v_3 - 7
\end{array}
$$

Константные верхняя и нижняя границы для v_1 превращаются в $v_0 \leqslant v_1 - 1$ и $v_1 \leqslant v_0 + 10$; константные границы для v_2 и v_3 обрабатываются аналогично. Преобразуя последнее ограничение в $v_1 \leqslant v_3 - 4$, мы получаем граф, показанный на рис. 11.21. Цикл $v_1 \to v_3 \to v_0 \to v_1$ имеет вес -1, так что данная система неравенств решения не имеет. □

Запоминание при выполнении

Часто приходится неоднократно решать одни и те же задачи анализа зависимостей данных, поскольку простые шаблоны обращений часто повторяются в программе. Одним из важных методов ускорения обработки зависимостей данных является *запоминание при выполнении* (memoization). При использовании этого

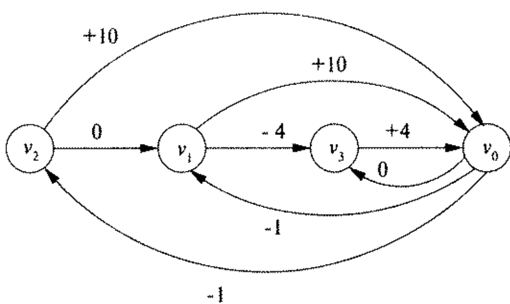


Рис. 11.21. Граф ограничений из примера 11.37

метода результаты решения задачи вносятся в специальную таблицу, и при решении новой задачи сначала выполняется поиск в таблице готового решения. Решение "с нуля" выполняется только в том случае, если такого решения еще нет в таблице.

11.6.5 Решение обобщенной задачи целочисленного линейного программирования

В этом разделе мы опишем общий подход к решению задачи целочисленного линейного программирования. Задача является NP-полной; наш алгоритм использует метод ветвей и границ, который в худшем случае может потребовать экспоненциального времени работы. Однако ситуация, когда эвристики из раздела 11.6.4 не в состоянии справиться с задачей, возникает достаточно редко, и даже когда мы вынуждены прибегнуть к описываемому здесь алгоритму, он редко требует выполнения шага ветвей и границ.

Наш подход вначале проверяет наличие рационального решения неравенств. Это классическая задача линейного программирования. Если рационального решения системы неравенств не существует, области данных, с которыми работают рассматриваемые обращения, не перекрываются и зависимостей данных нет. Если же рациональное решение существует, то мы сначала пытаемся доказать существование целочисленного решения (которое в таком случае обычно существует). Если же нам это не удается, то мы разбиваем многогранник, ограниченный неравенствами, на два меньших и решаем задачу рекурсивно.

Пример 11.38. Рассмотрим следующий простой цикл:

```
for(i = 1; i < 10; i++)
    Z[i] = Z[i+10];
```

Обращение $Z[i]$ работает с элементами $Z[1], \ldots, Z[9]$, а обращение $Z[i+10]$ — с элементами $Z[11], \ldots, Z[19]$. Эти диапазоны не перекрываются, а следова-

тельно, зависимостей данных нет. Более формально мы должны показать, что не существует двух динамических обращений i и i', таких, что $1 \leqslant i \leqslant 9$, $1 \leqslant i' \leqslant 9$ и $i = i' + 10$. Если бы такие целые числа i и i' существовали, то мы могли бы подставить $i' + 10$ вместо i и получить ограничения на i': $1 \leqslant i' \leqslant 9$ и $1 \leqslant i' + + 10 \leqslant 9$. Однако из $i' + 10 \leqslant 9$ вытекает $i' \leqslant -1$, что противоречит ограничению $1 \leqslant i'$. Таким образом, искомых целых чисел i и i' не существует. □

Алгоритм 11.39 описывает, как определить, может ли быть найдено целочисленное решение множества линейных неравенств на основе алгоритма исключения Фурье–Моцкина.

Алгоритм 11.39. Решение задачи целочисленного линейного программирования методом ветвей и границ

Вход: выпуклый многогранник S_n над переменными $v_1, \ldots, v_n$.

Выход: "да", если S_n имеет целочисленное решение; "нет" в противном случае.

Метод: выполнить алгоритм, представленный на рис. 11.22. □

1) Применить к S_n алгоритм 11.11 для переменных $v_n, v_{n-1}, \ldots, v_1$
   в указанном порядке
2) Пусть S_i — многогранник, полученный после исключения v_{i+1}
   для $i = n-1, n-2, \ldots, 0$;
3) **if** $S_0 = \varnothing$ **return** "нет";
   /* Рационального решения не существует, если S_0 (включающее
   только константы) содержит неудовлетворимые ограничения */
4) **for** $(i = 1; i \leqslant n; i++)$ {
5) **if** (S_i не включает целое значение) **break**;
6) Выбираем c_i, целое значение в средине диапазона v_i в S_i;
7) Изменяем S_i, заменяя v_i на c_i;
8) }
9) **if** $(i == n+1)$ **return** "да";
10) **if** $(i == 1)$ **return** "нет";
11) Пусть нижняя и верхняя границы v_i в S_i представляют собой l_i и u_i
    соответственно;
12) Рекурсивно применяем данный алгоритм к $S_n \cup \{v_i \leqslant \lfloor l_i \rfloor\}$
    и к $S_n \cup \{v_i \geqslant \lceil u_i \rceil\}$;
13) **if** (любой из вызовов алгоритмов вернул "да") **return** "да"
    else return "нет";

Рис. 11.22. Поиск целочисленного решения системы неравенств

В строках 1–3 алгоритм пытается найти рациональное решение неравенств. Если рационального решения не существует, не существует и целочисленного ре-

шения. Если же рациональное решение найдено, значит, неравенства определяют непустой многогранник. Относительно редко в таком многограннике нет ни одной целой точки — это означает, что многогранник должен быть очень тонким вдоль одного измерения и располагаться между целыми точками.

Строки 4–9 пытаются быстро определить наличие целочисленного решения. Каждый шаг алгоритма исключения Фурье–Моцкина дает многогранник, у которого на одно измерение меньше, чем у предыдущего. Мы рассматриваем многогранники в обратном порядке, начиная с многогранника с одним измерением и назначая этой переменной целочисленное решение, находящееся примерно посредине диапазона допустимых значений. Затем это значение подставляется во все другие многогранники, уменьшая тем самым количество переменных в них на 1. Этот процесс повторяется до тех пор, пока не будут обработаны все многогранники. При этом либо будет найдено целочисленное решение, либо будет найдена переменная, для которой целочисленного решения не существует.

Если мы не в состоянии найти целочисленное значение даже для первой переменной, значит, целочисленного решения задачи не существует (строка 10). В противном случае все, что мы знаем, — это то, что нет целочисленного решения, включающего данную комбинацию выбранных целых значений; окончательный вывод о существовании целочисленного решения на основании этих данных сделать нельзя. Строки 11–13 представляют шаг алгоритма с использованием метода ветвей и границ. Если переменная v_i имеет рациональное решение, но ее целое решение не найдено, разбиваем многогранник на два. В первом требуется, чтобы v_i была целым числом, меньшим, чем найденное рациональное решение, а во втором — бо́льшим. Если ни один из многогранников на дает решения, зависимостей не существует.

11.6.6 Резюме

Мы показали, что существенная часть информации, которую компилятор может собрать из обращений к массивам, представляет собой определенные стандартные математические концепции. Пусть дана функция обращения $\mathcal{F} = \langle \mathbf{F}, \mathbf{f}, \mathbf{B}, \mathbf{b} \rangle$.

1. Размерность области данных, к которым выполняется обращение, определяется рангом матрицы $\mathbf{F}$. Размерность пространства обращений к одному и тому же месту определяется нуль-пространством $\mathbf{F}$. Итерации, разности между которыми принадлежат нуль-пространству $\mathbf{F}$, обращаются к одним и тем же элементам массива.

2. Итерации с собственным временны́м повторным использованием обращения отделяются векторами в нуль-пространстве $\mathbf{F}$. Собственное пространственное повторное использование может быть вычислено аналогично, как

ответ на вопрос, когда две итерации используют одну и ту же строку (а не один и тот же элемент). Два обращения, $\mathbf{Fi}_1 + \mathbf{f}_1$ и $\mathbf{Fi}_2 + \mathbf{f}_2$, совместно используют локальность вдоль направления $\mathbf{d}$, если $\mathbf{d}$ — частное решение уравнения $\mathbf{Fd} = (\mathbf{f}_1 - \mathbf{f}_2)$. В частности, если $\mathbf{d}$ — направление, соответствующее наиболее глубоко вложенному циклу, т.е. представляет собой вектор $[0, 0, \ldots, 0, 1]$, то в случае построчного хранения массива имеется пространственная локальность.

3. Задача анализа зависимостей данных — могут ли два обращения обращаться к одним и те же данным — эквивалентна задаче целочисленного линейного программирования. Зависимости данных имеются, если для двух функций обращения существуют такие целочисленные векторы $\mathbf{i}$ и $\mathbf{i}'$, что $\mathbf{Bi} \geqslant \mathbf{0}$, $\mathbf{B}'\mathbf{i}' \geqslant \mathbf{0}$ и $\mathbf{Fi} + \mathbf{f} = \mathbf{F}'\mathbf{i}' + \mathbf{f}'$.

11.6.7 Упражнения к разделу 11.6

Упражнение 11.6.1. Найдите НОД для следующих множеств целых чисел:

а) $\{16, 24, 56\}$;

б) $\{-45, 105, 240\}$;

! в) $\{84, 105, 180, 315, 350\}$.

Упражнение 11.6.2. Для цикла

```
for(i = 0; i < 10; i++)
    A[i] = A[10-i];
```

укажите все

а) истинные зависимости (запись, за которой следует чтение из той же ячейки памяти);

б) антизависимости (чтение, за которым следует запись той же ячейки памяти);

в) зависимости через выход (запись, за которой следует другая запись в ту же ячейку памяти).

! **Упражнение 11.6.3.** Во врезке об алгоритме Евклида имеется ряд утверждений, приведенных без доказательства. Докажите следующее.

а) Алгоритм Евклида всегда работает. В частности, $\gcd(b, c) = \gcd(a, b)$, где c — ненулевой остаток от деления a/b.

б) $\gcd(a, b) = \gcd(a, -b)$.

в) $\gcd(a_1, a_2, \ldots, a_n) = \gcd(\gcd(a_1, a_2), a_3, a_4, \ldots, a_n)$ при $n > 2$.

г) НОД является функцией от множества целых чисел, т.е. их порядок значения не имеет. Покажите выполнимость *закона коммутативности* для НОД: $\gcd(a, b) = \gcd(b, a)$. Затем докажите более сложный *закон ассоциативности*: $\gcd(\gcd(a, b), c) = \gcd(a, \gcd(b, c))$. Наконец, покажите, что из этих законов вытекает, что НОД множества чисел всегда один и тот же, независимо от порядка вычисления НОД для пар целых чисел из этого множества.

д) Если S и T — множества целых чисел, то $\gcd(S \cup T) = \gcd(\gcd(S) \cup \gcd(T))$.

! **Упражнение 11.6.4.** Найдите другое решение второго диофантового уравнения в примере 11.33.

Упражнение 11.6.5. Примените проверку независимых переменных в следующей ситуации. Вложенность циклов представляет собой

```
for (i=0; i<100; i++)
    for (j=0; j<100; j++)
        for (k=0; k<100; k++)
```

Внутри вложенности имеется присваивание, включающее обращения к массивам. Определите, имеются ли зависимости данных для каждой из следующих инструкций:

а) `A[i,j,k] = A[i+100,j+100,k+100];`

б) `A[i,j,k] = A[j+100,k+100,i+100];`

в) `A[i,j,k] = A[j-50,k-50,i-50];`

г) `A[i,j,k] = A[i+99,k+100,j]`.

Упражнение 11.6.6. В ограничениях

$$1 \leqslant x \leqslant y - 100$$
$$3 \leqslant x \leqslant 2y - 50$$

устраните x, заменяя ее константной нижней границей y.

Упражнение 11.6.7. Примените проверку вычетов циклов к следующему множеству ограничений:

$$0 \leqslant x \leqslant 99 \quad y \leqslant x - 50$$
$$0 \leqslant y \leqslant 99 \quad z \leqslant y - 60$$
$$0 \leqslant z \leqslant 99$$

Упражнение 11.6.8. Примените проверку вычетов циклов к следующему множеству ограничений:

$$\begin{array}{ll} 0 \leqslant x \leqslant 99 & y \leqslant x - 50 \\ 0 \leqslant y \leqslant 99 & z \leqslant y + 60 \\ 0 \leqslant z \leqslant 99 & x \leqslant z + 20 \end{array}$$

Упражнение 11.6.9. Примените проверку вычетов циклов к следующему множеству ограничений:

$$\begin{array}{ll} 0 \leqslant x \leqslant 99 & y \leqslant x - 100 \\ 0 \leqslant y \leqslant 99 & z \leqslant y + 60 \\ 0 \leqslant z \leqslant 99 & x \leqslant z + 50 \end{array}$$

11.7 Поиск параллельности, не требующей синхронизации

Теперь, когда у нас разработана теория аффинного обращения к массивам, повторного использования данных и зависимостей между ними, пора применить ее к распараллеливанию и оптимизации реальных программ. Как говорилось в разделе 11.1.4, при поиске параллелизма важно минимизировать взаимодействия между процессорами. Начнем с рассмотрения задачи распараллеливания приложения, когда между процессорами нет никакого взаимодействия или синхронизации. Такое ограничение может показаться чисто академическим примером — как часто можно встретить программы с таким видом параллелизма? Но на самом деле такие программы встречаются в реальности, и алгоритм для решения такой задачи оказывается весьма полезным. Кроме того, концепции, использованные при решении этой задачи, могут быть затем расширены для работы с синхронизацией и взаимодействием процессоров.

11.7.1 Вводный пример

На рис. 11.23 приведен фрагмент трансляции на C (с массивами в стиле Fortran для ясности) 5000-строчного многосеточного алгоритма для решения трехмерных уравнений Эйлера, написанного на Fortran. Бо́льшую часть времени программа тратит на выполнение небольшого количества подпрограмм наподобие показанных на рис. 11.23. Это типично для многих численных программ, которые часто состоят из множества циклов for с различными уровнями вложенности, содержащих много обращений к массивам, причем все они представляют собой аффинные выражения от индексов охватывающих циклов. Чтобы пример был обозримым, мы убрали из исходной программы строки со сходными характеристиками.

```
for (j = 2; j <= jl; j++)
   for (i = 2, i <= il, i++) {
      AP[j,i]     = ...;
      T           = 1.0/(1.0 + AP[j,i]);
      D[2,j,i]    = T*AP[j,i];
      DW[1,2,j,i] = T*DW[1,2,j,i];
   }
for (k = 3; k <= kl-1; k++)
   for (j = 2; j <= jl; j++)
      for (i = 2; i <= il; i++) {
         AM[j,i]     = AP[j,i];
         AP[j,i]     = ...;
         T           = ...AP[j,i] - AM[j,i]*D[k-1,j,i]...;
         D[k,j,i]    = T*AP[j,i];
         DW[1,k,j,i] = T*(DW[1,k,j,i] + DW[1,k-1,j,i])...;
      }
...
for (k = kl-1; k >= 2; k--)
   for (j = 2; j <= jl; j++)
      for (i = 2; i <= il; i++)
         DW[1,k,j,i] = DW[1,k,j,i] + D[k,j,i]*DW[1,k+1,j,i];
```

Рис. 11.23. Фрагмент кода многосеточного алгоритма

Код на рис. 11.23 работает со скалярной переменной T и различными массивами разной размерности. Сначала рассмотрим использование переменной T. Поскольку каждая итерация цикла использует одну и ту же переменную T, мы не можем выполнять их параллельно. Но T применяется только для хранения дважды используемых в одной и той же итерации общих подвыражений. В первом из трех вложений циклов на рис. 11.23 каждая итерация наиболее глубоко вложенного цикла записывает в T значение, которое тут же используется два раза в пределах той же итерации. Можно устранить зависимости, заменив каждое использование T выражением, стоящим справа в операции присваивания, без изменения семантики программы. Можно также заменить скаляр T массивом. Тогда каждая итерация (j, i) будет использовать собственный элемент массива $T[j, i]$.

При таком изменении вычисление каждого элемента массива в каждой инструкции присваивания зависит только от других элементов массивов с теми же значениями последних двух компонентов (соответственно j и i). Таким образом, можно сгруппировать все операции, работающие с (j, i)-м элементом массивов, в один модуль и выполнять его в исходном последовательном порядке. Такая модификация дает $(jl - 1) \times (il - 1)$ вычислительных модулей, независимых друг от друга. Заметим, что вторая и третья вложенности в исходной программе включают третий цикл с индексом k. Однако в связи с отсутствием зависимостей между

динамическими обращениями с одними и теми же значениями j и i можно безопасно выполнять циклы k внутри циклов j и i, т.е. внутри вычислительного модуля.

Знание того, что эти вычислительные модули независимы, позволяет выполнить множество преобразований этого кода. Например, вместо выполнения кода так, как указано в исходной программе, однопроцессорная система может осуществить те же вычисления путем выполнения за один раз модулей с независимыми вычислениями. Это приводит нас к коду, показанному на рис. 11.24, в котором результаты вычислений тут же используются в других вычислениях, что приводит к повышению временно́й локальности.

```
for (j = 2; j <= jl; j++)
   for (i = 2; i <= il; i++) {
      AP[j,i]         = ...;
      T[j,i]          = 1.0/(1.0 + AP[j,i]);
      D[2,j,i]        = T[j,i]*AP[j,i];
      DW[1,2,j,i]     = T[j,i]*DW[1,2,j,i];
      for (k = 3; k <= kl-1; k++) {
         AM[j,i]      = AP[j,i];
         AP[j,i]      = ...;
         T[j,i]       = ...AP[j,i] - AM[j,i]*D[k-1,j,i]...;
         D[k,j,i]     = T[j,i]*AP[j,i];
         DW[1,k,j,i] = T[j,i]*(DW[1,k,j,i] + DW[1,k-1,j,i])...;
      }
      ...
      for (k = kl-1; k >= 2; k--)
         DW[1,k,j,i] = DW[1,k,j,i] + D[k,j,i]*DW[1,k+1,j,i];
   }
```

Рис. 11.24. Результат преобразования кода, представленного на рис. 11.23

Независимые вычислительные модули могут также быть назначены разным процессорам и выполняться параллельно, без какой бы то ни было синхронизации или взаимодействия процессоров. Поскольку имеется $(jl - 1) \times (il - 1)$ независимых вычислительных модулей, можно загрузить работой до $(jl - 1) \times \times (il - 1)$ процессоров. При организации процессоров в виде двумерного массива с индексами (j, i), где $2 \leqslant j < jl$ и $2 \leqslant i < il$, SPMD-программа, выполняемая каждым из процессоров, представляет собой просто тело внутреннего цикла на рис. 11.24.

Приведенный пример иллюстрирует фундаментальный подход к поиску параллельности, не требующей синхронизации. Сначала мы разбиваем вычисления на максимально возможное количество независимых модулей. Это разбиение показывает, какие варианты планирования доступны. Затем мы некоторым обра-

зом распределяем вычислительные модули по процессорам в зависимости от их количества. Наконец мы генерируем SPMD-программу, которая выполняется на каждом из процессоров.

11.7.2 Разбиения аффинного пространства

Говорят, что вложенность циклов имеет k степеней параллельности, если в ней имеется k распараллеливаемых циклов, т.е. таких циклов, что между их различными итерациями отсутствуют зависимости данных. Например, код на рис. 11.24 имеет 2 степени параллельности. Оказывается удобно назначать операции в вычислении с k степенями параллельности массиву процессоров с k измерениями.

Изначально будем считать, что каждое измерение процессорного массива имеет столько же процессоров, сколько итераций в соответствующем цикле. После обнаружения всех независимых вычислительных модулей мы отображаем эти "виртуальные" процессоры на фактические. На практике каждый процессор отвечает за выполнение достаточно большого количества итераций, поскольку в противном случае выполняемая процессором работа не сможет компенсировать накладные расходы распараллеливания.

Мы разбиваем распараллеливаемую программу на элементарные инструкции, такие как трехадресные команды. Для каждой инструкции мы находим *разбиение аффинного пространства*, которое отображает каждый динамический экземпляр инструкции, идентифицируемый его индексом цикла, на идентификатор процессора.

Пример 11.40. Как говорилось выше, код на рис. 11.24 имеет две степени параллельности. Мы рассматриваем массив процессоров как двумерное пространство. Пусть (p_1, p_2) — идентификатор процессора в массиве. Схема распараллеливания, рассматривавшаяся в разделе 11.7.1, может быть описана простой функцией аффинного разбиения. Все инструкции в первой вложенности циклов имеют одно и то же аффинное разбиение

$$\begin{bmatrix} p_1 \\ p_2 \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} j \\ i \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

Все инструкции во второй и третьей вложенностях имеют следующее аффинное разбиение:

$$\begin{bmatrix} p_1 \\ p_2 \end{bmatrix} = \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} k \\ j \\ i \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

□

Алгоритм поиска параллельности, не требующей синхронизации, состоит из трех шагов.

1. Для каждой инструкции программы находим аффинное разбиение, максимизирующее степень параллельности. Заметим, что в общем случае в качестве вычислительного модуля рассматриваются не отдельные обращения, а инструкции. К каждому обращению в инструкции должно применяться одно и то же аффинное разбиение. Такое группирование обращений имеет смысл, поскольку между обращениями в одной инструкции практически всегда имеются зависимости.

2. Назначаем полученные независимые вычислительные модули процессорам и выбираем чередование шагов для каждого процессора. При назначении следует учитывать вопросы локальности.

3. Генерируем SPMD-программу для каждого процессора.

Далее мы рассмотрим, как найти функции аффинного разбиения, как генерировать последовательную программу, которая выполняет полученные при разбиении разделы друг за другом, и как генерировать SPMD-программу, которая выполняет разделы на разных процессорах. После того как мы рассмотрим обработку параллельности с синхронизацией в разделах 11.8–11.9.9, в разделе 11.10 мы вернемся к упомянутому выше шагу 2 и обсудим оптимизацию локальности в однопроцессорных и многопроцессорных системах.

11.7.3 Ограничения разбиений пространства

Для отсутствия взаимодействия каждая пара операций с зависимостями данных должна быть назначена одному и тому же процессору. Эти ограничения мы будем называть "ограничениями разбиений пространства". Любое отображение, удовлетворяющее этим ограничениям, создает независимые друг от друга разделы. Заметим, что эти ограничения могут быть удовлетворены, если поместить все операции в один раздел. К сожалению, в этом "решении" параллельность отсутствует как таковая. Наша цель заключается в создании максимально возможного количества независимых разделов, удовлетворяющих ограничениям разбиения пространства, т.е. операции не помещаются в один раздел, если без этого можно обойтись.

Поскольку мы ограничили себя аффинными разбиениями, вместо количества независимых модулей мы можем максимизировать степень (количество измерений) параллелизма. Иногда оказывается возможным создать больше независимых модулей при использовании *кусочно-аффинного* разбиения (piecewise affine partition). Кусочно-аффинное разбиение разделяет экземпляры одного обращения на различные множества, позволяя при этом выполнять для каждого множества свое аффинное разбиение. Однако здесь мы не будем рассматривать эту возможность.

Формально аффинное разбиение программы *не требует синхронизации* (synchronization free) тогда и только тогда, когда для каждых двух (не обязательно

различных) зависимых обращений $\mathcal{F}_1 = \langle \mathbf{F}_1, \mathbf{f}_1, \mathbf{B}_1, \mathbf{b}_1 \rangle$ в инструкции s_1, вложенной в d_1 циклов, и $\mathcal{F}_2 = \langle \mathbf{F}_2, \mathbf{f}_2, \mathbf{B}_2, \mathbf{b}_2 \rangle$ в инструкции s_2, вложенной в d_2 циклов, разбиения $\langle \mathbf{C}_1, \mathbf{c}_1 \rangle$ и $\langle \mathbf{C}_2, \mathbf{c}_2 \rangle$ инструкций s_1 и s_2 соответственно удовлетворяют следующим *ограничениям разбиения пространства*:

- для всех $\mathbf{i}_1$ из Z^{d_1} и $\mathbf{i}_2$ из Z^{d_2}, таких, что

  а) $\mathbf{B}_1\mathbf{i}_1 + \mathbf{b}_1 \geqslant \mathbf{0}$,

  б) $\mathbf{B}_2\mathbf{i}_2 + \mathbf{b}_2 \geqslant \mathbf{0}$ и

  в) $\mathbf{F}_1\mathbf{i}_1 + \mathbf{f}_1 = \mathbf{F}_2\mathbf{i}_2 + \mathbf{f}_2$,

  выполняется соотношение $\mathbf{C}_1\mathbf{i}_1 + \mathbf{c}_1 = \mathbf{C}_2\mathbf{i}_2 + \mathbf{c}_2$.

Цель алгоритма распараллеливания состоит в поиске для каждой инструкции разбиения с наивысшим рангом, удовлетворяющего приведенным ограничениям.

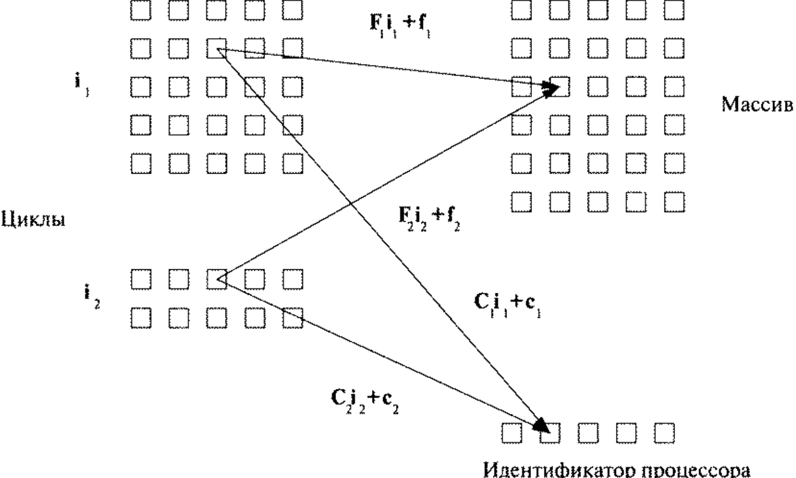


Рис. 11.25. Ограничения разбиений пространства

Показанная на рис. 11.25 диаграмма иллюстрирует суть ограничений разбиений пространства. Предположим, что есть два статических обращения в двух вложенностях циклов с индексными векторами $\mathbf{i}_1$ и $\mathbf{i}_2$. Эти обращения зависимы в том смысле, что они вместе обращаются как минимум к одному общему элементу массива и как минимум одно из них — обращение для записи. На рисунке показаны некоторые динамические обращения в двух циклах, которые обращаются к одному и тому же элементу массива согласно функциям аффинного доступа $\mathbf{F}_1\mathbf{i}_1 + \mathbf{f}_1$ и $\mathbf{F}_2\mathbf{i}_2 + \mathbf{f}_2$. Синхронизация в этом случае не будет нужна только в том случае, когда аффинные разбиения для двух статических обращений $\mathbf{C}_1\mathbf{i}_1 + \mathbf{c}_1$ и $\mathbf{C}_2\mathbf{i}_2 + \mathbf{c}_2$ назначают динамические обращения одному и тому же процессору.

Если мы выберем аффинное разбиение с рангом, равным максимальному рангу среди всех инструкций, то мы получим наибольший возможный параллелизм. Однако при таком разбиении некоторые процессоры могут простаивать в то время, когда другие процессоры выполняют инструкции, аффинные разбиения которых имеют меньший ранг. Эта ситуация может быть вполне приемлема, если время, затрачиваемое на выполнение этих инструкций, относительно мало́. В противном случае можно выбрать аффинное разбиение, ранг которого меньше максимально возможного, но при этом больше 0.

В примере 11.41 показана небольшая программа, разработанная для иллюстрации мощи этого метода. Реальные приложения обычно существенно проще, но могут иметь граничные условия, напоминающие некоторые из рассматривавшихся нами. Данный пример будет использоваться нами и далее в этой главе в качестве иллюстрации того, что в случае программы с аффинным доступом и относительно простыми ограничениями разбиения пространства решение может быть получено путем применения стандартных методов линейной алгебры и что требуемая SPMD-программа может быть механически сгенерирована на основе аффинных разбиений.

Пример 11.41. В этом примере показано, каким образом формулируются ограничения разбиения пространства для программы, состоящей из небольшой вложенности циклов с двумя инструкциями, s_1 и s_2, показанной на рис. 11.26.

```
for (i = 1; i <= 100; i++)
    for (j = 1; j <= 100; j++) {
        X[i,j] = X[i,j] + Y[i-1,j]; /* (s1) */
        Y[i,j] = Y[i,j] + X[i,j-1]; /* (s2) */
    }
```

Рис. 11.26. Вложенность циклов, демонстрирующая длинную цепочку зависимых операций

Зависимости данных этой программы показаны на рис. 11.27. Здесь каждая черная точка представляет экземпляр инструкции s_1, а каждая белая точка — экземпляр инструкции s_2. Точка с координатами (i, j) представляет экземпляр инструкции, выполняемой для данных значений индексов циклов. Заметим, однако, что в каждой паре для одного и того же значения (i, j) экземпляр s_2 находится непосредственно под экземпляром s_1, так что масштаб j (по вертикали) на рисунке больше масштаба i (по горизонтали).

Заметим, что $X[i, j]$ записывается инструкцией $s_1(i, j)$, т.е. экземпляром инструкции s_1 со значениями индексов i и j. После оно считывается инструкцией $s_2(i, j+1)$, так что инструкция $s_1(i, j)$ должна предшествовать инструкции $s_2(i, j+1)$. Вот почему на диаграмме от черных точек к белым направлены верти-

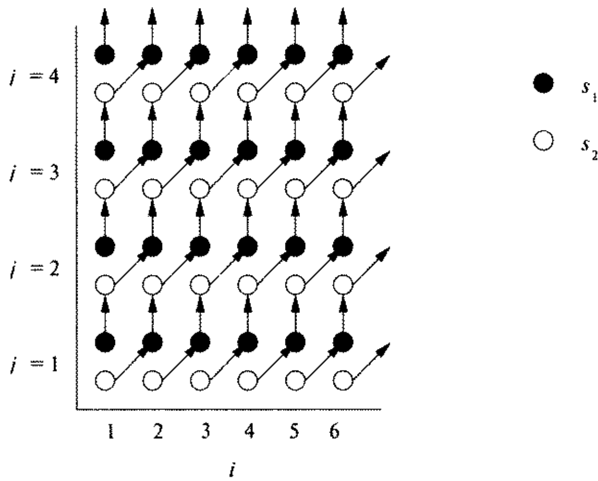


Рис. 11.27. Зависимости в коде из примера 11.41

кальные стрелки. Аналогично $Y[i,j]$ записывается инструкцией $s_2(i,j)$ и позже считывается инструкцией $s_1(i+1,j)$. Таким образом, инструкция $s_2(i,j)$ должна предшествовать инструкции $s_1(i+1,j)$, что поясняет наличие стрелок от белых точек к черным.

Из приведенной диаграммы легко увидеть, что данный код можно распараллелить без синхронизации, назначив каждую цепочку зависимых операций одному и тому же процессору. Однако написать SPMD-программу, которая реализует эту схему отображения, не так-то легко. В то время как циклы в исходной программе содержат по 100 итераций, всего имеется 200 цепочек, одна половина из которых начинается и заканчивается инструкциями s_1, а другая половина начинается и заканчивается инструкциями s_2. Длины цепочек варьируются от 1 до 100 итераций.

Поскольку имеются две инструкции, мы ищем два аффинных разбиения, по одному для каждой инструкции. Все, что нам надо, — выразить ограничения разбиений пространств для одномерных аффинных разбиений. Эти ограничения будут использованы позже методом, который пытается найти все независимые одномерные аффинные разбиения и скомбинировать их для получения многомерных аффинных разбиений. Таким образом, мы можем представить аффинное разбиение для каждой инструкции матрицей 1×2 и вектором 1×1, которые транслируют вектор индексов $[i,j]$ в единственный номер процессора. Пусть $\langle [C_{11}C_{12}], [c_1] \rangle$ и $\langle [C_{21}C_{22}], [c_2] \rangle$ представляют собой одномерные аффинные разбиения для инструкций s_1 и s_2 соответственно.

Применим шесть проверок зависимостей данных.

1. Между записями $X[i,j]$ в инструкции s_1.

2. Между записью $X[i,j]$ и чтением $X[i,j]$ в инструкции s_1.
3. Между записью $X[i,j]$ в инструкции s_1 и чтением $X[i,j-1]$ в инструкции s_2.
4. Между записями $Y[i,j]$ в инструкции s_2.
5. Между записью $Y[i,j]$ и чтением $Y[i,j]$ в инструкции s_2.
6. Между записью $Y[i,j]$ в инструкции s_2 и чтением $Y[i-1,j]$ в инструкции s_1.

Как видите, все проверки зависимостей данных простые и повторяющиеся. В коде присутствуют только две зависимости — в случае 3 между экземплярами обращений к $X[i,j]$ и $X[i,j-1]$ и в случае 6 между экземплярами обращений к $Y[i,j]$ и $Y[i-1,j]$.

Ограничения разбиений пространства, навязанные зависимостями данных между $X[i,j]$ в инструкции s_1 и $X[i,j-1]$ в инструкции s_2, можно выразить следующим образом.

Для всех (i,j) и (i',j'), таких, что

$$\begin{array}{ll} 1 \leqslant i \leqslant 100 & 1 \leqslant j \leqslant 100 \\ 1 \leqslant i' \leqslant 100 & 1 \leqslant j' \leqslant 100, \\ i = i' & j = j' - 1 \end{array}$$

имеем

$$\begin{bmatrix} C_{11} & C_{12} \end{bmatrix} \begin{bmatrix} i \\ j \end{bmatrix} + \begin{bmatrix} c_1 \end{bmatrix} = \begin{bmatrix} C_{21} & C_{22} \end{bmatrix} \begin{bmatrix} i' \\ j' \end{bmatrix} + \begin{bmatrix} c_2 \end{bmatrix}$$

Иначе говоря, первые четыре условия гласят, что (i,j) и (i',j') лежат внутри пространства итераций вложенности циклов, а последние два — что динамические обращения $X[i,j]$ и $X[i,j-1]$ относятся к одному и тому же элементу массива. Аналогично можно вывести и ограничения разбиений пространства и для обращений $Y[i-1,j]$ в инструкции s_2 и $Y[i,j]$ в инструкции s_1. □

11.7.4 Решение ограничений разбиений пространств

После того как ограничения разбиений пространства определены, для поиска аффинных разбиений, удовлетворяющих этим ограничениям, можно использовать стандартные методы линейной алгебры. Давайте рассмотрим поиск решения примера 11.41.

Пример 11.42. Аффинное разбиение для примера 11.41 можно найти следующим образом.

1. Создаем ограничения разбиений пространства, показанные в примере 11.41. Границы циклов используются только при определении зависимостей данных, но не в остальной части алгоритма.

2. В уравнениях неизвестными переменными являются i, i', j, j', C_{11}, C_{12}, c_1, C_{21}, C_{22} и c_2. Уменьшим количество неизвестных, воспользовавшись уравнениями из функций обращений: $i = i'$ и $j = j' - 1$. Мы используем метод исключения Гаусса, который из четырех переменных делает две — скажем, $t_1 = i = i'$ и $t_2 = j = j' - 1$. Уравнение для разбиения при этом превращается в

$$\begin{bmatrix} C_{11} - C_{21} & C_{12} - C_{22} \end{bmatrix} \begin{bmatrix} t_1 \\ t_2 \end{bmatrix} + \begin{bmatrix} c_1 - c_2 - C_{22} \end{bmatrix} = 0$$

3. Приведенное выше уравнение выполняется для всех комбинаций t_1 и t_2. Таким образом, должны выполняться следующие соотношения:

$$\begin{aligned} C_{11} - C_{21} &= 0 \\ C_{12} - C_{22} &= 0 \\ c_1 - c_2 - C_{22} &= 0 \end{aligned}$$

   Если мы выполним те же действия над ограничениями, связанными с обращениями к $Y[i-1, j]$ и $Y[i, j]$, мы получим

$$\begin{aligned} C_{11} - C_{21} &= 0 \\ C_{12} - C_{22} &= 0 \\ c_1 - c_2 + C_{21} &= 0 \end{aligned}$$

   Собрав все вместе и упростив, мы получим следующие соотношения:

$$C_{11} = C_{21} = -C_{22} = -C_{12} = c_2 - c_1$$

4. Ищем все независимые решения уравнений, на этом шаге включая только неизвестные из матрицы коэффициентов и игнорируя неизвестные в константных векторах. В матрице коэффициентов имеется только один независимый выбор, так что искомые аффинные разбиения могут иметь ранг не выше 1. Для простоты выбираем $C_{11} = 1$. Приравнять C_{11} к нулю нельзя, так как это приведет к матрице коэффициентов с нулевым рангом и отображению всех итераций на один и тот же процессор. Получаем $C_{21} = 1$, $C_{22} = -1$ и $C_{12} = -1$.

5. Ищем постоянные члены. Мы знаем, что их разность $c_2 - c_1$ должна быть равна -1. Для простоты выбираем $c_2 = 0$; тогда $c_1 = -1$.

Пусть p — идентификатор процессора, выполняющего итерацию (i, j). При использовании этого обозначения аффинное разбиение принимает вид

$$s_1 : \begin{bmatrix} p \end{bmatrix} = \begin{bmatrix} 1 & -1 \end{bmatrix} \begin{bmatrix} i \\ j \end{bmatrix} + \begin{bmatrix} -1 \end{bmatrix}$$

$$s_2 : \begin{bmatrix} p \end{bmatrix} = \begin{bmatrix} 1 & -1 \end{bmatrix} \begin{bmatrix} i \\ j \end{bmatrix} + \begin{bmatrix} 0 \end{bmatrix}$$

Иначе говоря, (i, j)-я итерация s_1 назначается процессору $p = i - j - 1$, а (i, j)-я итерация s_2 — процессору $p = i - j$. □

Алгоритм 11.43. Поиск не требующего синхронизации аффинного разбиения программы с наивысшим рангом

ВХОД: программа с аффинным обращением к массиву.

ВЫХОД: разбиение.

МЕТОД: выполнить следующее.

1. Найти все зависимые пары обращений в программе среди всех пар обращений $\mathcal{F}_1 = \langle \mathbf{F}_1, \mathbf{f}_1, \mathbf{B}_1, \mathbf{b}_1 \rangle$ в инструкции s_1, вложенной в d_1 циклов, и $\mathcal{F}_2 = \langle \mathbf{F}_2, \mathbf{f}_2, \mathbf{B}_2, \mathbf{b}_2 \rangle$ в инструкции s_2, вложенной в d_2 циклов. Пусть $\langle \mathbf{C}_1, \mathbf{c}_1 \rangle$ и $\langle \mathbf{C}_2, \mathbf{c}_2 \rangle$ представляют (пока что неизвестные) разбиения инструкций s_1 и s_2 соответственно. Ограничения разбиений пространства гласят, что если

$$\mathbf{F}_1\mathbf{i}_1 + \mathbf{f}_1 = \mathbf{F}_2\mathbf{i}_2 + \mathbf{f}_2,$$

то

$$\mathbf{C}_1\mathbf{i}_1 + \mathbf{c}_1 = \mathbf{C}_2\mathbf{i}_2 + \mathbf{c}_2$$

для всех $\mathbf{i}_1$ и $\mathbf{i}_2$ в соответствующих границах циклов. Обобщим область определения итераций так, чтобы она включала все $\mathbf{i}_1$ из Z^{d_1} и $\mathbf{i}_2$ из Z^{d_2}; т.е. предполагается, что границы распространяются от минус до плюс бесконечности. Это предположение имеет смысл, поскольку аффинное разбиение не может использовать тот факт, что индексная переменная может принимать только ограниченное множество целочисленных значений.

2. Для каждой пары зависимых обращений снижаем количество неизвестных в векторах индексов.

   а) Заметим, что $\mathbf{Fi} + \mathbf{f}$ представляет собой тот же вектор, что и

$$\begin{bmatrix} \mathbf{F} & \mathbf{f} \end{bmatrix} \begin{bmatrix} \mathbf{i} \\ 1 \end{bmatrix}$$

Иначе говоря, добавляя дополнительный компонент 1 в низ вектора-столбца $\mathbf{i}$, вектор-столбец $\mathbf{f}$ можно сделать дополнительным, последним столбцом матрицы $\mathbf{F}$. Таким образом, уравнение $\mathbf{F}_1\mathbf{i}_1+\mathbf{f}_1 = \mathbf{F}_2\mathbf{i}_2+$ $+ \mathbf{f}_2$ можно переписать как

$$\begin{bmatrix}\mathbf{F}_1 & -\mathbf{F}_2 & (\mathbf{f}_1 - \mathbf{f}_2)\end{bmatrix}\begin{bmatrix}\mathbf{i}_1\\ \mathbf{i}_2\\ 1\end{bmatrix} = 0$$

б) Приведенные выше уравнения в общем случае имеют несколько решений. Однако мы все равно можем использовать исключение Гаусса для решения уравнений относительно $\mathbf{i}_1$ и $\mathbf{i}_2$. Мы исключаем максимально возможное количество переменных, пока не останемся только с теми переменными, исключить которые невозможно. Получающееся в результате решение для $\mathbf{i}_1$ и $\mathbf{i}_2$ имеет вид

$$\begin{bmatrix}\mathbf{i}_1\\ \mathbf{i}_2\\ 1\end{bmatrix} = \mathbf{U}\begin{bmatrix}\mathbf{t}\\ 1\end{bmatrix}$$

Здесь $\mathbf{U}$ — верхнетреугольная матрица, а $\mathbf{t}$ — вектор свободных переменных, пробегающий по всем целым числам.

в) Можно применить использованный в п. 2, *а* способ, чтобы переписать уравнение разбиений. Подставляя вектор $(\mathbf{i}_1, \mathbf{i}_2, 1)$ в результат, полученный в п. 2, *б*, ограничения разбиений можно записать в виде

$$\begin{bmatrix}\mathbf{C}_1 & -\mathbf{C}_2 & (\mathbf{c}_1 - \mathbf{c}_2)\end{bmatrix}\mathbf{U}\begin{bmatrix}\mathbf{t}\\ 1\end{bmatrix} = 0$$

3. Избавимся от переменных, не связанных с разбиениями. Приведенные выше уравнения выполняются для всех комбинаций $\mathbf{t}$, если

$$\begin{bmatrix}\mathbf{C}_1 & -\mathbf{C}_2 & (\mathbf{c}_1 - \mathbf{c}_2)\end{bmatrix}\mathbf{U} = \mathbf{0}$$

Перепишем эти уравнения в виде $\mathbf{A}\mathbf{x} = \mathbf{0}$, где $\mathbf{x}$ — вектор всех неизвестных коэффициентов аффинного разбиения.

4. Найдем ранг аффинного разбиения и решение для матриц коэффициентов. Поскольку ранг аффинного разбиения не зависит от значений константных членов разбиения, мы убираем все неизвестные из константных векторов наподобие $\mathbf{c}_1$ и $\mathbf{c}_2$, тем самым заменяя уравнение $\mathbf{A}\mathbf{x} = \mathbf{0}$ упрощенными ограничениями $\mathbf{A}'\mathbf{x}' = \mathbf{0}$. Мы находим решение уравнения $\mathbf{A}'\mathbf{x}' = \mathbf{0}$ и выражаем его как $\mathbf{B}$ — множество базисных векторов нуль-пространства $\mathbf{A}'$.

5. Находим константные члены. Выводим по одной строке искомого аффинного разбиения из каждого базисного вектора в **B** и, используя уравнение $\mathbf{Ax} = \mathbf{0}$, получаем константные члены. □

Заметим, что в шаге 3 игнорируются ограничения, накладываемые границами циклов на переменные **t**. Ограничения в результате оказываются более строгими, так что алгоритм оказывается безопасным, т.е. мы накладываем ограничения на **C** и **c** в предположении произвольности **t**. По-видимому, могут существовать и иные решения для **C** и **c**, корректные только потому, что некоторые значения **t** невозможны. То, что мы не ищем эти иные решения, может привести к снижению степени оптимизации, но не может привести к изменениям в программе, которые изменили бы ее функциональность.

11.7.5 Простой алгоритм генерации кода

Алгоритм 11.43 генерирует аффинные разбиения, которые разделяют вычисления на независимые части. Полученные части могут быть назначены разным процессорам произвольным образом, поскольку они не зависят друг от друга. Процессору может быть назначено несколько вычислительных модулей, причем их вычисления могут чередоваться, лишь бы при этом операции в пределах каждого модуля, которые зависят друг от друга, выполнялись в требуемом порядке.

Относительно просто сгенерировать корректную программу для заданного аффинного разбиения. Мы сначала рассмотрим алгоритм 11.45 — простой метод генерации кода для одного процессора, который выполняет каждую из независимых частей последовательно. Такой код оптимизирует временну́ю локальность, поскольку обращения к массиву с несколькими использованиями весьма близки по времени. Кроме того, код может быть легко преобразован в SPMD-программу, которая выполняет каждую часть на отдельном процессоре. К сожалению, генерируемый таким образом код неэффективен; позже мы рассмотрим оптимизации, которые повышают его эффективность.

Основная идея заключается в следующем. У нас имеются границы индексных переменных вложенности циклов, а в алгоритме 11.43 мы определили разбиение для обращений некоторой инструкции s. Предположим, что мы хотим сгенерировать последовательный код, который последовательно выполняет действия каждого процессора. Мы создаем внешний цикл, который обходит все процессоры, т.е. каждая итерация этого цикла выполняет операции, назначенные одному из процессоров, соответствующему данной итерации. Исходная программа вставляется в этот цикл в качестве его тела; кроме того, добавляются проверки, гарантирующие, что каждый процессор выполняет только те операции, которые были ему назначены. Таким образом, мы гарантируем, что процессор выполняет все предназначенные ему команды и делает это в исходном последовательном порядке.

Пример 11.44. Сгенерируем код, который последовательно выполняет независимые части из примера 11.41. Исходная последовательная программа с рис. 11.26 повторена на рис. 11.28.

```
for (i = 1; i <= 100; i++)
    for (j = 1; j <= 100; j++) {
        X[i,j] = X[i,j] + Y[i-1,j]; /* (s1) */
        Y[i,j] = Y[i,j] + X[i,j-1]; /* (s2) */
    }
```

Рис. 11.28. Исходная последовательная программа, представленная на рис. 11.26 (повтор)

В примере 11.42 алгоритм аффинного разбиения обнаруживает одну степень параллелизма. Таким образом, пространство процессоров может быть представлено одной переменной p. Вспомним также, что мы выбрали аффинное разбиение так, что для всех значений индексных переменных i и j ($1 \leqslant i \leqslant 100$ и $1 \leqslant j \leqslant 100$)

1. экземпляр (i, j) инструкции s_1 назначается процессору $p = i - j - 1$;
2. экземпляр (i, j) инструкции s_2 назначается процессору $p = i - j$.

Можно сгенерировать код в три этапа.

1. Для каждой инструкции находим все идентификаторы процессоров, участвующих в вычислениях. Объединим ограничения $1 \leqslant i \leqslant 100$ и $1 \leqslant j \leqslant 100$ с одним из уравнений, $p = i-j-1$ или $p = i-j$, и исключим i и j, получив новые ограничения:
   а) $-100 \leqslant p \leqslant 98$ при использовании функции $p = i - j - 1$, которая была получена для инструкции s_1;
   б) $-99 \leqslant p \leqslant 99$ при использовании функции $p = i - j$, которая была получена для инструкции s_2.
2. Находим идентификаторы всех процессоров, участвующих в выполнении любой инструкции. После объединения указанных выше диапазонов мы получаем $-100 \leqslant p \leqslant 99$; этих границ достаточно, чтобы охватить все экземпляры обеих инструкций — и s_1, и s_2.
3. Сгенерируем код, который последовательно выполняет вычисления каждой части разбиения. Код, показанный на рис. 11.29, имеет внешний цикл, проходящий по всем идентификаторам процессоров, участвующих в вычислениях (строка 1). Для каждого идентификатора генерируются индексы

всех итераций исходной последовательной программы (строки 2 и 3), чтобы выбрать те из них, которые должен выполнить процессор p. Проверки в строках 4 и 6 обеспечивают выполнение инструкций s_1 и s_2 только тогда, когда их должен выполнять процессор p.

```
1)  for (p = -100; p <= 99; p++)
2)      for (i = 1; i <= 100; i++)
3)          for (j = 1; j <= 100; j++) {
4)              if (p == i-j-1)
5)                  X[i,j] = X[i,j] + Y[i-1,j]; /* (s1) */
6)              if (p == i-j)
7)                  Y[i,j] = X[i,j-1] + Y[i,j]; /* (s2) */
8)          }
```

Рис. 11.29. Переписанный код с рис. 11.28 с итерациями в пространстве процессоров

Несмотря на то что сгенерированный код корректен, он очень неэффективен. Во-первых, хотя каждый процессор выполняет вычисления не более чем из 99 итераций, он генерирует индексы циклов для 100×100 итераций, на порядки больше, чем это необходимо. Во-вторых, каждому суммированию во внутреннем цикле предшествует проверка, являющаяся источником накладных расходов. С неэффективностями такого рода мы будем бороться в разделах 11.7.6 и 11.7.7 соответственно. □

Хотя код на рис. 11.29 и выглядит как созданный для однопроцессорной системы, можно взять его внутренние циклы в строках 2–8 и выполнить их на 200 различных процессорах, каждый из которых имеет разное значение p — от -100 до 99. Можно разделить ответственность за выполнение внутреннего цикла и между меньшим количеством процессоров, лишь бы каждый процессор знал, за какие значения p он отвечает, и выполнял строки 2–8 для этих значений p.

Алгоритм 11.45. Генерация кода, последовательно выполняющего части разбиения программы

Вход: программа P с аффинным обращением к массиву. Каждая инструкция s в программе имеет связанные с ней границы вида $\mathbf{B}_s\mathbf{i} + \mathbf{b}_s \geqslant \mathbf{0}$, где $\mathbf{i}$ — вектор индексов циклов вложенности, в которой находится s. С s связано также разбиение $\mathbf{C}_s\mathbf{i} + \mathbf{c}_s = \mathbf{p}$, где $\mathbf{p}$ — m-мерный вектор переменных, представляющий идентификатор процессора; m — максимальный ранг разбиения среди разбиений для всех инструкций программы P.

Выход: программа, эквивалентная P, с итерациями в пространстве процессоров вместо итераций, связанных с индексами циклов.

Метод: делаем следующее.

1. Для каждой инструкции используем исключение Фурье–Моцкина, чтобы удалить из границ все индексные переменные.

2. Используем алгоритм 11.13 для определения границ идентификаторов разбиения.

3. Генерируем циклы, по одному для каждого из m измерений в пространстве процессоров. Пусть $\mathbf{p} = [p_1, p_2, \ldots, p_m]$ — вектор переменных для этих циклов, т.е. для каждого измерения в пространстве процессоров имеется одна переменная. Диапазон каждой переменной цикла p_i определяется объединением пространств разбиений для всех инструкций программы P.

Заметим, что объединение пространств разбиений не обязательно выпуклое. Чтобы алгоритм оставался простым, вместо перечисления только тех частей, в которых имеются непустые вычисления, можно установить нижнюю границу для каждого p_i равной минимальному значению среди всех нижних границ инструкций, а верхнюю — максимальному значению среди всех верхних границ. В этом случае некоторые значения $\mathbf{p}$ могут не иметь никаких операций.

Код, выполняемый каждой частью, представляет собой исходную последовательную программу. Однако каждая инструкция защищена предикатом, так что выполняются только принадлежащие части операции. □

Пример работы алгоритма 11.45 будет приведен ниже. Пока же вспомним, что получающийся таким образом код очень далек от оптимального.

11.7.6 Устранение пустых итераций

Рассмотрим первое из двух преобразований, необходимых для генерации эффективного SPMD-кода. Код, выполняемый каждым процессором, проходит по всем итерациям исходной программы и отбирает только те из них, которые должны быть выполнены этим процессором. Если код имеет k степеней параллельности, то в результате каждый процессор выполняет работу, которая на k порядков величины больше необходимой. Цель первого преобразования состоит в сокращении границ циклов для устранения всех пустых итераций.

Начнем с рассмотрения инструкций в программе по одной. Выполняемое каждой частью пространство итераций инструкции представляет собой исходное пространство итераций плюс ограничения, навязываемые аффинным разбиением. Плотные границы для каждой инструкции можно сгенерировать путем применения к новому пространству итераций алгоритма 11.13. Новый индексный вектор подобен исходному последовательному индексному вектору с добавлением идентификаторов процессоров в качестве внешних индексов. Вспомним, что алгоритм генерирует плотные границы для каждого индекса, выраженные через индексы охватывающих циклов.

После нахождения пространств итераций различных инструкций мы объединяем их цикл за циклом, получая границы как объединения границ для каждой инструкции. Некоторые циклы могут свестись к одной итерации, как проиллюстрировано в приведенном далее примере 11.46, и мы можем просто устранить такой цикл и установить индексную переменную равной значению для этой единственной итерации.

Пример 11.46. Для приведенного на рис. 11.30, *а* цикла алгоритм 11.43 создает два аффинных разбиения:

$$s_1 : p = i$$
$$s_2 : p = j$$

Алгоритм 11.45 генерирует код, показанный на рис. 11.30, *б*. Применение алгоритма 11.13 к инструкции s_1 дает границы $p \leqslant i \leqslant p$, или просто $i = p$. Аналогично для инструкции s_2 алгоритм находит $j = p$. Таким образом, мы получаем код, приведенный на рис. 11.30, *в*. Распространение копий переменных i и j приводит к устранению ненужных проверок и коду, показанному на рис. 11.30, *г*. □

Вернемся теперь к примеру 11.44 и проиллюстрируем объединение нескольких пространств итераций разных инструкций.

Пример 11.47. Уплотним границы циклов в коде из примера 11.44. Пространство итераций, выполняемое частью p для инструкции s_1, определяется следующими уравнениями и неравенствами:

$$-100 \leqslant p \leqslant 99$$
$$1 \leqslant i \leqslant 100$$
$$1 \leqslant j \leqslant 100$$
$$i - p - 1 = j$$

Применение алгоритма 11.13 приводит к ограничениям, показанным на рис. 11.31, *а*. Алгоритм 11.13 генерирует ограничение $p + 2 \leqslant i \leqslant 100 + p + 1$ из $i - p - 1 = j$ и $1 \leqslant j \leqslant 100$ и уплотняет верхнюю границу p до 98. Аналогичные границы для каждой переменной инструкции s_2 показаны на рис. 11.31, *б*.

Пространства итераций для s_1 и s_2 на рис. 11.31 похожи, но, как и следовало ожидать, исходя из рис. 11.27, некоторые из границ отличаются на единицу. Код на рис. 11.32 выполняется в пределах объединения пространств итераций. Например, для i в качестве нижней границы используется $\max(1, p + 1)$, а в качестве верхней — $\min(100, 101 + p)$. Обратите внимание, что внутренний цикл состоит из двух итераций, за исключением первого и последнего выполнений, когда выполняется только одна итерация. Таким образом, накладные расходы на генерацию индексов циклов снижаются на один порядок величины. Поскольку пространство

```
for (i=1; i<=N; i++)
    Y[i] = Z[i]; /* (s1) */
for (j=1; j<=N; j++)
    X[j] = Y[j]; /* (s2) */
```

а) Исходный код

```
for (p=1; p<=N; p++) {
    for (i=1; i<=N; i++)
        if (p == i)
            Y[i] = Z[i]; /* (s1) */
    for (j=1; j<=N; j++)
        if (p == j)
            X[j] = Y[j]; /* (s2) */
}
```

б) Результат применения алгоритма 11.45

```
for (p=1; p<=N; p++) {
    i = p;
    if (p == i)
        Y[i] = Z[i]; /* (s1) */
    j = p;
    if (p == j)
        X[j] = Y[j]; /* (s2) */
}
```

в) Результат применения алгоритма 11.13

```
for (p=1; p<=N; p++) {
    Y[p] = Z[p]; /* (s1) */
    X[p] = Y[p]; /* (s2) */
}
```

г) Окончательный код

Рис. 11.30. Код к примеру 11.46

итераций выполняется большее количество раз, чем каждая из инструкций s_1 и s_2, проверки для выбора выполняемых инструкций остаются необходимыми. □

$$\begin{aligned} j:&\quad i-p-1 \leqslant j \leqslant i-p-1\\ &\quad 1 \leqslant j \leqslant 100\\[1em] i:&\quad p+2 \leqslant i \leqslant 100+p+1\\ &\quad 1 \leqslant i \leqslant 100\\[1em] p:&\quad -100 \leqslant p \leqslant 98 \end{aligned}$$

а) Границы для инструкции s_1

$$\begin{aligned} j:&\quad i-p \leqslant j \leqslant i-p\\ &\quad 1 \leqslant j \leqslant 100\\[1em] i:&\quad p+1 \leqslant i \leqslant 100+p\\ &\quad 1 \leqslant i \leqslant 100\\[1em] p:&\quad -99 \leqslant p \leqslant 99 \end{aligned}$$

б) Границы для инструкции s_2

Рис. 11.31. Плотные границы p, i и j для рис. 11.29

```
for (p = -100; p <= 99; p++)
    for (i = max(1,p+1); i <= min(100,101+p); i++)
        for (j = max(1,i-p-1); j <= min(100,i-p); j++) {
            if (p == i-j-1)
                X[i,j] = X[i,j] + Y[i-1,j]; /* (s1) */
            if (p == i-j)
                Y[i,j] = X[i,j-1] + Y[i,j]; /* (s2) */
        }
```

Рис. 11.32. Код с рис. 11.29, улучшенный при помощи более плотных границ

11.7.7 Устранение проверок из внутреннего цикла

Второе преобразование состоит в устранении проверок из внутреннего цикла. Как видно из приведенного выше примера, проверки необходимы, если пространства итераций инструкций в цикле пересекаются, но не полностью. Чтобы избежать необходимости проверок, разобьем пространство итераций на подпространства, каждое из которых выполняет одинаковое множество инструкций. Та-

кая оптимизация требует дублирования кода и должна использоваться только для устранения проверок во внутренних циклах.

Для разделения пространства итераций с целью устранения проверок во внутренних циклах мы многократно выполняем следующие действия до тех пор, пока все проверки из внутренних циклов не будут устранены.

1. Выбираем цикл, который содержит инструкции с разными границами.

2. Разделяем цикл, используя условие, при котором некоторая инструкция исключается как минимум из одного из его компонентов. Условие выбирается среди границ различных перекрывающихся многогранников. Лучше, если все итерации некоторой инструкции находятся только в одной из полуплоскостей условия.

3. Генерируем код для каждого из пространств итераций отдельно.

Пример 11.48. Удалим проверки из кода на рис. 11.32. Инструкции s_1 и s_2 отображаются на одно и то же множество идентификаторов процессоров, за исключением граничных частей. Таким образом, пространство разбиений разделяется на три подпространства:

1. $p = -100$;

2. $-99 \leqslant p \leqslant 98$;

3. $p = 99$.

Код каждого подпространства может быть уточнен с учетом значения (или значений) p. На рис. 11.33 приведен код для каждого из трех пространств итераций.

Обратите внимание на то, что первому и третьему пространствам не требуются циклы по i и j, поскольку при конкретных значениях p, определяющих эти пространства, циклы становятся вырожденными, содержащими только одну итерацию. Например, в пространстве 1 подстановка $p = -100$ в границы циклов ограничивает i значением 1, а j — значением 100. Присваивания значений переменной p в первом и третьем пространствах представляют собой "мертвый код", который может быть удален.

Далее мы разделяем цикл с индексом i в пространстве 2. Вновь первая и последняя итерации оказываются отличными от остальных. Таким образом, мы разбиваем цикл на три подпространства:

а) $\max(1, p+1) \leqslant i < p+2$, где выполняется только инструкция s_2;

б) $\max(1, p+2) \leqslant i \leqslant \min(100, 100+p)$, где выполняются инструкции s_1 и s_2;

```
/* Пространство 1 */
p = -100;
i = 1;
j = 100;
X[i,j] = X[i,j] + Y[i-1,j]; /* (s1) */

/* Пространство 2 */
for (p = -99; p <= 98; p++)
    for (i = max(1,p+1); i <= min(100,101+p); i++)
        for (j = max(1,i-p-1); j <= min(100,i-p); j++) {
            if (p == i-j-1)
                X[i,j] = X[i,j] + Y[i-1,j]; /* (s1) */
            if (p == i-j)
                Y[i,j] = X[i,j-1] + Y[i,j]; /* (s2) */
        }

/* Пространство 3 */
p = 99;
i = 100;
j = 1;
Y[i,j] = X[i,j-1] + Y[i,j]; /* (s2) */
```

Рис. 11.33. Разбиение пространства итераций по значениям p

в) $101 + p < i \leqslant \min(101 + p, 100)$, где выполняется только инструкция s_1.

Вложенность циклов в пространстве 2 на рис. 11.33 можно, таким образом, переписать так, как показано на рис. 11.34, *а*.

На рис. 11.34, *б* показана оптимизированная программа. Мы подставили код на рис. 11.34, *а* во вложенность циклов на рис. 11.33. Кроме того, мы внесли присваивания p, i и j в обращения к массивам. При оптимизации на уровне промежуточного кода некоторые из этих присваиваний будут идентифицированы как общие подвыражения и удалены из кода обращения к массивам. □

11.7.8 Преобразования исходного кода

Мы видели, как на основе простых аффинных разбиений для каждой инструкции можно получить программы, существенно отличающиеся от исходных. Но из рассмотренных примеров не видно, как аффинные разбиения коррелируют с изменениями на уровне исходного текста. В этом разделе показано, как можно относительно легко объяснить изменения исходного кода, если разделить аффинные разбиения на серии примитивных преобразований.

```
/* Пространство (2) */
for (p = -99; p <= 98; p++) {
    /* Пространство (2a) */
    if (p >= 0) {
        i = p+1;
        j = 1;
        Y[i,j] = X[i,j-1] + Y[i,j]; /* (s2) */
    }
    /* Пространство (2b) */
    for (i = max(1,p+2); i <= min(100,100+p); i++) {
        j = i-p-1;
        X[i,j] = X[i,j] + Y[i-1,j]; /* (s1) */
        j = i-p;
        Y[i,j] = X[i,j-1] + Y[i,j]; /* (s2) */
    }
    /* Пространство (2c) */
    if (p <= -1) {
        i = 101+p;
        j = 100;
        X[i,j] = X[i,j] + Y[i-1,j]; /* (s1) */
    }
}
```

а) Разбиение пространства 2 по значениям i

```
/* Пространство (1); p = -100 */
X[1,100] = X[1,100] + Y[0,100];                     /* (s1) */

/* Пространство (2) */
for (p = -99; p <= 98; p++) {
    if (p >= 0)
        Y[p+1,1] = X[p+1,0] + Y[p+1,1];              /* (s2) */
    for (i = max(1,p+2); i <= min(100,100+p); i++) {
        X[i,i-p-1] = X[i,i-p-1] + Y[i-1,i-p-1];     /* (s1) */
        Y[i,i-p] = X[i,i-p-1] + Y[i,i-p];           /* (s2) */
    }
    if (p <= -1)
        X[101+p,100]=X[101+p,100]+Y[101+p-1,100]; /* (s1) */
}

/* Пространство (3); p = 99 */
Y[100,1] = X[100,0] + Y[100,1];                     /* (s2) */
```

б) Оптимизированный код, эквивалентный коду на рис. 11.28

Рис. 11.34. Код к примеру 11.48

Семь примитивных аффинных преобразований

Каждое аффинное преобразование может быть выражено в виде ряда примитивных аффинных преобразований, каждое из которых соответствует простому изменению на уровне исходного текста. Существует семь типов примитивных преобразований. Первые четыре из них проиллюстрированы на рис. 11.35, а последние три, известные как *унимодулярные преобразования*, показаны на рис. 11.36.

На рисунках показано по одному примеру для каждого примитива: исходный код, аффинное разбиение и код, получившийся в результате преобразования. Изображены также зависимости данных для кода до и после преобразования. Из диаграмм зависимостей данных видно, что каждый примитив соответствует простому геометрическому преобразованию и порождает относительно простое преобразование кода. Вот эти семь примитивов.

1. *Слияние*. Преобразование слияния характеризуется отображением нескольких индексов циклов исходной программы на один и тот же индекс цикла. Новый цикл объединяет инструкции из разных циклов.

2. *Расщепление*. Расщепление представляет собой преобразование, обратное слиянию. Оно отображает один и тот же индекс цикла для разных инструкций на различные индексы циклов в преобразованном коде. Таким образом, исходный цикл разбивается на несколько циклов.

3. *Реиндексирование*. Реиндексирование сдвигает динамические выполнения инструкции на постоянное количество итераций. Аффинное преобразование при этом содержит константный член.

4. *Масштабирование*. Последовательные итерации в исходной программе разносятся с использованием постоянного множителя. Соответствующее аффинное преобразование имеет положительный коэффициент, не равный 1.

5. *Реверс*. Выполнение итераций в цикле в обратном порядке. Реверс характеризуется наличием коэффициента -1 в аффинном преобразовании.

6. *Перестановка*. Перестановка внешнего и внутреннего циклов. Аффинное преобразование состоит из переставленных строк тождественной матрицы.

7. *Сдвиг*. Обход пространства итераций "под углом". Аффинное преобразование представляет собой унимодулярную матрицу с единицами на главной диагонали.

Геометрическая интерпретация распараллеливания

Приведенные аффинные преобразования, кроме слияния, порождаются путем применения алгоритма аффинного разбиения, не требующего синхронизации,

Исходный код	Разбиение	Преобразованный код

`for (i=1; i<=N; i++)`
`  Y[i] = Z[i]; /*s1*/`
`for (j=1; j<=N; j++)`
`  X[j] = Y[j]; /*s2*/`
	Слияние (fusion)
	$s_1 : p = i$
	$s_2 : p = j$
		`for (p=1; p<=N; p++){`
		`  Y[p] = Z[p];`
		`  X[p] = Y[p];`
		`}`

`for (p=1; p<=N; p++){`
`  Y[p] = Z[p];`
`  X[p] = Y[p];`
`}`
	Расщепление (fission)
	$s_1 : i = p$
	$s_2 : j = p$
		`for (i=1; i<=N; i++)`
		`  Y[i] = Z[i]; /*s1*/`
		`for (j=1; j<=N; j++)`
		`  X[j] = Y[j]; /*s2*/`

`for (i=1; i<=N; i++) {`
`  Y[i] = Z[i];    /*s1*/`
`  X[i] = Y[i-1]; /*s2*/`
`}`
	Реиндексирование (reindexing)
	$s_1 : p = i$
	$s_2 : p = i - 1$
		`if (N>=1) X[1]=Y[0];`
		`for (p=1; p<=N-1; p++){`
		`  Y[p]=Z[p];`
		`  X[p+1]=Y[p];`
		`}`
		`if (N>=1) Y[N]=Z[N];`

`for (i=1; i<=N; i++)`
`  Y[2*i] = Z[2*i]; /*s1*/`
`for (j=1; j<=2N; j++)`
`  X[j]=Y[j];       /*s2*/`
	Масштабирование (scaling)
	$s_1 : p = 2 * i$
	$(s_2 : p = j)$
		`for (p=1; p<=2*N; p++){`
		`  if (p mod 2 == 0)`
		`    Y[p] = Z[p];`
		`  X[p] = Y[p];`
		`}`

Рис. 11.35. Примитивные аффинные преобразования (часть 1)

Исходный код	Разбиение	Преобразованный код

`for (i=0; i<=N; i++)`
`  Y[N-i] = Z[i]; /*s1*/`
`for (j=0; j<=N; j++)`
`  X[j] = Y[j]; /*s2*/`
s_1
s_2
	Реверс (reversal)
	$s_1 : p = N - i$
	$(s_2 : p = j)$
		`for (p=0; p<=N; p++){`
		`  Y[p] = Z[N-p];`
		`  X[p] = Y[p];`
		`}`
		s_1
		s_2

`for (i=1; i<=N; i++)`
`  for (j=0; j<=M; j++)`
`    Z[i,j] =`
`      Z[i-1,j];`
	Перестановка (permutation)
	$\begin{bmatrix} p \\ q \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} i \\ j \end{bmatrix}$
		`for (p=0; p<=M; p++)`
		`  for (q=1; q<=N; i++)`
		`    Z[q,p] = Z[q-1,p];`

`for (i=1; i<=N+M-1; i++)`
`  for (j=max(1,i+N);`
`      j<=min(i,M); j++)`
`    Z[i,j] =`
`      Z[i-1,j-1];`
	Сдвиг (skewing)
	$\begin{bmatrix} p \\ q \end{bmatrix} = \begin{bmatrix} 1 & -1 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} i \\ j \end{bmatrix} + \begin{bmatrix} 0 \\ 1 \end{bmatrix}$
		`for (p=1; p<=N; p++)`
		`  for (q=1; q<=M; q++)`
		`    Z[p,q-p] =`
		`      Z[p-1,q-p-1];`

Рис. 11.36. Примитивные аффинные преобразования (часть 2)

к соответствующим исходным кодам. (Как слияние может распараллеливать код с синхронизацией, мы рассмотрим в следующем разделе.) В каждом из примеров генерируемый код содержит (внешний) распараллеливаемый цикл, итерации которого могут быть назначены различным процессорам, причем синхронизация при этом не требуется.

Унимодулярные преобразования

Унимодулярное преобразование представлено только унимодулярной матрицей коэффициентов, без константного вектора. *Унимодулярная матрица* — это квадратная матрица, детерминант которой равен ± 1. Важность унимодулярных матриц заключается в том, что они взаимно однозначно отображают n-мерное пространство итераций на другой n-мерный многогранник.

В приведенных примерах иллюстрируется наличие простой геометрической интерпретации распараллеливания. Ребра зависимостей всегда идут от более раннего экземпляра к более позднему. Так, зависимости между различными инструкциями, не вложенными ни в один общий цикл, следуют лексическому порядку; зависимости между инструкциями, вложенными в один и тот же цикл, следуют лексикографическому порядку. Геометрически зависимости двумерной вложенности циклов всегда направлены в диапазоне $[0°, 180°)$, что означает, что угол зависимости должен быть меньше 180°, но не меньше 0°.

Аффинные преобразования изменяют порядок итераций таким образом, что все зависимости являются зависимостями между операциями, находящимися в одной и той же итерации охватывающего цикла. Распараллелить простой исходный текст можно путем изображения зависимостей и геометрического поиска соответствующего преобразования.

11.7.9 Упражнения к разделу 11.7

Упражнение 11.7.1. Дан цикл

```
for (i = 2; i < 100; i++)
    A[i] = A[i-2];
```

а) Чему равно наибольшее количество процессоров, которые могут быть эффективно использованы для выполнения этого цикла?

б) Перепишите код с использованием процессора p в качестве параметра.

в) Запишите ограничения разбиения пространства для данного цикла и найдите одно их решение.

г) Что собой представляет аффинное разбиение данного цикла с наивысшим рангом?

Упражнение 11.7.2. Повторите упражнение 11.7.1 для вложенностей циклов, представленных на рис. 11.37.

```
for (i = 0; i <= 97; i++)
    A[i] = A[i+2];
```

а)

```
for (i = 1; i <= 100; i++)
    for (j = 1; j <= 100; j++)
        for (k = 1; k <= 100; k++) {
            A[i,j,k] = A[i,j,k] + B[i-1,j,k];
            B[i,j,k] = B[i,j,k] + C[i,j-1,k];
            C[i,j,k] = C[i,j,k] + A[i,j,k-1];
        }
```

!б)

```
for (i = 1; i <= 100; i++)
    for (j = 1; j <= 100; j++)
        for (k = 1; k <= 100; k++) {
            A[i,j,k] = A[i,j,k] + B[i-1,j,k];
            B[i,j,k] = B[i,j,k] + A[i,j-1,k];
            C[i,j,k] = C[i,j,k] + A[i,j,k-1] + B[i,j,k];
        }
```

!в)

Рис. 11.37. Исходные тексты к упражнению 11.7.2

Упражнение 11.7.3. Перепишите код

```
for (i = 0; i < 100; i++)
   A[i] = 2*A[i];
for (j = 0; j < 100; j++)
   A[j] = A[j] + 1;
```

так, чтобы он состоял из одного цикла. Перепишите этот код с использованием номера процессора p так, чтобы этот код мог быть разделен между 100 процессорами, причем итерация p выполнялась бы процессором p.

Упражнение 11.7.4. В коде

```
for (i = 1; i < 100; i++)
   for (j = 1; j < 100; j++)
      A[i,j] =
         (A[i-1,j]+A[i+1,j]+A[i,j-1]+A[i,j+1])/4; /* (s) */
```

единственные ограничения заключаются в том, что инструкция s, образующая тело вложенности циклов, должна выполнять итерации $s(i-1,j)$ и $s(i,j-1)$ до итерации $s(i,j)$. Убедитесь в том, что это единственные необходимые ограничения. Затем перепишите код так, чтобы внешний цикл имел индексную переменную p и на p-й итерации внешнего цикла выполнялись все экземпляры $s(i,j)$, такие, что $i+j=p$.

Упражнение 11.7.5. Повторите упражнение 11.7.4, но так, чтобы на p-й итерации внешнего цикла выполнялись те экземпляры s, для которых $i-j=p$.

! **Упражнение 11.7.6.** Объедините циклы

```
for (i = 0; i < 100; i++)
    A[i] = B[i];
for (j = 98; j >= 0; j = j-2)
    B[i] = i;
```

в один цикл с сохранением всех имеющихся зависимостей.

Упражнение 11.7.7. Покажите, что матрица

$$\begin{bmatrix} 2 & 1 \\ 1 & 1 \end{bmatrix}$$

унимодулярна. Опишите преобразование над двумерной вложенностью циклов, выполняемое ею.

Упражнение 11.7.8. Повторите упражнение 11.7.7 для матрицы

$$\begin{bmatrix} 1 & 3 \\ 2 & 5 \end{bmatrix}$$

11.8 Синхронизация между параллельными циклами

Если не разрешать процессорам синхронизироваться, то большинство программ не смогут быть распараллелены. Но, если добавить хотя бы небольшое постоянное количество синхронизирующих операций в программу, степень распараллеливания существенно возрастает. Сначала мы рассмотрим параллельность, которая становится возможной при использовании постоянного количества синхронизаций, а в следующем разделе — общий случай вставки синхронизирующих операций в цикл.

11.8.1 Постоянное количество синхронизаций

Программа, в которой отсутствует распараллеливание, не нуждающееся в синхронизации, может содержать последовательность циклов, некоторые из которых распараллеливаемы, если рассматривать их независимо. Распараллелить такие циклы можно путем введения синхронизирующих барьеров до и после их выполнения (см. пример 11.49).

Пример 11.49. На рис. 11.38 показано программное представление алгоритма интегрирования с чередующимся направлением. В нем нет параллельности, не требующей синхронизации. Зависимости в первой вложенности циклов требуют, чтобы каждый процессор работал со столбцом массива X; во второй вложенности циклов каждый процессор должен работать со строкой массива X. Чтобы обеспечить отсутствие необходимости взаимодействия, весь массив должен обрабатываться одним процессором, так что ни о какой параллельности не может быть и речи. Заметим, однако, что при этом оба цикла по отдельности вполне распараллеливаемы.

```
for (i = 1; i < n; i++)
    for (j = 0; j < n; j++)
        X[i,j] = f(X[i,j] + X[i-1,j]);
for (i = 0; i < n; i++)
    for (j = 1; j < n; j++)
        X[i,j] = g(X[i,j] + X[i,j-1]);
```

Рис. 11.38. Две последовательные вложенности циклов

Один из способов распараллеливания кода состоит в том, что различные процессоры работают с различными столбцами массива в первом цикле, после чего выполняются синхронизация и ожидание завершения работы всеми процессорами, а после этого процессоры приступают к работе со строками массива во втором цикле. Таким путем все вычисления алгоритма могут быть распараллелены при помощи добавления единственной операции синхронизации. Заметим, что в то время как достаточно только одной операции синхронизации, такое распараллеливание требует пересылки почти всех данных матрицы X между процессорами. Снизить степень взаимодействия между процессорами можно путем введения дополнительных операций синхронизации, о чем мы поговорим в разделе 11.9.9. □

Может показаться, что описанный подход применим только к программам, состоящим из последовательностей вложенностей циклов. Однако преобразования кода могут создать дополнительные возможности оптимизации. Можно применить расщепление цикла для разделения циклов в исходном тексте на несколько меньших циклов, которые затем можно по отдельности распараллелить при помощи разделения их барьерами (см. пример 11.50).

Пример 11.50. Рассмотрим цикл

```
for (i=1; i<=n; i++) {
    X[i] = Y[i] + Z[i];    /* (s1) */
    W[A[i]] = X[i];        /* (s2) */
}
```

При отсутствии информации о значениях массива A мы должны предполагать, что доступ в инструкции s_2 может записывать любые элементы массива W. Таким образом, экземпляры s_2 должны выполняться последовательно в том же порядке, в котором они находятся в исходной программе.

Здесь нет параллелизма, не требующего синхронизации, так что алгоритм 11.43 просто назначит все вычисления одному и тому же процессору. Однако параллельно можно выполнять как минимум экземпляры инструкций s_1. Можно распараллелить часть данного кода, обеспечив выполнение разных экземпляров инструкции s_1 разными процессорами, после чего один процессор, скажем, нулевой, выполнит все инструкции s_2 в одном последовательном цикле, как показано в SPMD-коде на рис. 11.39. □

```
X[p] = Y[p] + Z[p]; /* (s1) */
/* Барьер синхронизации */
if (p == 0)
    for (i=1; i<=n; i++)
        W[A[i]] = X[i]; /* (s2) */
```

Рис. 11.39. SPMD-код к примеру 11.50; p — переменная, в которой хранится идентификатор процессора

11.8.2 Графы зависимостей программ

Чтобы обнаружить все возможные при помощи добавления постоянного количества синхронизаций распараллеливания, можно применить "жадное" расщепление исходной программы. Разобьем циклы на максимально возможное количество отдельных циклов, а затем независимо распараллелим каждый из циклов.

Чтобы найти все возможные расщепления цикла, воспользуемся абстракцией *графа зависимостей программы* (program-dependence graph — PDG). Граф зависимостей программы представляет собой граф, узлами которого являются инструкции присваивания в программе, а ребра указывают зависимости данных между инструкциями и их направления. Ребро от инструкции s_1 к инструкции s_2 имеется в том случае, когда имеется зависимость данных между некоторым динамическим экземпляром s_1 и более *поздним* динамическим экземпляром s_2.

Для построения PDG программы мы сначала находим все зависимости данных между каждой парой (не обязательно различных) статических обращений в каждой паре (не обязательно различных) инструкций. Предположим, мы определили, что существует зависимость между обращением $\mathcal{F}_1$ в инструкции s_1 и обращением $\mathcal{F}_2$ в инструкции s_2. Вспомним, что экземпляр инструкции определяется индексным вектором $\mathbf{i} = [i_1, i_2, \ldots, i_m]$, где i_k — индекс цикла k-й вложенности, в котором находится рассматриваемая инструкция.

1. Если существует зависимая пара экземпляров, $\mathbf{i}_1$ инструкции s_1 и $\mathbf{i}_2$ инструкции s_2, и $\mathbf{i}_1$ в исходной программе выполняется до $\mathbf{i}_2$, что записывается как $\mathbf{i}_1 \prec_{s_1 s_2} \mathbf{i}_2$, то в PDG имеется ребро от s_1 к s_2.

2. Аналогично, если существует зависимая пара экземпляров, $\mathbf{i}_1$ инструкции s_1 и $\mathbf{i}_2$ инструкции s_2, и $\mathbf{i}_2 \prec_{s_1 s_2} \mathbf{i}_1$, то в PDG имеется ребро от s_2 к s_1.

Заметим, что возможна ситуация, когда зависимости данных между инструкциями s_1 и s_2 генерируют как ребро от s_1 к s_2, так и ребро от s_2 к s_1.

В частном случае совпадения инструкций s_1 и s_2 $\mathbf{i}_1 \prec_{s_1 s_2} \mathbf{i}_2$ тогда и только тогда, когда $\mathbf{i}_1 \prec \mathbf{i}_2$ ($\mathbf{i}_1$ лексикографически меньше $\mathbf{i}_2$). В общем случае s_1 и s_2 могут быть различными инструкциями, возможно, принадлежащими разным вложенностям циклов.

Пример 11.51. В программе из примера 11.50 между экземплярами инструкций s_1 зависимостей нет. Однако i-й экземпляр инструкции s_2 должен выполняться после i го экземпляра инструкции s_1. Что еще хуже, поскольку ссылка $W[A[i]]$ может приводить к записи любого элемента массива W, i-й экземпляр s_2 зависит от всех предыдущих экземпляров s_2, т.е. инструкция s_2 зависит сама от себя. PDG для программы из примера 11.50 показан на рис. 11.40. Обратите внимание на наличие в графе единственного цикла, содержащего только узел s_2. □

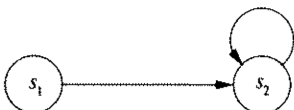


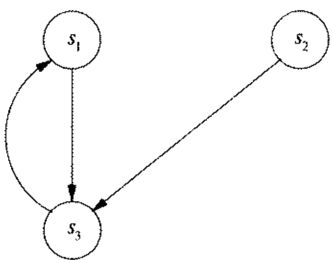
Рис. 11.40. Граф зависимостей программы для кода из примера 11.50

Граф зависимостей программы облегчает определение того, можно ли разделить инструкции в цикле. Инструкции, соединенные в цикл в PDG, разделены быть не могут. Если $s_1 \to s_2$ — зависимость между двумя инструкциями в цикле, то некоторый экземпляр s_1 должен быть выполнен до некоторого экземпляра s_2, и наоборот. Заметим, что такая взаимозависимость наблюдается, только если s_1

и s_2 находятся в одном общем цикле. Из-за этой взаимозависимости мы не можем выполнить все экземпляры одной инструкции до другой, а значит, и выполнить расщепление. С другой стороны, если зависимость $s_1 \rightarrow s_2$ — однонаправленная, то можно разделить цикл и сначала выполнить все инструкции s_1, а затем все инструкции s_2.

```
for (i = 0; i < n; i++) {
    Z[i] = Z[i] / W[i];              /* (s1) */
    for (j = i; j < n; j++) {
        X[i,j] = Y[i,j]*Y[i,j];      /* (s2) */
        Z[j] = Z[j] + X[i,j];        /* (s3) */
    }
}
```

а) Программа



б) Ее граф зависимостей

Рис. 11.41. Программа и граф зависимостей к примеру 11.52

Пример 11.52. На рис. 11.41, *б* показан граф зависимостей программы для кода, представленного на рис. 11.41, *а*. Инструкции s_1 и s_3 принадлежат циклу в графе, а значит, не могут быть помещены в разные циклы. Однако мы можем отделить инструкцию s_2 и выполнить все ее экземпляры до остальных вычислений, как показано на рис. 11.42. Первый цикл распараллеливаем, второй — нет. Можно распараллелить первый цикл, поместив барьеры до и после параллельных вычислений. □

11.8.3 Иерархическое время

Вычислить отношение $\prec_{s_1 s_2}$ в общем виде очень сложно, но имеется семейство программ, к которым применимы описываемые в данном разделе оптимизации и для которых существует простой способ вычисления зависимостей. Предположим, что программа блочно структурирована, состоит из циклов и простых

```
for (i = 0; i < n; i++)
    for (j = i; j < n; j++)
        X[i,j] = Y[i,j]*Y[i,j];    /* (s2) */
for (i = 0; i < n; i++) {
    Z[i] = Z[i] / W[i];            /* (s1) */
    for (j = i; j < n; j++)
        Z[j] = Z[j] + X[i,j];      /* (s3) */
}
```

Рис. 11.42. Группирование сильно связанных компонентов вложенности циклов

арифметических операций и не содержит иных управляющих конструкций. Инструкция программы представляет собой либо инструкцию присваивания, либо последовательность инструкций, либо цикл, телом которого является инструкция. Таким образом, управляющие структуры образуют иерархию. На вершине этой иерархии находится узел, представляющий инструкцию всей программы в целом. Инструкции присваивания являются листьями. Если инструкция представляет собой последовательность, то ее дочерними узлами являются инструкции последовательности, располагающиеся слева направо в соответствии с их лексикографическим порядком. Если инструкция является циклом, то ее дочерними узлами являются компоненты тела цикла, обычно представляющие собой последовательность из одной или нескольких инструкций.

```
s0;
L1:  for (i = 0; ...) {
         s1;
      L2:  for (j = 0; ...) {
               s2;
               s3;
           }
      L3:  for (k = 0; ... )
               s4;
      s5;
     }
```

Рис. 11.43. Иерархически структурированная программа

Пример 11.53. Иерархическая структура программы, которая представлена на рис. 11.43, показана на рис. 11.44. Иерархическая природа последовательности выполнения видна из рис. 11.45. Единственный экземпляр s_0 предшествует всем другим операциям, будучи первой выполняемой инструкцией. Затем мы выполня-

ем все команды из первой итерации внешнего цикла перед командами из второй итерации и т.д. Для всех динамических экземпляров, для которых индекс цикла i имеет значение 0, инструкции s_1, L_2, L_3 и s_5 выполняются в лексическом порядке. Можно повторить те же рассуждения для генерации остальной части порядка выполнения. □

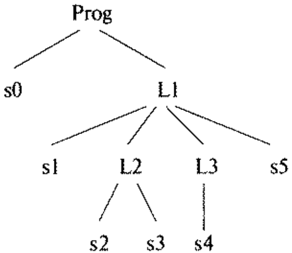


Рис. 11.44. Иерархическая структура программы из примера 11.53

1:	s_0				
2:	L_1	$i = 0$	s_1		
3:			L_2	$j = 0$	s_2
4:					s_3
5:				$j = 1$	s_2
6:					s_3
7:				...	
8:			L_3	$k = 0$	s_4
9:				$k = 1$	s_4
10:				...	
11:			s_5		
12:		$i = 1$	s_1		
13:			...		

Рис. 11.45. Порядок выполнения программы из примера 11.53

Можно разрешить упорядочение двух экземпляров из двух разных инструкций иерархическим методом. Если имеются общие циклы, в которых находятся эти инструкции, то надо сравнить значения индексов этих общих циклов, начиная с внешнего цикла. Как только между индексными значениями будет найдено несоответствие, их разность определит порядок инструкций. Переходить к сравнению

значений индексов внутренних циклов следует только тогда, когда значения индексов внешних циклов совпадают. Этот процесс аналогичен сравнению времени, выраженному в часах, минутах и секундах. Сначала мы сравниваем часы; если эти значения одинаковы, мы переходим к сравнению минут и т.д. Если индексные значения одинаковы для всех общих циклов, то разрешение порядка выполняется на основе относительного лексического расположения инструкций. Поэтому рассмотренный здесь порядок выполнения простых программ с вложенными циклами часто называют иерархическим временем (hierarchical time).

Пусть s_1 — инструкция, вложенная в цикл на глубине d_1, а s_2 — в цикл на глубине d_2, количество общих (внешних) циклов — d; понятно, что $d \leqslant d_1$ и $d \leqslant d_2$. Предположим, что экземпляр s_1 — $\mathbf{i} = [i_1, i_2, \ldots, i_{d_1}]$, а экземпляр s_2 — $\mathbf{j} = [j_1, j_2, \ldots, j_{d_2}]$.

$\mathbf{i} \prec_{s_1 s_2} \mathbf{j}$ тогда и только тогда, когда либо

1. $[i_1, i_2, \ldots, i_d] \prec [j_1, j_2, \ldots, j_d]$, либо

2. $[i_1, i_2, \ldots, i_d] = [j_1, j_2, \ldots, j_d]$ и s_1 лексически появляется перед s_2.

Предикат $[i_1, i_2, \ldots, i_d] \prec [j_1, j_2, \ldots, j_d]$ можно записать как дизъюнкцию линейных неравенств:

$$(i_1 < j_1) \vee (i_1 = j_1 \wedge i_2 < j_2) \vee \ldots \vee (i_1 = j_1 \wedge \ldots \wedge i_{d-1} = j_{d-1} \wedge i_d < j_d)$$

Ребро PDG от s_1 к s_2 существует в том случае, когда условие зависимости данных и одно из выражений дизъюнкции могут быть истинны одновременно. Таким образом, для решения вопроса о существовании одного ребра нам может потребоваться решить d или $d + 1$ целочисленных линейных программ, в зависимости от того, находится ли инструкция s_1 лексически до s_2.

11.8.4 Алгоритм распараллеливания

Теперь мы представим простой алгоритм, который сначала разделяет вычисления на максимально возможное количество циклов, а затем независимо распараллеливает их.

Алгоритм 11.54. Максимизация степени параллельности с использованием $O(1)$ синхронизаций

Вход: программа с обращениями к массивам.

Выход: SPMD-код с константным количеством барьеров синхронизации.

Метод: выполняем следующие действия.

1. Строим граф зависимостей программы и разбиваем инструкции на сильно связанные компоненты. Вспомним, что сильно связанным компонентом называется максимальный подграф исходного графа, в котором из каждого узла подграфа можно достичь любого другого его узла (см. раздел 10.5.8).

2. Преобразуем код так, чтобы он выполнял сильно связанные компоненты в топологическом порядке, при необходимости применяя расщепление.

3. Применяем алгоритм 11.43 к каждому сильно связанному компоненту для поиска распараллеливаний, не требующих синхронизации. Барьеры синхронизации вставляются до и после каждого распараллеливаемого сильно связанного компонента. □

Алгоритм 11.54 находит все степени параллелизма при использовании $O(1)$ синхронизаций, но при этом он имеет ряд слабых мест. Во-первых, он может вносить излишние синхронизации. Очевидно, что если мы применим этот алгоритм к программе, которая может быть распараллелена без синхронизации, то алгоритм будет распараллеливать каждую инструкцию независимо, вводя при этом барьеры синхронизации между параллельными циклами, выполняющими каждую инструкцию. Во-вторых, при ограниченном количестве синхронизаций схема распараллеливания при каждой синхронизации может перемещать между процессорами большое количество данных. В некоторых случаях стоимость взаимодействий процессоров делает параллелизм слишком дорогим, так что такую программу лучше выполнять последовательно, на одном процессоре. В следующем разделе мы рассмотрим вопросы повышения локальности и снижения количества взаимодействий процессоров.

11.8.5 Упражнения к разделу 11.8

Упражнение 11.8.1. Примените алгоритм 11.54 к коду на рис. 11.46.

```
for (i=0; i<100; i++)
    A[i] = A[i] + X[i]; /* (s1) */
for (i=0; i<100; i++)
    for (j=0; j<100; j++)
        B[i,j] = Y[i,j] + A[i] + A[j]; /* (s2) */
```

Рис. 11.46. Код к упражнению 11.8.1

Упражнение 11.8.2. Примените алгоритм 11.54 к коду на рис. 11.47.

Упражнение 11.8.3. Примените алгоритм 11.54 к коду на рис. 11.48.

11.9 Конвейеризация

При конвейеризации задача разбивается на ряд стадий, выполняемых на разных процессорах. Например, задача, вычисляемая с использованием цикла из n

```
for (i=0; i<100; i++)
    A[i] = A[i] + X[i]; /* (s1) */
for (i=0; i<100; i++) {
    B[i] = B[i] + A[i]; /* (s2) */
    for (j=0; j<100; j++)
        C[j] = Y[j] + B[j]; /* (s3) */
}
```

Рис. 11.47. Код к упражнению 11.8.2

```
for (i=0; i<100; i++)
    A[i] = A[i] + X[i]; /* (s1) */
for (i=0; i<100; i++) {
    for (j=0; j<100; j++)
        B[j] = A[i] + Y[j]; /* (s2) */
    C[i] = B[i] + Z[i]; /* (s3) */
    for (j=0; j<100; j++)
        D[i,j] = A[i] + B[j]; /* (s4) */
}
```

Рис. 11.48. Код к упражнению 11.8.3

итераций, может быть структурирована, как конвейер из n стадий. Каждая стадия назначается своему процессору; когда один процессор завершает выполнение своей стадии, результаты его работы передаются другому процессору в качестве входных данных.

Далее мы рассмотрим концепцию конвейеризации более детально. Затем в разделе 11.9.2 для иллюстрации условий применимости конвейеризации будет приведен численный алгоритм реального времени, известный как последовательная сверхрелаксация. Формально ограничения, которые необходимо разрешить, будут определены в разделе 11.9.6, а описание алгоритма для их разрешения — в разделе 11.9.7. Программы, имеющие множественные независимые решения ограничений временны́х разбиений, известны как имеющие внешние *полностью переставляемые циклы* (fully permutable loops); такие циклы, как будет показано в разделе 11.9.8, легко конвейеризуются.

11.9.1 Что такое конвейеризация

Наши первоначальные попытки распараллеливания циклов разбивали итерации вложенности циклов таким образом, чтобы две итерации, использующие одни и те же данные, назначались одному и тому же процессору. Конвейеризация позволяет процессорам использовать одни и те же данные, но в общем случае делает

это только "локально", с передачей данных от одного процессора другому, соседнему с ним в пространстве процессоров. Вот простой пример.

Пример 11.55. Рассмотрим цикл

```
for (i = 1; i <= m; i++)
    for (j = 1; j <= n; j++)
        X[i] = X[i] + Y[i,j];
```

Этот код суммирует i-ю строку Y и добавляет ее к i-му элементу X. Внутренний цикл, соответствующий суммированию, должен выполняться последовательно из-за зависимости данных[7]. Однако разные задачи суммирования независимы. Этот код можно распараллелить, заставляя каждый процессор выполнять отдельное суммирование. Процессор i обращается к строке i массива Y и обновляет i-й элемент массива X.

В качестве альтернативы можно структурировать процессоры для выполнения конвейерного суммирования и получить параллелизм за счет перекрытия суммирований, как показано на рис. 11.49. Говоря более точно, каждая итерация внутреннего цикла может рассматриваться как стадия конвейера: стадия j берет элемент X, сгенерированный предыдущей стадией, добавляет его к элементу Y и передает результат следующей стадии. Заметим, что в этом случае каждый процессор обращается к одному столбцу, а не к строке, массива Y. Если Y хранится постолбцово, то при такой организации вычислений повышается локальность.

Время	Процессоры		
	1	2	3
1	$X[1] += Y[1,1]$		
2	$X[2] += Y[2,1]$	$X[1] += Y[1,2]$	
3	$X[3] += Y[3,1]$	$X[2] += Y[2,2]$	$X[1] += Y[1,3]$
4	$X[4] += Y[4,1]$	$X[3] += Y[3,2]$	$X[2] += Y[2,3]$
5		$X[4] += Y[4,2]$	$X[3] += Y[3,3]$
6			$X[4] += Y[4,3]$

Рис. 11.49. Конвейерное выполнение примера 11.55 при $m = 4$ и $n = 3$

Можно инициировать новую задачу, как только первый процессор выполнит первую стадию предыдущей. Изначально конвейер пуст, и только первый процессор выполняет первую стадию. После ее завершения результат передается второму процессору, в то время как первый процессор начинает вторую задачу,

[7]Вспомните, что в свое время мы договорились не использовать коммутативность и ассоциативность сложения.

и т.д. Таким образом конвейер постепенно заполняется, пока все процессоры не будут заняты. Когда первый процессор завершает последнюю задачу, конвейер начинает опустошаться; при этом все большее количество процессоров освобождается — до тех пор, пока последний процессор не выполнит свою последнюю задачу. В устойчивом состоянии конвейером из n процессоров могут параллельно выполняться n задач. □

Интересно сравнить конвейеризацию с простым параллелизмом, когда разные процессоры выполняют разные задачи.

- Конвейеризация применима только ко вложенностям с глубиной не менее 2. Каждую итерацию внешнего цикла можно рассматривать как задачу, а итерации во внутреннем цикле — как стадии этой задачи.

- Задачи, выполняемые конвейерно, могут зависеть одна от другой. Информация, относящаяся к одной и той же стадии каждой задачи, хранится одним и тем же процессором; таким образом, результаты, сгенерированные i-й стадией задачи, могут использоваться i-й стадией следующей задачи без затрат на сообщение между процессорами. Аналогично каждый элемент входных данных, используемый одной стадией различных задач, как было проиллюстрировано примером 11.55, хранится только одним процессором.

- Если задачи независимы, то простое распараллеливание дает более высокую степень использования процессора, поскольку процессоры могут выполнять всю свою работу без затрат на накладные расходы по заполнению и освобождению конвейера. Однако, как показано в примере 11.55, шаблон обращения к данным в конвейерной схеме отличается от такового в случае простого распараллеливания. Конвейеризация может оказаться предпочтительнее, если она снижает объем межпроцессорного взаимодействия.

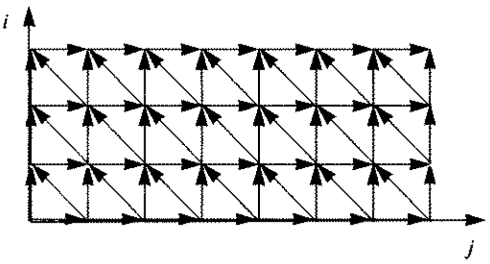
11.9.2 Последовательная сверхрелаксация: пример

Последовательная сверхрелаксация (successive over-relaxation — SOR) представляет собой способ ускорения сходимости методов релаксации для решения систем линейных уравнений. На рис. 11.50, *а* показан относительно простой пример, иллюстрирующий соответствующий шаблон обращения к данным. Здесь новое значение элемента массива зависит от значений его соседей. Такая операция выполняется до тех пор, пока не будет достигнуто выполнение некоторого критерия сходимости.

На рис. 11.50, *б* показаны ключевые зависимости данных. Здесь не приведены зависимости, которые могут быть получены из уже имеющихся на рисунке. Например, итерация $[i, j]$ зависит от итераций $[i, j-1]$, $[i, j-2]$ и т.д. Из приведенных зависимостей очевидно, что параллелизм, не требующий синхронизации,

```
for (i = 0; i <= m; i++)
    for (j = 0; j <= n; j++)
        X[j+1] = 1/3 * (X[j] + X[j+1] + X[j+2]);
```

а) Исходный текст



б) Зависимости данных в приведенном коде

Рис. 11.50. Пример последовательной сверхрелаксации

в данном случае не имеет места. Поскольку наидлиннейшая цепочка зависимостей состоит из $O(m+n)$ ребер, при добавлении синхронизации мы получим одну степень параллелизма и сможем выполнять $O(mn)$ операций за $O(m+n)$ единиц времени.

В частности, заметим, что итерации, лежащие вдоль 150-градусных диагоналей[8] на рис. 11.50, *б*, не связаны никакими зависимостями. Они зависят только от итераций, лежащих на диагоналях, находящихся ближе к началу координат. Следовательно, можно распараллелить приведенный код путем выполнения итераций поочередно на каждой из диагоналей, начиная с начала координат и двигаясь от него в сторону увеличения значений индексных переменных. Будем говорить об итерациях вдоль каждой диагонали как о *волновом фронте* (wavefront), а саму схему распараллеливания именовать *волновой* (wavefronting).

11.9.3 Полностью переставляемые циклы

Сначала рассмотрим понятие *полной переставляемости* (full permutability), концепции, которая используется в конвейеризации и других оптимизациях. Циклы *полностью переставляемы* (fully permutable), если они могут быть произвольным образом переставлены без изменения семантики исходной программы. Если циклы приведены к полностью переставляемому виду, код можно легко конвейеризовать и применить к нему преобразования, такие как блокирование, для повышения локальности данных.

[8]Последовательностей точек, образованных многократными перемещениями на 1 вниз и на 2 вправо.

SOR-код, такой как на рис. 11.50, *а*, полностью переставляемым не является. Как было показано в разделе 11.7.8, перестановка двух циклов означает, что итерации в исходном пространстве итераций выполняются не построчно, а постолбцово. Например, исходное вычисление в итерации $[2, 3]$ будет выполнено до итерации $[1, 4]$, что нарушает показанные на рис. 11.50, *б* зависимости.

Мы можем, однако, преобразовать код так, чтобы сделать его полностью переставляемым. Применение аффинного преобразования

$$\begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix}$$

к исходному коду дает код, показанный на рис. 11.51, *а*. Такой преобразованный код полностью переставляем, и его переставленная версия показана на рис. 11.51, *в*. Пространства итераций и зависимости данных этих двух программ приведены на рис. 11.51, *б* и *д* соответственно. Из показанных схем легко увидеть, что такое упорядочение сохраняет относительный порядок между всеми зависимыми парами обращений.

При перестановке циклов мы радикально изменяем множество операций, выполняемых в каждой итерации внешнего цикла. То, что у нас имеется эта степень свободы при планировании, означает, что имеется определенная нежесткость в упорядочении операций в программе. Нежесткость в планировании означает наличие возможностей для распараллеливания. Ниже в этом разделе будет показано, что если вложенность циклов содержит k внешних полностью переставляемых циклов, то добавление $O(n)$ синхронизаций позволяет получить $O(k-1)$ степеней параллелизма (здесь n — количество итераций в цикле).

11.9.4 Конвейеризация полностью переставляемых циклов

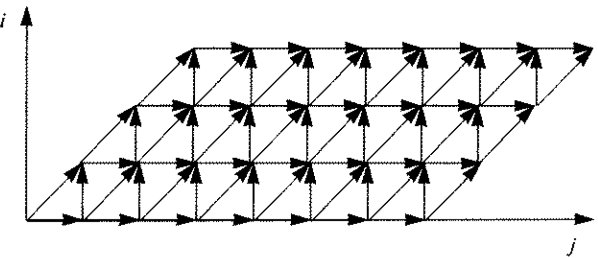
Вложенность циклов с k внешними полностью переставляемыми циклами может быть структурирована как конвейер с $O(k-1)$ размерностями. В SOR-примере $k = 2$, так что процессоры можно структурировать как линейный конвейер.

Конвейеризовать SOR-код можно двумя разными способами, показанными на рис. 11.52, *а* и *б*, соответствующими двум перестановкам на рис. 11.51, *а* и *в* соответственно. В каждом случае каждый столбец пространства итераций составляет задачу, а каждая строка — стадию. Мы назначаем стадию i процессору i; таким образом, каждый процессор выполняет внутренний цикл кода. Игнорируя граничные условия, процессор может выполнить итерацию i только после того, как процессор $p-1$ выполнит итерацию $i-1$.

Предположим, что каждый процессор требует в точности одного и того же количества времени для выполнения итераций, а синхронизация выполняется мгно-

```
for (i = 0; i <= m; i++)
    for (j = i; j <= i+n; j++)
        X[j-i+1] = 1/3 * (X[j-i] + X[j-i+1] + X[j-i+2]);
```

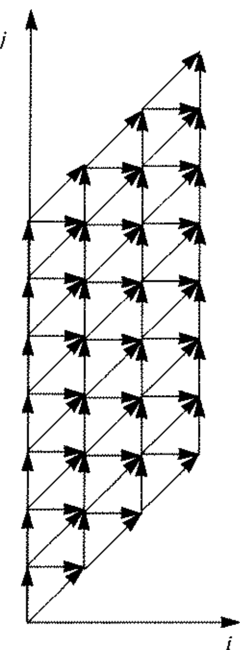
а) Код на рис. 11.50 после преобразования $\begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix}$



б) Зависимости данных кода из части а

```
for (j = 0; j <= m+n; j++)
    for (i = max(0,j); i <= min(m,j), i++)
        X[j-i+1] = 1/3 * (X[j-i] + X[j-i+1] + X[j-i+2]);
```

в) Перестановка циклов в коде из части а



г) Зависимости данных кода из части в

Рис. 11.51. Полностью переставляемая версия кода, приведенного на рис. 11.50

```
/* 0 <= p <= m */
for (j = p; j <= p+n; j++) {
    if (p > 0) wait (p-1);
    X[j-p+1] = 1/3 * (X[j-p] + X[j-p+1] + X[j-p+2]);
    if (p < min (m,j)) signal (p+1);
}
```

а) Процессоры назначены строкам

```
/* 0 <= p <= m+n */
for (i = max(0,p); i <= min(m,p); i++) {
    if (p > max(0,i)) wait (p-1);
    X[p-i+1] = 1/3 * (X[p-i] + X[p-i+1] + X[p-i+2]);
    if (p < m+n) & (p > i) signal (p+1);
}
```

б) Процессоры назначены столбцам

Рис. 11.52. Две реализации конвейеризации кода, представленного на рис. 11.51

венно. Обе схемы параллельно выполняют одни и те же итерации; единственное отличие заключается в том, что при этом используются другие назначения процессорам. Все выполняемые параллельно итерации лежат на 135-градусных диагоналях в пространстве итераций на рис. 11.51, *б*, которые соответствуют 150-градусным диагоналям в пространстве итераций исходного кода (см. рис. 11.50, *б*).

Однако на практике процессоры с кэшированием не всегда выполняют один и тот же код за одно и то же время; варьируется и время синхронизации. В отличие от применения барьеров синхронизации, когда все процессоры работают по жесткой схеме, конвейеризация требует от процессора синхронизации и обмена информацией не более чем с двумя другими процессорами. Таким образом, конвейеризация характеризуется ослабленным волновым фронтом, позволяя одним процессорам ненадолго вырываться вперед, а другим отставать. Такая гибкость сокращает время, которое процессоры затрачивают на ожидание других процессоров, и повышает производительность параллельной работы.

Приведенные выше схемы конвейеризации — всего лишь два из множества возможных способов конвейеризации вычислений. Как уже говорилось, если цикл полностью переставляем, имеется достаточная свобода в выборе способа распараллеливания кода. Первая схема отображает итерацию $[i, j]$ на процессор i; вторая отображает итерацию $[i, j]$ на процессор j. Можно создать альтернативные способы конвейеризации, отображая итерацию $[i, j]$ на процессор $c_0 i + c_1 j$, где c_0 и c_1 — положительные константы. Такая схема создает конвейеры с ослабленными волновыми фронтами в диапазоне от 90° до 180° (исключая граничные значения).

11.9.5 Общая теория

Рассмотренный выше пример иллюстрирует общую теорию, лежащую в основе конвейеризации: если можно получить как минимум два разных внешних цикла вложенности при удовлетворении всех зависимостей, то такие вычисления могут быть конвейеризованы. Вложенность с k внешними полностью переставляемыми циклами имеет степень конвейерного параллелизма, равную $k - 1$.

Вложенности, которые не могут быть конвейеризованы, не имеют возможности изменять внешние циклы. В примере 11.56 показан один из таких случаев. Чтобы удовлетворить все зависимости, каждая итерация внешнего цикла должна выполнять в точности те вычисления, которые содержатся в исходном коде. Однако такой код может все еще быть распараллелен на уровне внутренних циклов, для чего может потребоваться добавление как минимум n синхронизаций, где n — количество итераций во внешнем цикле.

```
for (i = 0; i < 100; i++) {
    for (j = 0; j < 100; j++)
        X[j] = X[j] + Y[i,j]; /* (s1) */
    Z[i] = X[A[i]]; /* (s2) */
}
```

а)

s_1 s_2

б)

Рис. 11.53. Последовательный внешний цикл (а) и его граф зависимостей программы (б)

Пример 11.56. На рис. 11.53 показана более сложная версия проблемы, с которой мы сталкивались в примере 11.50. Как видно из графа зависимостей программы на рис. 11.53, *б*, инструкции s_1 и s_2 принадлежат одному и тому же сильно связанному компоненту. Поскольку мы не знаем содержимое матрицы A, мы должны считать, что обращение в инструкции s_2 может считывать любой из элементов X. В приведенном коде имеется истинная зависимость инструкции s_2 от s_1 и антизависимость s_1 от s_2. Для конвейеризации нет никакой возможности, поскольку все операции, принадлежащие итерации i во внешнем цикле, должны предшествовать операциям в итерации $i + 1$. Продолжим поиск возможностей распараллеливания во внутреннем цикле. При этом мы обнаруживаем, что итерации во втором цикле могут быть распараллелены без применения синхронизации. Таким образом, все-

го нам потребуется 200 барьеров синхронизации — по одному до и после каждого выполнения внутреннего цикла. □

11.9.6 Ограничения временного разбиения

Перейдем теперь к задаче поиска конвейерного распараллеливания. Наша цель состоит в превращении вычисления в набор конвейеризуемых задач. При поиске конвейерного параллелизма мы не решаем, как при распараллеливании циклов, что и каким процессором будет выполняться. Вместо этого мы задаем следующий фундаментальный вопрос: каковы все возможные последовательности выполнения, которые соблюдают исходные зависимости данных в цикле? Очевидно, что всем зависимостям данных удовлетворяет исходная последовательность выполнения. Вопрос в том, существуют ли аффинные преобразования, позволяющие создать альтернативный план выполнения, при котором итерации внешнего цикла выполняют множество операций, отличающееся от исходного, но при этом удовлетворяют всем зависимостям данных. Если мы можем найти такие преобразования, то можем конвейеризовать цикл. Ключевой момент заключается в том, что если имеется свобода в планировании операций, то имеется и параллелизм; детально вопрос о том, как получить конвейерный параллелизм из таких преобразований, будет рассмотрен ниже.

Для того чтобы найти приемлемое переупорядочение внешнего цикла, нам надо найти одномерные аффинные преобразования, по одному для каждой инструкции, которые отображают исходные значения индекса цикла на номер итерации во внешнем цикле. Преобразования допустимы, если присваивания удовлетворяют всем зависимостям данных в программе. "Ограничения временно́го разбиения" (time-partition constraints), показанные ниже, просто гласят, что если одна операция зависит от другой, то первая не должна быть назначена итерации внешнего цикла более ранней, чем вторая. Если они назначены одной и той же итерации, то понятно, что первая операция должна быть выполнена в пределах этой итерации только после второй.

Отображение аффинного разбиения программы является корректным временны́м разбиением (legal-time partition) тогда и только тогда, когда для любых двух (не обязательно различных) зависимых обращений, скажем, $\mathcal{F}_1 = \langle \mathbf{F}_1, \mathbf{f}_1, \mathbf{B}_1, \mathbf{b}_1 \rangle$ в инструкции s_1, вложенной в d_1 циклов, и $\mathcal{F}_2 = \langle \mathbf{F}_2, \mathbf{f}_2, \mathbf{B}_2, \mathbf{b}_2 \rangle$ в инструкции s_2, вложенной в d_2 циклов, отображения одномерных разбиений $\langle \mathbf{C}_1, \mathbf{c}_1 \rangle$ и $\langle \mathbf{C}_2, \mathbf{c}_2 \rangle$ для инструкций s_1 и s_2 соответственно удовлетворяют ограничениям временно́го разбиения:

- для всех $\mathbf{i}_1$ из Z^{d_1} и $\mathbf{i}_2$ из Z^{d_2}, таких, что

  а) $\mathbf{i}_1 \prec_{s_1 s_2} \mathbf{i}_2$,

  б) $\mathbf{B}_1 \mathbf{i}_1 + \mathbf{b}_1 \geqslant \mathbf{0}$,

в) $\mathbf{B}_2\mathbf{i}_2 + \mathbf{b}_2 \geqslant \mathbf{0}$ и

г) $\mathbf{F}_1\mathbf{i}_1 + \mathbf{f}_1 = \mathbf{F}_2\mathbf{i}_2 + \mathbf{f}_2$,

выполняется $\mathbf{C}_1\mathbf{i}_1 + \mathbf{c}_1 \leqslant \mathbf{C}_2\mathbf{i}_2 + \mathbf{c}_2$.

Это ограничение, проиллюстрированное на рис. 11.54, выглядит удивительно похожим на ограничения разбиения пространства. Это ослабление ограничений разбиения пространства состоит в том, что если две итерации обращаются к одной и той же ячейке памяти, то они не обязательно должны быть отображены в один и тот же раздел; мы требуем только, чтобы сохранялся относительный порядок их выполнения, т.е. в этих ограничениях вместо = используется $\leqslant$.

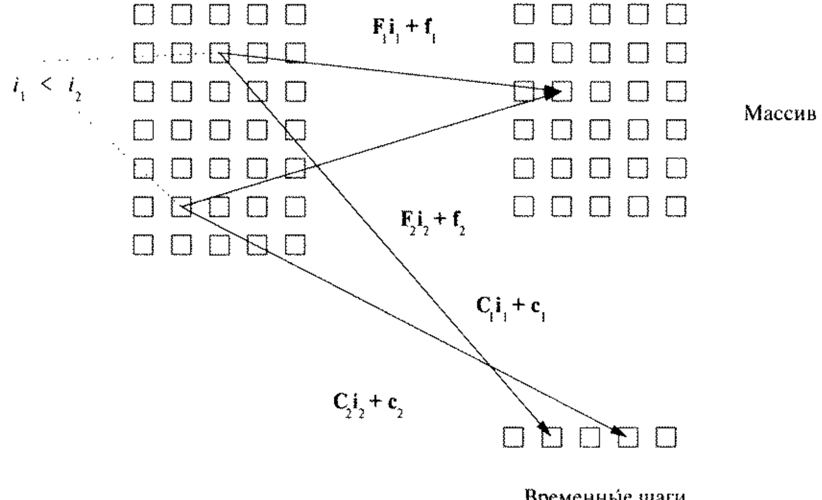


Рис. 11.54. Ограничения временных разбиений

Мы знаем, что существует как минимум одно решение ограничений временны́х разбиений. Можно отобразить операции в каждой итерации внешнего цикла на ту же самую итерацию, и при этом все зависимости данных будут удовлетворены. Это — решение ограничений временны́х разбиений для программы, которая не может быть конвейеризована. С другой стороны, если можно найти несколько независимых решений ограничений временны́х разбиений, то программа может быть конвейеризована. Каждое независимое решение соответствует циклу во внешней полностью переставляемой вложенности. Как и следовало ожидать, у программы из примера 11.56, у которой отсутствует конвейеризуемый параллелизм, имеется только одно независимое решение ограничений временны́х разбиений, а у примера SOR-кода имеется два независимых решения.

Пример 11.57. Рассмотрим пример 11.56, в частности зависимости данных между обращениями к массиву X в инструкциях s_1 и s_2. Поскольку в s_2 обращение

не является аффинным, аппроксимируем это обращение путем моделирования матрицы X в анализе зависимостей, включающих s_2, просто как скалярной переменной. Пусть (i, j) — индексное значение динамического экземпляра s_1, а i' — индексное значение динамического экземпляра s_2. Пусть отображениями s_1 и s_2 являются соответственно $\langle [C_{11}, C_{12}], c_1 \rangle$ и $\langle [C_{21}], c_2 \rangle$.

Рассмотрим сначала временны́е ограничения, накладываемые зависимостями s_2 от s_1. Мы получаем $i \leqslant i'$, т.е. преобразованная (i, j)-я итерация s_1 должна выполняться не позже преобразованной i'-й итерации s_2:

$$\begin{bmatrix} C_{11} & C_{12} \end{bmatrix} \begin{bmatrix} i \\ j \end{bmatrix} + c_1 \leqslant C_{21} i' + c_2$$

Раскрывая это соотношение, получаем

$$C_{11} i + C_{21} j + c_1 \leqslant C_{21} i' + c_2$$

Поскольку j может иметь произвольно большое значение, не зависящее от i и i', C_{12} должно быть равно 0. Таким образом, одно из возможных решений ограничений следующее:

$$C_{11} = C_{21} = 1 \text{ и } C_{12} = c_1 = c_2 = 0$$

Такие же рассуждения о зависимости s_1 от s_2 и s_2 от самой себя приводят к аналогичному ответу. В этом частном решении i-я итерация внешнего цикла, состоящая из экземпляра i инструкции s_2 и всех экземпляров (i, j) инструкции s_1, получает временну́ю отметку i. Прочие допустимые варианты выбора C_{11}, C_{21}, c_1 и c_2 дают аналогичные назначения, хотя имеются временны́е метки, когда ничего не происходит. Иначе говоря, все способы планирования внешнего цикла требуют, чтобы итерации выполнялись в том же порядке, в котором они выполняются в исходном коде. Это утверждение справедливо и когда все 100 итераций выполняются на одном и том же процессоре, и когда они выполняются на 100 различных процессорах, и в любом промежуточном между этими двумя крайностями случае. □

Пример 11.58. В SOR-коде, показанном на рис. 11.50, *а*, обращение для записи $X[j+1]$ зависит от самого себя и от трех обращений для чтения. Мы ищем отображение $\langle [C_1, C_2], c \rangle$ для инструкции присваивания, такое, что при наличии зависимости (i', j') от (i, j) выполняется соотношение

$$\begin{bmatrix} C_1 & C_2 \end{bmatrix} \begin{bmatrix} i \\ j \end{bmatrix} + \begin{bmatrix} c \end{bmatrix} \leqslant \begin{bmatrix} C_1 & C_2 \end{bmatrix} \begin{bmatrix} i' \\ j' \end{bmatrix} + \begin{bmatrix} c \end{bmatrix}$$

По определению $(i, j) \prec (i', j')$; т.е. либо $i < i'$, либо $(i = i' \wedge j < j')$.

Рассмотрим три пары зависимостей данных.

1. Истинная зависимость обращения для чтения $X[j+2]$ от обращения для записи $X[j+1]$. Поскольку эти экземпляры должны обращаться к одному и тому же месту в памяти, $j+1 = j'+2$, или $j = j'+1$. Подставляя $j = j'+1$ во временны́е ограничения, мы получаем

   $$C_1\left(i'-i\right) - C_2 \geqslant 0$$

   Поскольку $j = j'+1$, $j > j'$, и предыдущие ограничения сводятся к $i < i'$. Следовательно,

   $$C_1 - C_2 \geqslant 0$$

2. Антизависимость обращения для записи $X[j+1]$ от обращения для чтения $X[j+2]$. Здесь $j+2 = j'+1$, или $j = j'-1$. Подставляя $j = j'-1$ во временны́е ограничения, получаем

   $$C_1\left(i'-i\right) + C_2 \geqslant 0$$

   При $i = i'$ получаем

   $$C_2 \geqslant 0$$

   При $i < i'$, поскольку $C_2 \geqslant 0$, получаем

   $$C_1 \geqslant 0$$

3. Зависимость по выходу обращения для записи $X[j+1]$ от себя самого. Здесь $j = j'$. Временны́е ограничения сводятся к

   $$C_1\left(i'-i\right) \geqslant 0$$

   Поскольку существенны только значения $i < i'$, мы вновь получаем

   $$C_1 \geqslant 0$$

Остальные зависимости не приводят к новым ограничениям. Всего существует три ограничения:

$$\begin{aligned} C_1 &\geqslant 0 \\ C_2 &\geqslant 0 \\ C_1 - C_2 &\geqslant 0 \end{aligned}$$

Двумя независимыми решениями этих ограничений являются

$$\begin{bmatrix} 1 \\ 0 \end{bmatrix} \text{ и } \begin{bmatrix} 1 \\ 1 \end{bmatrix}$$

Первое решение сохраняет порядок выполнения итераций во внешнем цикле. Как исходный SOR-код на рис. 11.50, *а*, так и преобразованный код на рис. 11.51, *а* являются примерами таких схем. Второе решение размещает итерации вдоль 135-градусных диагоналей в том же внешнем цикле. В качестве соответствующего примера можно привести код, показанный на рис. 11.51, *б*.

Обратите внимание на существование многих других возможных пар независимых решений. Например,

$$\begin{bmatrix} 1 \\ 1 \end{bmatrix} \text{ и } \begin{bmatrix} 2 \\ 1 \end{bmatrix}$$

также представляют собой независимые решения тех же ограничений. Для упрощения преобразования кода мы выбираем среди решений наипростейшие векторы. □

11.9.7 Решение временны́х ограничений с использованием леммы Фаркаша

Поскольку ограничения временно́го разбиения подобны ограничениям пространственного разбиения, нельзя ли использовать для их решения аналогичный алгоритм? К сожалению, небольшие отличия между этими двумя задачами превращаются в большие технические различия в методах их решения. Алгоритм 11.43 просто находит $\mathbf{C}_1$, $\mathbf{c}_1$, $\mathbf{C}_2$ и $\mathbf{c}_2$, такие, что для всех $\mathbf{i}_1$ из Z^{d_1} и $\mathbf{i}_2$ из Z^{d_2}, если

$$\mathbf{F}_1\mathbf{i}_1 + \mathbf{f}_1 = \mathbf{F}_2\mathbf{i}_2 + \mathbf{f}_2,$$

то

$$\mathbf{C}_1\mathbf{i}_1 + \mathbf{c}_1 = \mathbf{C}_2\mathbf{i}_2 + \mathbf{c}_2$$

Линейные неравенства, связанные с границами циклов, используются только для определении наличия зависимости между двумя обращениями, и не более того.

При поиске решений ограничений временны́х разбиений игнорировать линейные неравенства $\mathbf{i} \prec \mathbf{i}'$ нельзя; такое игнорирование приводит к тому, что допустимым является только тривиальное решение, помещающее все итерации в один раздел. Таким образом, алгоритм поиска решений ограничений временны́х разбиений должен работать как с уравнениями, так и с неравенствами.

Обобщенная задача, которую мы хотим решить, формулируется следующим образом. Для данной матрицы $\mathbf{A}$ найти вектор $\mathbf{c}$, такой, что для всех векторов $\mathbf{x}$, таких, что $\mathbf{Ax} \geqslant \mathbf{0}$, выполняется $\mathbf{c}^T\mathbf{x} \geqslant 0$. Другими словами, мы ищем такое $\mathbf{c}$, что скалярное произведение $\mathbf{c}$ и любых координат в многограннике, определяемом неравенствами $\mathbf{Ax} \geqslant \mathbf{0}$, всегда имеет неотрицательное значение.

Помочь в решении задачи может *лемма Фаркаша* (Farkas' lemma). Пусть $\mathbf{A}$ — матрица действительных чисел размером $m \times n$, а $\mathbf{c}$ — ненулевой вектор размером

О лемме Фаркаша

Доказательство леммы можно найти во многих книгах по линейному программированию. Эта лемма, доказанная в 1901 году, является одной из *теорем альтернатив*. Все эти теоремы эквивалентны, но, несмотря на многолетние попытки, до сих пор так и не найдено простое, интуитивно понятное доказательство ни данной леммы, ни одного из ее эквивалентов.

n. Лемма Фаркаша гласит, что решение в действительных числах имеет либо *основная* система неравенств

$$\mathbf{A}\mathbf{x} \geqslant \mathbf{0}, \mathbf{c}^T\mathbf{x} < 0,$$

либо *дуальная*

$$\mathbf{A}^T\mathbf{y} = \mathbf{c}, \mathbf{y} \geqslant \mathbf{0},$$

но не обе одновременно.

Дуальная система может быть решена с применением исключения Фурье–Моцкина переменных $\mathbf{y}$. Для каждого $\mathbf{c}$, для которого имеется решение дуальной системы, лемма Фаркаша гарантирует отсутствие решения основной системы, т.е. можно доказать, что $\mathbf{c}^T\mathbf{x} \geqslant 0$ для всех $\mathbf{x}$, таких, что $\mathbf{A}\mathbf{x} \geqslant \mathbf{0}$, путем поиска решения $\mathbf{y}$ дуальной системы $\mathbf{A}^T\mathbf{y} = \mathbf{c}$ и $\mathbf{y} \geqslant \mathbf{0}$.

Алгоритм 11.59. Поиск множества допустимых максимально независимых отображений временны́х разбиений для внешнего последовательного цикла

Вход: вложенность циклов с обращениями к массиву.

Выход: максимальное множество линейно независимых отображений временны́х разбиений.

Метод: алгоритм состоит из следующих шагов.

1. Найдем все зависимые пары обращений в программе.

2. Пусть для каждой пары зависимых обращений $\mathcal{F}_1 = \langle \mathbf{F}_1, \mathbf{f}_1, \mathbf{B}_1, \mathbf{b}_1 \rangle$ в инструкции s_1, вложенной в d_1 циклов, и $\mathcal{F}_2 = \langle \mathbf{F}_2, \mathbf{f}_2, \mathbf{B}_2, \mathbf{b}_2 \rangle$ в инструкции s_2, вложенной в d_2 циклов $\langle \mathbf{C}_1, c_1 \rangle$ и $\langle \mathbf{C}_2, c_2 \rangle$, представляют собой (неизвестные) отображения временны́х разбиений s_1 и s_2 соответственно. Вспомним, что ограничения временны́х разбиений гласят, что

   - для всех $\mathbf{i}_1$ из Z^{d_1} и $\mathbf{i}_2$ из Z^{d_2}, таких, что

     а) $\mathbf{i}_1 \prec_{s_1 s_2} \mathbf{i}_2$,

     б) $\mathbf{B}_1\mathbf{i}_1 + \mathbf{b}_1 \geqslant \mathbf{0}$,

в) $\mathbf{B}_2\mathbf{i}_2 + \mathbf{b}_2 \geqslant \mathbf{0}$ и

г) $\mathbf{F}_1\mathbf{i}_1 + \mathbf{f}_1 = \mathbf{F}_2\mathbf{i}_2 + \mathbf{f}_2$,

выполняется $\mathbf{C}_1\mathbf{i}_1 + \mathbf{c}_1 \leqslant \mathbf{C}_2\mathbf{i}_2 + \mathbf{c}_2$.

Поскольку $\mathbf{i}_1 \prec_{s_1 s_2} \mathbf{i}_2$ — дизъюнктивное объединение ряда условий, можно создать систему ограничений для каждого условия и решить каждое из них отдельно, как описано далее.

а) Аналогично шагу 2, *а* из алгоритма 11.43, применим исключение Гаусса к уравнениям

$$\mathbf{F}_1\mathbf{i}_1 + \mathbf{f}_1 = \mathbf{F}_2\mathbf{i}_2 + \mathbf{f}_2$$

для приведения вектора

$$\begin{bmatrix} \mathbf{i}_1 \\ \mathbf{i}_2 \\ 1 \end{bmatrix}$$

к некоторому вектору неизвестных $\mathbf{x}$.

б) Пусть $\mathbf{c}$ — все неизвестные в отображениях раздела. Выразим линейные неравенства ограничений, связанных с отображениями, в виде

$$\mathbf{c}^T\mathbf{D}\mathbf{x} \geqslant \mathbf{0}$$

для некоторой матрицы $\mathbf{D}$.

в) Запишем предшествующие ограничения индексных переменных цикла и границы цикла как

$$\mathbf{A}\mathbf{x} \geqslant \mathbf{0}$$

для некоторой матрицы $\mathbf{A}$.

г) Применим лемму Фаркаша. Поиск $\mathbf{x}$, удовлетворяющего двум приведенным выше ограничениям, эквивалентен поиску $\mathbf{y}$, такого, что

$$\mathbf{A}^T\mathbf{y} = \mathbf{D}^T\mathbf{c} \text{ и } \mathbf{y} \geqslant \mathbf{0}$$

Заметим, что здесь $\mathbf{D}^T\mathbf{c}$ соответствует $\mathbf{c}^T$ в лемме Фаркаша и что мы используем обратный вид леммы.

д) Применим исключение Фурье–Моцкина для удаления переменных $\mathbf{y}$ и выразим ограничения, накладываемые на коэффициенты $\mathbf{c}$, в виде $\mathbf{E}\mathbf{c} \geqslant \mathbf{0}$.

е) Пусть $\mathbf{E}'\mathbf{c}' \geqslant \mathbf{0}$ — система без константных членов.

3. Найдем максимальное множество линейно независимых решений системы $\mathbf{E}'\mathbf{c}' \geqslant \mathbf{0}$ с использованием алгоритма Б.1 из приложения Б. Подход этого сложного алгоритма заключается в отслеживании текущего множества решений для каждой из инструкций и в последующем инкрементном поиске более независимых решений путем вставки ограничений, которые обеспечивают линейную независимость решения как минимум для одной инструкции.

4. Из каждого найденного решения $\mathbf{c}'$ выведем одно аффинное отображение временнóго разбиения. Постоянный член получается при использовании $\mathbf{Ec} \geqslant \mathbf{0}$. □

Пример 11.60. Ограничения из примера 11.57 могут быть переписаны как

$$\begin{bmatrix} -C_{11} & -C_{12} & C_{21} & (c_2 - c_1) \end{bmatrix} \begin{bmatrix} i \\ j \\ i' \\ 1 \end{bmatrix} \geqslant \mathbf{0}$$

$$\begin{bmatrix} -1 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} i \\ j \\ i' \\ 1 \end{bmatrix} \geqslant \mathbf{0}$$

Лемма Фаркаша гласит, что эти ограничения эквивалентны следующим:

$$\begin{bmatrix} -1 \\ 0 \\ 1 \\ 0 \end{bmatrix} \begin{bmatrix} z \end{bmatrix} = \begin{bmatrix} -C_{11} \\ -C_{12} \\ C_{21} \\ c_2 - c_1 \end{bmatrix} \text{ и } z \geqslant 0$$

Решив данную систему, получаем

$$C_{11} = C_{21} \geqslant 0 \text{ и } C_{12} = c_2 - c_1 = 0$$

Обратите внимание на то, что частное решение, полученное в примере 11.57, удовлетворяет указанным ограничениям. □

11.9.8 Преобразования кода

Если существует k независимых решений ограничений временнóго разбиения вложенности циклов, то ее можно преобразовать во вложенность с k внешними

полностью переставляемыми циклами, что, в свою очередь, обеспечит возможность создания $k-1$ степеней конвейеризации или $k-1$ внутренних распараллеливаемых циклов. Кроме того, к полностью переставляемым циклам можно применить блокирование как для повышения локальности данных в однопроцессорной системе, так и для снижения синхронизации процессоров при параллельном выполнении программы.

Использование полностью переставляемых циклов

Из k независимых решений ограничений временно́го разбиения можно легко создать вложенность циклов с k внешними полностью переставляемыми циклами. Этого можно добиться, делая k-е решение k-й строкой нового преобразования. После создания аффинного преобразования можно применить алгоритм 11.45 для генерации кода.

Пример 11.61. Вспомним решения, найденные в примере 11.58 для SOR-кода:

$$\begin{bmatrix} 1 \\ 0 \end{bmatrix} \text{ и } \begin{bmatrix} 1 \\ 1 \end{bmatrix}$$

Делая первое решение первой строкой, а второе, соответственно, второй, получаем преобразование

$$\begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix},$$

которое приводит к коду, показанному на рис. 11.51, *а*.

Если же сделать первой строкой второе решение, то получится преобразование

$$\begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix},$$

которое дает нам код, показанный на рис. 11.51, *в*. □

Легко видеть, что такие преобразования дают корректную последовательную программу. Первая строка разбивает все пространство итераций в соответствии с первым решением. Временны́е ограничения гарантируют, что такая декомпозиция не нарушает никакие зависимости данных. Затем мы разбиваем итерации в каждом внешнем цикле в соответствии со вторым решением. Такое разбиение также должно быть допустимым, поскольку мы работаем с подмножествами исходного пространства итераций. То же самое относится и к остальным строкам матрицы. Поскольку мы можем упорядочить решения произвольным образом, циклы полностью переставляемы.

Использование конвейеризации

Цикл с k внешними полностью переставляемыми циклами легко преобразовать в код с $k-1$ степенью конвейерного параллелизма.

Пример 11.62. Вернемся к нашему SOR-примеру. После того как циклы преобразованы в полностью переставляемые, мы знаем, что итерация $[i_1, i_2]$ может быть выполнена при условии выполнения итераций $[i_1, i_2 - 1]$ и $[i_1 - 1, i_2]$. Гарантировать такой порядок выполнения в конвейере можно следующим образом. Назначим итерацию i_1 процессору p_1. Каждый процессор выполняет итерации во внутреннем цикле в исходном последовательном порядке, гарантируя, таким образом, что итерация $[i_1, i_2]$ выполняется после итерации $[i_1, i_2 - 1]$. Кроме того, потребуем, чтобы процессор p ожидал сигнала от процессора $p-1$ о том, что он выполнил итерацию $[p-1, i_2]$, перед тем как приступить к выполнению итерации $[p, i_2]$. Такой метод генерирует конвейеризованный код, показанный на рис. 11.52, *а* и *б*, из полностью переставляемых циклов, показанных на рис. 11.51, *а* и *в* соответственно. □

В общем случае имеется k внешних полностью переставляемых циклов, причем итерация со значениями индексов $(i_1, \ldots, i_k)$ может быть выполнена без нарушения зависимостей данных при условии выполненности итераций

$$[i_1 - 1, i_2, \ldots, i_k], [i_1, i_2 - 1, i_3, \ldots, i_k], \ldots, [i_1, \ldots, i_{k-1}, i_k - 1]$$

Можно назначить части разбиения из первых $k-1$ измерений пространства итераций $O\left(n^{k-1}\right)$ процессорам следующим образом. Каждый процессор отвечает за одно множество итераций, у которых индексы $k-1$ измерений совпадают, а значения k-го индекса принимают все возможные значения. Каждый процессор выполняет итерации в k-м цикле последовательно. Процессор, соответствующий значениям $[p_1, p_2, \ldots, p_{k-1}]$ первых $k-1$ индексов циклов, может выполнить итерацию i в k-м цикле только по получении сигналов от процессоров

$$[p_1 - 1, p_2, \ldots, p_{k-1}], \ldots, [p_1, \ldots, p_{k-2}, p_{k-1} - 1]$$

о том, что они выполнили свои i-е итерации в k-м цикле.

Волновое распараллеливание

Легко также сгенерировать $k-1$ внутренних распараллеливаемых циклов из вложенности с k внешними полностью переставляемыми циклами. Хотя конвейеризация предпочтительнее, данная информация приводится здесь для полноты изложения.

Мы разбиваем вычисления во вложенности циклов с k внешними полностью переставляемыми циклами с использованием новой индексной переменной i', где

i' определяется как некоторая комбинация всех индексов k циклов. Например, одной из таких комбинаций является $i' = i_1 + \cdots + i_k$.

Мы создаем внешний последовательный цикл, который итерирует i' разделов разбиения в возрастающем порядке; вычисления в каждом из разделов упорядочены, как и ранее. Первые $k - 1$ циклов в каждом разделе гарантированно распараллеливаемы. Интуитивно в двумерном пространстве итераций такое преобразование при выполнении внешнего цикла группирует итерации вдоль 135-градусных диагоналей. Эта стратегия гарантирует, что итерации в каждой итерации внешнего цикла не будут иметь зависимостей данных.

Блокирование

Полностью переставляемая вложенность циклов глубиной k может быть блокирована в k размерностях. Вместо назначения итераций процессорам на основе значения индексов внешнего или внутреннего цикла мы можем агрегировать блоки итераций в один модуль. Блокирование полезно как для повышения степени локальности данных, так и для минимизации накладных расходов конвейеризации.

Предположим, что имеется двумерная полностью переставляемая вложенность циклов, как на рис. 11.55, *а*, и мы хотим разбить вычисления на $b \times b$ блоков. Порядок выполнения блочной версии кода показан на рис. 11.56, а сам код — на рис. 11.55, *б*.

```
for (i=0; i<n; i++)
    for (j=1; j<n; j++) {
        <S>
    }
```

а) Простая вложенность циклов

```
for (ii = 0; ii<n; i+=b)
    for (jj = 0; jj<n; jj+=b)
        for (i = ii*b; i <= min(ii*b-1, n); i++)
            for (j = ii*b; j <= min(jj*b-1, n); j++) {
                <S>
            }
```

б) Блочная версия вложенности циклов

Рис. 11.55. Двумерная вложенность циклов и ее блочная версия

Если назначить каждый блок одному процессору, то передача данных от одной итерации к другой в пределах блока не потребует межпроцессорного взаимодействия. В качестве альтернативы можно огрубить уровень модульности конвейеризации, назначив одному процессору столбец блоков. Заметим, что каждый про-

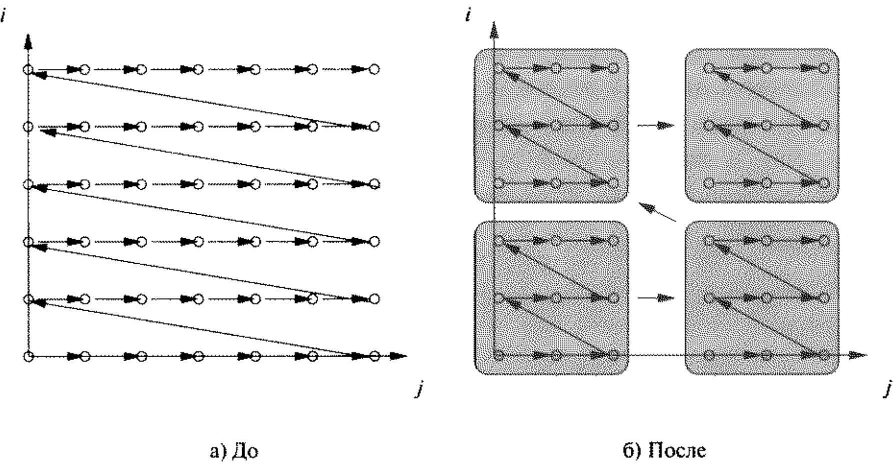


Рис. 11.56. Порядок выполнения до и после блокирования вложенности циклов глубиной 2

цессор синхронизируется со своими предшественниками и преемниками только на границах блоков. Таким образом, еще одно достоинство блокирования заключается в том, что программа требует взаимодействия обращений к данным только на границах блока с соседними блоками. Значения, являющиеся внутренними по отношению к блоку, обрабатываются только одним процессором.

Пример 11.63. Воспользуемся реальным численным алгоритмом – разложением Холецкого (Cholesky decomposition) – для иллюстрации того, как алгоритм 11.59 обрабатывает вложенности циклов с конвейерным параллелизмом. Показанный на рис. 11.57 код реализует алгоритм, обрабатывающий двумерный массив данных за время $O\left(n^3\right)$. Выполняемое пространство итераций представляет собой треугольную пирамиду, поскольку итерации по j выполняются только до значения индекса внешнего цикла i, а по k – только до значения j. Цикл содержит четыре инструкции, вложенные в разные циклы.

В результате применения алгоритма 11.59 к этой программе находятся три приемлемых временны́х измерения. Он включает все операции, некоторые из которых изначально находились во вложенностях глубиной 1 и 2 в трехмерной, полностью переставляемой вложенности циклов. Полученный код (вместе с отображениями) показан на рис. 11.58.

Программа генерации кода предохраняет от выполнения операций за пределами границ исходных циклов, гарантируя, что новая программа выполняет только те операции, которые были в исходном коде. Можно конвейеризовать этот код, отображая трехмерную структуру на двумерное пространство процессоров. Итерации $(i2, j2, k2)$ назначаются процессору с идентификатором $(i2, j2)$. Каждый

```
for (i = 1; i <= N; i++) {
    for (j = 1; j <= i-1; j++) {
        for (k = 1; k <= j-1; k++)
            X[i,j] = X[i,j] - X[i,k] * X[j,k];
        X[i,j] = X[i,j] / X[j,j];
    }
    for (m = 1; m <= i-1; m++)
        X[i,i] = X[i,i] - X[i,m] * X[i,m];
    X[i,i] = sqrt(X[i,i]);
}
```

Рис. 11.57. Разложение Холецкого

```
for (i2 = 1; i2 <= N; i2++)
    for (j2 = 1; j2 <= i2; j2++) {
        /* Начало кода для процессора (i2,j2) */
        for (k2 = 1; k2 <= i2; k2++) {

            // Отображение: i2 = i, j2 = j, k2 = k
            if (j2<i2 && k2<j2)
                X[i2,j2] = X[i2,j2] - X[i2,k2] * X[j2,k2];

            // Отображение: i2 = i, j2 = j, k2 = j
            if (j2==k2 && j2<i2)
                X[i2,j2] = X[i2,j2] / X[j2,j2];

            // Отображение: i2 = i, j2 = i, k2 = m
            if (i2==j2 && k2<i2)
                X[i2,i2] = X[i2,i2] - X[i2,k2] * X[i2,k2];

            // Отображение: i2 = i, j2 = i, k2 = i
            if (i2==j2 && j2==k2)
                X[k2,k2] = sqrt(X[k2,k2]);
        }
        /* Конец кода для процессора (i2,j2) */
    }
```

Рис. 11.58. Код на рис. 11.57, переписанный как полностью переставляемая вложенность циклов

процессор выполняет наиболее глубоко вложенный цикл — с индексом $k2$. Прежде чем выполнить k-ю итерацию, процессор ожидает сигналов от процессоров с идентификаторами $(i2-1, j2)$ и $(i2, j2-1)$. После выполнения итерации он, в свою очередь, посылает сигналы процессорам $(i2+1, j2)$ и $(i2, j2+1)$. □

11.9.9 Параллелизм с минимальной синхронизацией

В последних трех разделах были описаны три мощных алгоритма распараллеливания. Алгоритм 11.43 находит параллелизм, не требующий синхронизации; алгоритм 11.54 находит параллелизм, требующий константного количества синхронизаций, а алгоритм 11.59 находит конвейеризуемый параллелизм, требующий $O(n)$ синхронизаций, где n — количество итераций во внешнем цикле. В качестве первого приближения наша цель состоит в распараллеливании как можно большего количества вычислений при внесении как можно меньшего количества синхронизаций.

Приведенный ниже алгоритм 11.64 находит в программе все степени параллелизма, начиная с наиболее крупномасштабного. На практике, чтобы распараллелить код для многопроцессорной системы, нам не требуется работать с параллелизмом на всех возможных уровнях. Мы начинаем с внешних циклов и идем вглубь до тех пор, пока не будут распараллелены все вычисления, а все процессоры — полностью загружены.

Алгоритм 11.64. Поиск всех степеней максимально крупномодульного параллелизма в программе

ВХОД: распараллеливаемая программа.

ВЫХОД: распараллеленная версия исходной программы.

МЕТОД: выполняем следующие действия.

1. Находим максимальную степень параллелизма, не требующего синхронизации, путем применения к программе алгоритма 11.43.

2. Находим максимальную степень параллелизма, требующего $O(1)$ синхронизаций, путем применения алгоритма 11.54 к каждому пространственному разбиению, найденному на шаге 1. (Если параллелизм, не требующий синхронизации, в программе не обнаружен, вся программа рассматривается как единственная часть пространственного разбиения.)

3. Находим максимальную степень параллелизма, требующего $O(n)$ синхронизаций. Применяем алгоритм 11.59 к каждой из частей разбиений, найденных на шаге 2, для поиска конвейеризуемого параллелизма. Затем применим алгоритм 11.54 к каждой из частей, назначенных каждому процессору (или к телу последовательного цикла, если конвейеризуемый параллелизм не обнаружен).

4. Находим максимальную степень параллелизма с последовательно возрастающими степенями синхронизаций, для чего рекурсивно применяем шаг 3 к вычислениям, принадлежащим каждому из пространственных разбиений, сгенерированных на предыдущем шаге. □

Пример 11.65. Вернемся к примеру 11.56. Шаги 1 и 2 алгоритма 11.64 не обнаруживают никакого параллелизма, так что нам требуется больше чем константное количество синхронизаций для распараллеливания данного кода. Применение на шаге 3 алгоритма 11.59 определяет, что имеется только один внешний цикл, который представляет собой внешний цикл в исходном коде на рис. 11.53. Таким образом, цикл не обладает конвейеризуемым параллелизмом. Во второй части шага 3 мы применяем алгоритм 11.54 для распараллеливания внутреннего цикла. Код в пределах раздела рассматривается нами как целая программа, с тем отличием, что номер раздела при этом считается символьной константой. В итоге мы обнаруживаем, что внутренний цикл распараллеливаем, а значит, код может быть распараллелен с использованием n барьеров синхронизации. □

Алгоритм 11.64 находит в программе весь возможный параллелизм на каждом уровне синхронизации. Алгоритм предпочитает схемы распараллеливания с меньшей степенью синхронизации, однако меньшая степень синхронизации не означает минимизации взаимодействия. Далее мы рассмотрим две модификации алгоритма, которые призваны преодолеть это слабое место.

Учет стоимости взаимодействия

Если не найден параллелизм, не требующий синхронизации, шаг 2 алгоритма 11.64 распараллеливает каждый сильно связанный компонент независимо. Однако может оказаться возможным распараллелить ряд компонентов без синхронизации и взаимодействия. Одно из решений состоит в жадном поиске параллелизма, не требующего синхронизации, среди подмножеств графа зависимостей программы, которые совместно используют большое количество данных.

Если между сильно связанными компонентами требуется взаимодействие, то можно заметить, что одно взаимодействие может оказаться существенно дороже других. Например, стоимость перемещения матрицы значительно больше, чем простой обмен информацией между двумя соседними процессорами. Предположим, что s_1 и s_2 — инструкции в двух различных сильно связанных компонентах, обращающихся к одним и тем же данным в итерациях $\mathbf{i}_1$ и $\mathbf{i}_2$ соответственно. Если мы не можем найти отображения разделов $\langle \mathbf{C}_1, \mathbf{c}_1 \rangle$ и $\langle \mathbf{C}_2, \mathbf{c}_2 \rangle$ для инструкций s_1 и s_2, такие, что

$$\mathbf{C}_1\mathbf{i}_1 + \mathbf{c}_1 - \mathbf{C}_2\mathbf{i}_2 - \mathbf{c}_2 = \mathbf{0},$$

то вместо удовлетворения указанного ограничения можно попытаться удовлетворить ограничению

$$\mathbf{C}_1\mathbf{i}_1 + \mathbf{c}_1 - \mathbf{C}_2\mathbf{i}_2 - \mathbf{c}_2 \leqslant \delta,$$

где δ — малая константа.

Компромисс между синхронизацией и взаимодействием

Иногда оказывается более выгодным выполнить большее количество синхронизаций, минимизируя при этом обмен информацией. В примере 11.66 рассматривается одна такая ситуация. Таким образом, если мы не можем распараллелить код с использованием взаимодействия только между соседними строго связанными компонентами, то вместо независимого распараллеливания каждого компонента мы должны попытаться конвейеризовать вычисления. Как показано в примере 11.66, конвейеризация может быть применена к последовательности циклов.

Пример 11.66. Для алгоритма интегрирования из примера 11.49 мы показали, что независимая оптимизация первой и второй вложенностей циклов обнаруживает параллелизм в каждой вложенности. Однако такая схема требует пересылки матрицы между циклами, что приводит к пересылке $O\left(n^2\right)$ данных. Если для поиска конвейеризуемого параллелизма использовать алгоритм 11.59, то обнаружится, что вся программа может быть преобразована в полностью переставляемую вложенность циклов, как показано на рис. 11.59. Затем для снижения накладных расходов, связанных с взаимодействием, можно применить блокирование. Такая схема может привести к использованию $O(n)$ синхронизаций, но потребует существенно меньшего количества взаимодействий. □

```
for (j = 0; j < n; j++)
    for (i = 1; i < n+1; i++) {
        if (i < n) X[i,j] = f(X[i,j] + X[i-1,j])
        if (j > 0) X[i-1,j] = g(X[i-1,j],X[i-1,j-1]);
    }
```

Рис. 11.59. Полностью переставляемая вложенность циклов для кода из примера 11.49

11.9.10 Упражнения к разделу 11.9

Упражнение 11.9.1. В разделе 11.9.4 мы рассматривали возможность использования диагоналей, отличных от горизонтальной и вертикальной осей, для конвейеризации кода на рис. 11.51. Напишите код, аналогичный циклам на рис. 11.52, для диагоналей под углом а) 135°, б) 120°.

Упражнение 11.9.2. Код на рис. 11.55, *б* можно упростить, если n нацело делится на b. Перепишите код при выполнении этого условия.

Упражнение 11.9.3. На рис. 11.60 приведена программа для вычисления первых 100 строк треугольника Паскаля (т.е. $P[i,j]$ представляет количество способов выбрать j предметов из i, $0 \leqslant j \leqslant i < 100$).

а) Перепишите код в виде единой, полностью переставляемой вложенности циклов.

б) Воспользуйтесь конвейером из 100 процессоров для реализации этого кода. Напишите код для каждого процессора p с использованием переменной p и укажите необходимые синхронизации.

в) Перепишите код с использованием квадратных блоков размером 10×10 итераций каждый. Поскольку итерации образуют треугольник, всего потребуется $1 + 2 + \cdots + 10 = 55$ блоков. Приведите код для процессора (p_1, p_2), которому назначен p_1-й блок в направлении i и p_2-й — в направлении j, с использованием переменных p_1 и p_2.

```
for (i=0; i<100; i++) {
    P[i,0] = 1; /* s1 */
    P[i,i] = 1; /* s2 */
}
for (i=2; i<100; i++)
    for (j=1; j<i; j++)
        P[i,j] = P[i-1,j-1] + P[i-1,j]; /* s3 */
```

Рис. 11.60. Вычисление треугольника Паскаля

! **Упражнение 11.9.4.** Повторите упражнение 11.9.2 для кода на рис. 11.61. Обратите внимание, что итерации в этой задаче образуют куб со стороной 10. Таким образом, в части *в* блоки имеют размер $10 \times 10 \times 10$, а их общее количество — 1000 штук.

! **Упражнение 11.9.5.** Применим алгоритм 11.59 к простому примеру ограничений временнóго разбиения. Далее предполагается, что вектор $\mathbf{i}_1$ представляет собой (i_1, j_1), а вектор $\mathbf{i}_2$ — (i_2, j_2). Условие $\mathbf{i}_1 \prec_{s_1 s_2} \mathbf{i}_2$ состоит из дизъюнкции

а) $i_1 < i_2$ или

б) $i_1 = i_2$ и $j_1 < j_2$.

```
for (i=0; i<100; 1++) {
    A[i, 0,0] = B1[i]; /* s1 */
    A[i,99,0] = B2[i]; /* s2 */
}
for (j=1; j<99; j++) {
    A[ 0,j,0] = B3[j]; /* s3 */
    A[99,j,0] = B4[j]; /* s4 */
}
for (i=0; i<99; i++)
    for (j=0; j<99; j++)
        for (k=1; k<100; k++)
            A[i,j,k] = 4*A[i,j,k-1] + A[i-1,j,k-1] +
                A[i+1,j,k-1] + A[i,j-1,k-1] +
                A[i,j+1,k-1]; /* s5 */
```

Рис. 11.61. Код к упражнению 11.9.4

Прочими уравнениями и неравенствами являются

$$2i_1 + j_1 - 10 \geqslant 0$$
$$i_2 + 2j_2 - 20 \geqslant 0$$
$$i_1 = i_2 + j_2 - 50$$
$$j_1 = j_2 + 40$$

Наконец, неравенство временнóго разбиения с неизвестными c_1, d_1, e_1, c_2, d_2 и e_2 имеет вид

$$c_1 i_1 + d_1 j_1 + e_1 \leqslant c_2 i_2 + d_2 j_2 + e_2.$$

а) Решите ограничения временнóго разбиения для случая 1, т.е. для случая, когда $i_1 < i_2$. В частности, исключите максимально возможное количество переменных из i_1, j_1, i_2 и j_2 и создайте матрицы D и A, как в алгоритме 11.59. Затем примените лемму Фаркаша к получившимся матричным неравенствам.

б) Повторите часть *а* для случая 2, когда $i_1 = i_2$ и $j_1 < j_2$.

11.10 Оптимизации локальности

Производительность процессора — как в однопроцессорной, так и в многопроцессорной системе — существенно зависит от поведения кэша. Промахи кэша могут потребовать для обработки десятков тактов, так что большое количество промахов кэша может существенно снизить производительность процессора.

В контексте многопроцессорной системы с общей шиной к дальнейшему снижению производительности процессоров могут привести конфликты шины.

Как мы увидим, даже если мы всего лишь хотим повысить локальность в однопроцессорной системе, алгоритм аффинного разбиения для распараллеливания применим и как средство для обнаружения возможных преобразований циклов. В этом разделе будут описаны три методики повышения локальности данных в однопроцессорных и многопроцессорных системах.

1. Повышение временнóй локальности вычисленных результатов путем использования их сразу же после генерации. Это достигается разделением вычислений на независимые части и выполнением всех зависимых операций в пределах части как можно ближе друг к другу.

2. *Сжатие массива* (array contraction) снижает размерность массива и количество ячеек памяти, к которым выполняется обращение. Сжатие массива можно применить, только если в каждый момент времени используется только одна ячейка массива.

3. Помимо повышения временнóй локальности вычисленных результатов необходимо оптимизировать и их пространственную локальность, а также как временнýю, так и пространственную локальность данных, предназначенных только для чтения. Вместо поочередного выполнения каждой части кода ряд частей чередуется таким образом, что повторные использования в разных частях оказываются расположенными ближе друг к другу.

11.10.1 Временнáя локальность вычисляемых данных

Алгоритм аффинного разбиения размещает вместе все зависимые операции; при последовательном выполнении полученных частей повышается временнáя локальность вычисляемых данных. Вернемся к многосеточному примеру, который рассматривался в разделе 11.7.1. Применение алгоритма 11.43 для распараллеливания кода на рис. 11.23 обнаруживает две степени параллелизма. Код на рис. 11.24 содержит два внешних цикла, которые последовательно итерируют независимые части. Этот преобразованный код имеет повышенную временнýю локальность, поскольку вычисляемые результаты используются здесь же, в этой же самой итерации.

Таким образом, даже если наша цель заключается в оптимизации для последовательного выполнения, оказывается выгодным использование распараллеливания для поиска связанных операций и размещения их рядом друг с другом. Алгоритм, который используется здесь, подобен алгоритму 11.64, который обнаруживает все уровни параллелизма, начиная с охватывающего внешнего цикла. Как говорилось в разделе 11.9.9, если на каком-то уровне не удается найти параллелизм, не требующий синхронизации, то алгоритм распараллеливает сильно

связанные компоненты по отдельности. Такое распараллеливание имеет тенденцию к повышению обмена информацией между процессорами. Таким образом, мы объединяем раздельно распараллеленные сильно связанные компоненты, если они разделяют одни и те же повторные использования.

11.10.2 Сжатие массива

Оптимизация сжатия массива предоставляет еще одну иллюстрацию компромисса между памятью и параллельностью, о котором мы говорили в разделе 10.2.3 в контексте параллелизма на уровне команд. Так же как применение большего количества регистров обеспечивает более высокий уровень параллелизма на уровне команд, так и использование большего количества памяти повышает параллелизм на уровне циклов. Как показано в многосеточном примере в разделе 11.7.1, преобразование вре́менной скалярной переменной в массив позволяет разным итерациям использовать разные экземпляры вре́менных переменных и выполняться одновременно. И наоборот, при последовательном выполнении, в каждый момент времени работающем с одним элементом массива, массив можно сжать, заменив его скаляром, и заставить каждую итерацию работать с одной и той же ячейкой памяти.

В преобразованной многосеточной программе на рис. 11.24 каждая итерация внутреннего цикла производит и потребляет различные элементы массивов AP, AM, T и строку массива D. Если эти массивы не используются вне рассматриваемого фрагмента, итерации могут последовательно использовать одну и ту же память для данных вместо того, чтобы размещать значения в разных элементах и строках массивов. На рис. 11.62 показан результат снижения размерности массивов. Такой код работает быстрее исходного, поскольку он считывает и записывает меньшее количество данных. В частности, в случае, когда массив сводится к скалярной переменной, ее можно назначить регистру и полностью устранить необходимость обращения к памяти.

Чем меньше количество используемой памяти, тем меньше доступный параллелизм. Итерации в преобразованном коде на рис. 11.62 связаны зависимостями данных и больше не могут выполняться параллельно. Для распараллеливания кода между P процессорами можно расширить каждую скалярную переменную в P раз и предоставить каждому процессору собственную копию переменной. Таким образом, величина увеличения количества памяти непосредственно связана с количеством используемого параллелизма.

Существуют три основных повода для поиска возможностей сжатия массивов.

1. Высокоуровневые языки программирования для научных приложений, такие как Matlab и Fortran 90, поддерживают операции на уровне массивов. Каждое подвыражение операции с массивами приводит к появлению вре́менного массива. Поскольку массивы могут иметь большие размеры,

```
for (j = 2, j <= j1, j++)
    for (i = 2, i <= i1, i++) {
        AP           = ...;
        T            = 1.0/(1.0 +AP);
        D[2]         = T*AP;
        DW[1,2,j,i] = T*DW[1,2,j,i];
        for (k=3, k <= k1-1, k++) {
           AM           = AP;
           AP           = ...;
           T            = ...AP -AM*D[k-1]...;
           D[k]         = T*AP;
           DW[1,k,j,i] = T*(DW[1,k,j,i]+DW[1,k-1,j,i])...;
        }
        ...
        for (k=k1-1, k>=2, k--)
            DW[1,k,j,i] = DW[1,k,j,i] +D[k]*DW[1,k+1,j,i];
    }
```

Рис. 11.62. Код на рис. 11.23 после разбиения (рис. 11.24) и сжатия массивов

каждая операция с ними, такая как сложение или умножение, будет требовать большого количества чтений и записи ячеек памяти при относительно небольшом количестве арифметических операций. Важно переупорядочить операции так, чтобы данные потреблялись сразу же после их получения, и сжать массивы до скалярных переменных.

2. Суперкомпьютеры 80-х и 90-х годов были векторными машинами, так что масса научных приложений была оптимизирована именно для этих машин. Несмотря на наличие векторизующих компиляторов многие программисты продолжают писать код, выполняющий операции одновременно над целыми векторами. Примером такого стиля служит рассматривавшийся нами многосеточный код.

3. Возможности сжатия массивов привносятся и компилятором. Как иллюстрирует переменная T в многосеточном примере, компилятор расширяет массивы для повышения степени параллелизма. Мы должны выполнять сжатие там, где такое расширение не является необходимым.

Пример 11.67. Выражение с массивами $Z = W + X + Y$ транслируется в код

```
for (i=0; i<n; i++) T[i] = W[i] + X[i];
for (i=0; i<n; i++) Z[i] = T[i] + Y[i];
```

Если переписать код как

```
for (i=0; i<n; i++) { T = W[i] + X[i]; Z[i] = T + Y[i] }
```

это может значительно увеличить скорость его выполнения. Конечно, на уровне исходного текста С нам не надо использовать даже вре́менную переменную T, так как можно записать присваивание $Z[i]$ как единую инструкцию. Однако здесь мы пытаемся смоделировать уровень промежуточного кода, на котором выполняет свои операции векторный процессор. □

Алгоритм 11.68. Сжатие массива

ВХОД: программа, преобразованная алгоритмом 11.64.

ВЫХОД: эквивалентная программа с массивом с уменьшенным количеством размерностей.

МЕТОД: размерность массива может быть сведена к единственному элементу, если выполняются следующие условия.

1. Каждая независимая часть кода использует только один элемент массива.
2. Значение, которое хранилось в этом элементе до входа в часть, в этой части не используется.
3. Значение элемента не активно при выходе из части.

Определяем сжимаемые измерения — которые удовлетворяют трем указанным условиям — и заменяем их единственным элементом. □

В алгоритме 11.68 предполагается, что первоначально программа была трансформирована алгоритмом 11.64, который размещает все зависимые операции в одной части и последовательно выполняет части программы. Алгоритм находит те массивы, диапазоны активности элементов которых в разных итерациях не пересекаются. Если эти переменные не являются активными после выхода из цикла, алгоритм сжимает такие массивы, заставляя процессор работать с одним и тем же скаляром. После сжатия может оказаться необходимым выборочно расширить скаляры до массивов для обеспечения распараллеливания и иных оптимизаций локальности.

В данном случае требуется более сложный анализ активности переменных, чем рассматривавшийся в разделе 9.2.5. Если массив объявлен как глобальная переменная или представляет собой параметр процедуры, то, чтобы убедиться, что он не используется после выхода из цикла, требуется выполнить межпроцедурный анализ. Кроме того, необходимо определить активность каждого отдельного элемента массива — консервативное рассмотрение массива как скаляра будет слишком неточным.

11.10.3 Чередование частей

Различные части цикла зачастую считывают одни и те же данные, или читают и пишут одни и те же строки кэша. В этом и двух последующих разделах мы рассмотрим оптимизацию в случае повторного использования, пересекающего границы частей.

Повторное использование во внутренних блоках

Примем простую модель, в которой данные находятся в кэше, если они повторно используются в пределах небольшого количества итераций. Если внутренний цикл имеет большие или неизвестные границы, то выгода от локальности достигается только при повторном использовании, пересекающем границы итераций. Блокирование создает внутренние циклы с небольшими известными границами, позволяя воспользоваться повторными использованиями как в пределах блоков, так и с пересечением их границ. Таким образом, блокирование позволяет извлечь выгоду из повторного использования с большим количеством размерностей.

Пример 11.69. Рассмотрим код умножения матриц на рис. 11.5 и его блочную версию на рис. 11.7. Умножение матриц содержит повторные использования вдоль всех трех размерностей пространства итераций. В исходном коде внутренний цикл имеет n итераций, где n — неизвестное значение, которое может быть очень большим. Наша простая модель предполагает, что в кэше могут находиться только те данные, которые повторно используются итерациями внутреннего цикла.

В блочной версии три внутренних цикла выполняют трехмерный блок вычислений с B итерациями вдоль каждой стороны блока. Размер блока B выбирается компилятором достаточно малым для того, чтобы все строки кэша, считываемые и записываемые в пределах блока вычислений, могли разместиться в кэше. Таким образом, данные, повторно используемые итерациями третьего внешнего цикла, могут быть найдены в кэше. □

Будем называть внутреннее множество циклов с небольшими известными границами *внутренним блоком* (innermost block). Желательно, чтобы внутренний блок по возможности включал все размерности пространства итераций с повторными использованиями. Максимизация длин сторон блока не так важна. В случае примера с умножением матриц трехмерное блокирование снижает количество обращений к данным для каждой матрицы в B^2 раз. При наличии повторных использований лучше иметь дело с блоками высокой размерности и малой стороной, чем с большими блоками малой размерности.

Можно оптимизировать локальность внутренней полностью переставляемой вложенности циклов путем блокирования подмножеств циклов, разделяющих повторные использования. Можно обобщить понятие блокирования, чтобы воспользоваться повторными использованиями, обнаруженными среди итераций внешних параллельных циклов. Заметим, что блокирование, в первую очередь, чередует

выполнение небольшого количества экземпляров внутреннего цикла. При умножении матриц каждый экземпляр внутреннего цикла вычисляет один элемент результирующего массива; всего таких элементов — n^2. Блокирование чередует выполнение блоков экземпляров, вычисляя за один раз B итераций из каждого экземпляра. Аналогично можно чередовать итерации параллельных циклов, чтобы получить выгоду от повторных использований между ними.

Ниже будут определены два примитива, которые могут уменьшить дистанцию между повторными использованиями в разных итерациях. Мы будем многократно применять эти примитивы, начиная с внешнего цикла, до тех пор, пока все повторные использования не будут перемещены по соседству друг к другу во внутреннем блоке.

Чередование внутренних циклов в параллельном цикле

Рассмотрим ситуацию, когда внешний параллелизуемый цикл содержит внутренний цикл. Чтобы воспользоваться повторными использованиями из разных итераций внешнего цикла, мы чередуем выполнения фиксированного количества экземпляров внутреннего цикла, как показано на рис. 11.63. Создавая двумерные внутренние блоки, это преобразование сокращает расстояние между повторными использованиями последовательных итераций внешнего цикла.

```
for (i=0; i<n; i++)
  for (j=0; j<n; j++)
    <S>
```

а) Исходная программа

```
for (ii=0; ii<n; ii+=4)
  for (j=0; j<n; j++)
    for (i=ii; i<min(n, ii+4); i++)
      <S>
```

б) Преобразованный код

Рис. 11.63. Чередование четырех экземпляров внутреннего цикла

Шаг, превращающий цикл

```
for (i=0; i<n; i++)
    <S>
```

в

```
for (ii=0; ii<n; ii+=4)
    for (i=ii; i<min(n, ii+4); i++)
        <S>
```

известен как *располосование* (stripmining). В случае, когда внешний цикл на рис. 11.63 имеет небольшие известные границы, нам не надо выполнять описанные преобразования — можно просто переставить два цикла исходной программы.

Чередование инструкций в параллельном цикле

Рассмотрим ситуацию, когда параллелизуемый цикл содержит последовательность инструкций $s_1, s_2, \ldots, s_m$. Если некоторые из этих инструкций сами по себе являются циклами, то инструкции из соседних итераций могут быть разделены большим количеством операций. Повторным использованием между итерациями вновь можно воспользоваться при помощи чередования их выполнений, как показано на рис. 11.64. Такое преобразование *распределяет* обработанный цикл по инструкциям. Как и ранее, если внешний цикл имеет небольшое фиксированное количество итераций, можно просто распределить исходный цикл по всем инструкциям, не прибегая к располосованию.

```
for (i=0; i<n; i++) {
  <S1>
  <S2>
  ...
}
```

а) Исходная программа

```
for (ii=0; ii<n; ii+=4) {
  for (i=ii; i<min(n,ii+4); i++)
    <S1>
  for (i=ii; i<min(n,ii+4); i++)
   <S2>
  ...
}
```

б) Преобразованный код

Рис. 11.64. Преобразование, чередующее инструкции

Используем для обозначения выполнения инструкции s_i в итерации j запись $s_i(j)$. Вместо исходного последовательного порядка выполнения, показанного на рис. 11.65, *а*, код выполняется в порядке, показанном на рис. 11.65, *б*.

Пример 11.70. Вернемся к многосеточному примеру и покажем, как можно воспользоваться повторными использованиями между итерациями внешних параллельных циклов. Заметим, что обращения $DW[1,k,j,i]$, $DW[1,k-1,j,i]$ и $DW[1,k+1,j,i]$ во внутренних циклах в коде на рис. 11.62 обладают малой пространственной локальностью. Анализ повторного использования, описанный в разделе 11.5, указывает, что цикл с индексом i является носителем пространственной локальности, а цикл с индексом k — группового повторного использования. Цикл с индексом k и так является внутренним, так что нас интересует чередование операций с DW в блоке частей с последовательными значениями i.

Применим преобразование для получения чередующихся инструкций в цикле, что даст нам код, показанный на рис. 11.66, а затем применим то же преобразование к внутреннему циклу и получим код на рис. 11.67. Заметим, что в силу чередования B итераций из цикла с индексом i нам требуется преобразование переменных AP, AM и T в массивы, способные одновременно хранить B результатов. □

$$\begin{array}{llll}
s_1(0), & s_2(0), & \ldots, & s_m(0), \\
s_1(1), & s_2(1), & \ldots, & s_m(1), \\
s_1(2), & s_2(2), & \ldots, & s_m(2), \\
s_1(3), & s_2(3), & \ldots, & s_m(3), \\
s_1(4), & s_2(4), & \ldots, & s_m(4), \\
s_1(5), & s_2(5), & \ldots, & s_m(5), \\
s_1(6), & s_2(6), & \ldots, & s_m(6), \\
s_1(7), & s_2(7), & \ldots, & s_m(7), \\
\ldots, & & &
\end{array}$$

а) Исходный порядок

$$\begin{array}{llll}
s_1(0), & s_1(1), & s_1(2), & s_1(3), \\
s_2(0), & s_2(1), & s_2(2), & s_2(3), \\
\ldots, & & & \\
s_m(0), & s_m(1), & s_m(2), & s_m(3), \\
s_1(4), & s_1(5), & s_1(6), & s_1(7), \\
s_2(4), & s_2(5), & s_2(6), & s_2(7), \\
\ldots, & & & \\
s_m(4), & s_m(5), & s_m(6), & s_m(7), \\
\ldots, & & &
\end{array}$$

б) Преобразованный порядок

Рис. 11.65. Распределение обработанного цикла

11.10.4 Алгоритмы оптимизации локальности

Алгоритм 11.71 оптимизирует локальность для однопроцессорной системы, а алгоритм 11.72 оптимизирует как локальность, так и параллелизм для многопроцессорной системы.

Алгоритм 11.71. Оптимизация локальности данных в однопроцессорной системе

ВХОД: программа с аффинными обращениями к массиву.

ВЫХОД: эквивалентная программа, максимизирующая локальность данных.

МЕТОД: выполнить следующие действия.

1. Применить алгоритм 11.64 для оптимизации временно́й локальности вычисленных результатов.

```
for (j = 2, j <= jl, j++)
    for (ii = 2, ii <= il, ii+=b) {
        for (i = ii; i <= min(ii+b-1,il); i++) {
            ib          = i-ii+1;
            AP[ib]      = ...;
            T           = 1.0/(1.0 +AP[ib]);
            D[2,ib]     = T*AP[ib];
            DW[1,2,j,i] = T*DW[1,2,j,i];
        }
        for (i = ii; i <= min(ii+b-1,il); i++) {
            for (k=3, k <= kl-1, k++)
                ib          = i-ii+1;
                AM          = AP[ib];
                AP[ib]      = ...;
                T           = ...AP[ib]-AM*D[ib,k-1]...;
                D[ib,k]     = T*AP;
                DW[1,k,j,i] = T*(DW[1,k,j,i]+DW[1,k-1,j,i])...;
            }
        ...
        for (i = ii; i <= min(ii+b-1,il); i++)
            for (k=kl-1, k>=2, k--) {
               DW[1,k,j,i] = DW[1,k,j,i] +D[iw,k]*DW[1,k+1,j,i];
            /* Конец кода, выполняемого процессором (j,i) */
        }
    }
```

Рис. 11.66. Фрагмент кода, представленный на рис. 11.23, после разбиения, сжатия массивов и блокирования

2. Применить алгоритм 11.68 для сжатия массивов там, где это возможно.

3. Определить подпространство итераций, которые могут совместно использовать одни и те же данные или строки кэша, применив методы, описанные в разделе 11.5. Для каждой инструкции определить те измерения внешнего параллельного цикла, которые содержат повторные использования.

4. Для каждого внешнего параллельного цикла, являющегося носителем повторного использования, переместить блок итераций во внутренний блок путем многократного применения примитивов чередования.

5. Применить блокирование к подмножеству измерений во внутренней полностью переставляемой вложенности циклов, являющейся носителем повторного использования.

```
for (j = 2, j <= jl, j++)
    for (ii = 2, ii <= il, ii+=b) {
        for (i = ii; i <= min(ii+b-1,il); i++) {
            ib            = i-ii+1;
            AP[ib]        = ...;
            T             = 1.0/(1.0 +AP[ib]);
            D[2,ib]       = T*AP[ib];
            DW[1,2,j,i] = T*DW[1,2,j,i];
        }
        for (k=3, k <= kl-1, k++)
            for (i = ii; i <= min(ii+b-1,il); i++) {
                ib            = i-ii+1;
                AM            = AP[ib];
                AP[ib]        = ...;
                T             = ...AP[ib]-AM*D[ib,k-1]...;
                D[ib,k]       = T*AP;
                DW[1,k,j,i] = T*(DW[1,k,j,i]+DW[1,k-1,j,i])...;
            }
        ...
        for (k=kl-1, k>=2, k--) {
            for (i = ii; i <= min(ii+b-1,il); i++)
               DW[1,k,j,i] = DW[1,k,j,i] +D[iw,k]*DW[1,k+1,j,i];
            /* Конец кода, выполняемого процессором (j,i) */
        }
    }
```

Рис. 11.67. Фрагмент кода, представленного на рис. 11.23, после разбиения, сжатия массивов, блокирования и чередования внутреннего цикла

6. Блокировать внешнюю полностью переставляемую вложенность циклов для высших уровней иерархии памяти, таких как кэш третьего уровня и физическая память.

7. При необходимости расширить скаляры и массивы до длины блоков. □

Алгоритм 11.72. Оптимизация параллелизма и локальности данных в многопроцессорной системе

ВХОД: программа с аффинными обращениями к массиву.

ВЫХОД: эквивалентная программа, максимизирующая параллелизм и локальность данных.

МЕТОД: выполнить следующие действия.

1. Применить алгоритм 11.64 для распараллеливания исходной программы и создания SPMD-программы.

2. Применить алгоритм 11.71 к полученной в шаге 1 SPMD-программе для оптимизации ее локальности. □

11.10.5 Упражнения к разделу 11.10

Упражнение 11.10.1. Выполните сжатие массивов в следующих векторных операциях:

```
for (i=0; i<n; i++) T[i] = A[i] * B[i];
for (i=0; i<n; i++) D[i] = T[i] + C[i];
```

Упражнение 11.10.2. Выполните сжатие массивов в следующих векторных операциях:

```
for (i=0; i<n; i++) T[i] = A[i] + B[i];
for (i=0; i<n; i++) S[i] = C[i] + D[i];
for (i=0; i<n; i++) E[i] = T[i] * S[i];
```

Упражнение 11.10.3. Выполните располосование внешнего цикла на полосы шириной 10:

```
for (i=n-1; i>=0; i--)
    for (j=0; j<n; j++)
```

11.11 Прочие применения аффинных преобразований

До сих пор мы сосредоточивались на машинах, имеющих архитектуру с общей памятью, но теория аффинных преобразований циклов имеет много других приложений. Можно применить аффинные преобразования к другим видам параллелизма, включая машины с распределенной памятью, векторными командами, SIMD-командами (Single Instruction Multiple Data — одна команда, много данных) или машины с одновременным выполнением нескольких команд.

11.11.1 Машины с распределенной памятью

В случае машин с распределенной памятью, в которых процессоры взаимодействуют путем пересылки сообщений друг другу, еще более важную роль играет назначение процессорам больших независимых вычислительных модулей, таких как генерируемые алгоритмом аффинного разбиения. Помимо разбиения вычислений, при компиляции требуется решение ряда других вопросов.

1. *Распределение данных.* Если процессоры работают с разными частями массива, то каждому из них требуется выделить память, достаточную для хранения только используемой части. Для определения частей массивов, используемых каждым процессором, можно применить проецирование. Входными данными в этом случае являются линейные неравенства, представляющие границы циклов, функции обращения к массивам и аффинные разбиения, отображающие итерации на идентификаторы процессоров. Исключая индексы циклов, для каждого идентификатора процессора можно найти множество используемых им элементов массива.

2. *Код взаимодействия.* Требуется генерация явного кода для отправки и получения данных процессорами. В каждой точке синхронизации необходимо

   а) определить, какие данные, находящиеся у одного процессора, требуются другим процессорам;

   б) сгенерировать код, который находит все отсылаемые данные и пакует их в буфер;

   в) сгенерировать код, который определяет данные, необходимые процессору, распаковывает полученные сообщения и перемещает полученные данные в соответствующие места в памяти.

3. *Оптимизация.* Все взаимодействия не обязательно выполняются в точках синхронизации. Предпочтительно, чтобы каждый процессор отсылал данные, как только они становятся доступными, так что другим процессорам не нужно было бы ожидать необходимых им данных. С другой стороны, с обработкой каждого сообщения связаны существенные накладные расходы, так что следует избегать генерации слишком большого их количества.

Описанные здесь методы имеют и иные применения. Например, встроенные системы специального назначения могут использовать сопроцессоры для того, чтобы разгрузить основной процессор. Или вместо выборки данных в кэш встроенная система может использовать отдельный контроллер для загрузки и выгрузки данных из кэша или иных буферов данных, в то время как процессор работает с другими данными. В этих случаях для генерации кода и перемещения данных могут быть использованы методы, подобные описанным.

11.11.2 Процессоры с одновременным выполнением нескольких команд

Аффинные преобразования циклов могут использоваться и для оптимизации производительности машин с одновременным выполнением нескольких команд.

Как говорилось в разделе 10.5, производительность программно конвейеризованного цикла ограничена двумя факторами: наличием циклов в ограничениях предшествования и использованием критичных ресурсов. Путем изменения внутреннего цикла эти ограничения можно уменьшить.

Можно попытаться применить преобразование циклов для создания внутренних распараллеливаемых циклов, тем самым полностью устраняя циклы предшествования. Предположим, что программа имеет два цикла, причем внешний цикл распараллеливаемый, а внутренний — нет. Можно переставить эти два цикла, чтобы сделать распараллеливаемым внутренний цикл и тем самым увеличить количество возможностей для параллелизма на уровне команд. Заметим, что итерации во внутреннем цикле не обязаны быть полностью распараллеливаемыми. Достаточно, чтобы циклы зависимостей были довольно короткими, так чтобы аппаратные ресурсы использовались полностью.

Можно также ослабить ограничения, связанные с использованием ресурсов, корректно балансируя циклы. Предположим, что один цикл использует только слагаемое, а второй — только множитель. Или один цикл ограничен использованием памяти, а второй — вычислительными ресурсами. Такие пары циклов желательно объединить, чтобы одновременно использовать все функциональные единицы.

11.11.3 Векторные и SIMD-команды

Помимо одновременного выполнения нескольких команд, имеются два других важных вида параллелизма на уровне команд: векторные и SIMD-команды. В обоих случаях выполнение одной команды приводит к применению одной и той же операции к вектору данных.

Как уже упоминалось, многие ранние суперкомпьютеры использовали векторные команды. Векторные операции выполняются в стиле конвейера; элементы вектора выбираются последовательно, а вычисления над разными элементами перекрываются. В усовершенствованных векторных машинах операции могут быть *сцепленными* (chained): после получения элементов результирующего вектора они тут же используются операциями другой векторной команды, без ожидания, когда станут доступными все результаты целиком. Кроме того, в машинах со специальным аппаратным обеспечением (scatter/gather) элементы векторов не обязаны быть смежными; для указания местоположения элементов векторов используются индексные векторы.

SIMD-команды указывают, что одна и та же операция выполняется над смежными ячейками памяти. Такие команды параллельно загружают данные из памяти в специальные широкие регистры и проводят вычисления параллельно с использованием специальных аппаратных средств. Многие приложения мультимедиа, графические или цифровой обработки сигналов, с успехом используют такие операции. Недорогие медиапроцессоры могут достичь параллелизма на уровне команд

простым использованием SIMD-команд. Мощные процессоры способны комбинировать применение SIMD-команд с одновременным выполнением нескольких команд для достижения более высокой производительности.

Генерация SIMD-команд и векторных команд имеет много общего с оптимизацией локальности. Когда мы находим независимые части, которые работают со смежными ячейками памяти, мы располосовываем эти итерации и чередуем эти операции во внутренних циклах.

Генерация SIMD-команд имеет две дополнительные трудности. Во-первых, некоторые машины требуют, чтобы SIMD-данные, выбираемые из памяти, были выровнены. Например, может потребоваться, чтобы 256-байтные SIMD-операнды размещались в адресах, значения которых кратны 256. Если исходный цикл работает более чем с одним массивом, одновременное выравнивание всех данных может оказаться невозможным. Во-вторых, данные, используемые последовательными итерациями цикла, могут не быть смежными. Примерами могут служить многие важные алгоритмы цифровой обработки сигналов, такие как декодеры Витерби или быстрое Фурье-преобразование. Для использования SIMD-команд могут потребоваться дополнительные операции по перемещению данных.

11.11.4 Предвыборка

Никакая оптимизация локальности данных не в состоянии устранить все обращения к памяти; например, данные, которые используются программой в первый раз, должны быть выбраны из памяти. Для сокрытия задержки, связанной с операциями с памятью, во многих высокопроизводительных процессорах используются *команды предвыборки* (prefetch instructions). Предвыборка представляет собой машинную команду, которая указывает процессору, что, скорее всего, вскоре будут использованы некоторые данные, так что их желательно загрузить в кэш, если их там еще нет.

Для оценки вероятных промахов кэша можно применить анализ повторных использований из раздела 11.5. При генерации команд предвыборки следует принять во внимание два важных момента. Если выполняется обращение к последовательным ячейкам памяти, то для каждой строки кэша надо выполнить только одну команду предвыборки. Команды предвыборки должны быть выполнены заранее, чтобы данные находились в кэше в тот момент, когда они будут использоваться. Однако команды предвыборки не должны выполняться слишком рано, поскольку при этом в кэше могут оказаться замещенными данные, которые еще будут нужны; при слишком ранней предвыборке могут оказаться удаленными из кэша и данные, которые были загружены командами предвыборки.

Пример 11.73. Рассмотрим следующий код:

```
for (i=0; ii<3; i++)
```

```
for (j=0; j<100; j++)
    A[i,j] = ...;
```

Предположим, что целевая машина имеет команды предвыборки, которые могут одновременно выбирать только два слова, и что задержка при выполнении команды предвыборки составляет время, примерно требуемое для выполнения шести итераций цикла. Код предвыборки для данного примера показан на рис. 11.68.

```
for (i=0; ii<3; i++) {
    for (j=0; j<6; j+=2)
        prefetch(&A[i,j]);
    for (j=0; j<94; j+=2) {
        prefetch(&A[i,j+6]);
        A[i,j] = ...;
        A[i,j+1] = ...;
    }
    for (j=94; j<100; j++)
        A[i,j] = ...;
}
```

Рис. 11.68. Код, модифицированный для предвыборки данных

Мы дважды разворачиваем внутренний цикл с тем, чтобы предвыборка могла быть выполнена для каждой строки кэша. Для предвыборки данных за шесть итераций до их использования мы воспользовались концепцией программной конвейеризации. Пролог загружает данные для первых шести итераций. Устойчивое состояние цикла выполняет предвыборку за шесть команд до их участия в вычислениях. В эпилоге предвыборка не производится, здесь просто выполняются оставшиеся итерации. □

11.12 Резюме к главе 11

- *Параллелизм и локальность при обращении к массивам.* Наиболее важные возможности распараллеливания и оптимизации локальности связаны с обращением к массивам в циклах. Такие циклы обычно имеют ограниченные зависимости между обращениями к элементам массива, а сами обращения выполняются в соответствии с регулярным шаблоном, что при хорошей локальности позволяет эффективно использовать кэш.

- *Аффинные обращения.* Почти вся теория и методы распараллеливания и оптимизации локальности считают, что обращение к массивам аффинное, т.е. выражения для индексов массивов представляют собой линейные функции от индексов циклов.

- *Пространства итераций.* Вложенность циклов с d вложенными циклами определяет d-мерное пространство итераций. Точками пространства являются d-кортежи значений, которые могут принимать индексы циклов в процессе выполнения вложенности циклов. В случае аффинных обращений пределами каждого индекса цикла являются линейные функции от индексов внешних циклов, так что пространство итераций представляет собой многогранник.

- *Исключение Фурье–Моцкина.* Ключевой операцией над пространствами итераций является переупорядочение циклов, определяющих пространство итераций. Для этого требуется, чтобы многогранное пространство итераций было спроецировано на подмножество своих измерений. Алгоритм Фурье–Моцкина заменяет верхний и нижний пределы данной переменной неравенствами с участием границ.

- *Зависимости данных и обращения к массивам.* Центральной проблемой, которую следует решить при распараллеливании и оптимизации локальности, является вопрос о наличии зависимости данных между двумя обращениями к массиву (которые могут обращаться к одному и тому же элементу массива). В случае аффинности обращений и границ циклов задача может быть выражена как выяснение существования решения матрично-векторного уравнения в многограннике, определяющем пространство итераций.

- *Ранг матрицы и повторное использование.* Матрица, описывающая обращение к массиву, может предоставить важную информацию об этом обращении. Если ранг матрицы принимает максимально возможное значение (минимум от количества строк и столбцов), то в процессе итераций повторное обращение не выполняется ни к одному элементу. Если массив хранится построчно (постолбцово), то ранг матрицы с удаленной последней (первой) строкой указывает, обладает ли обращение хорошей локальностью, т.е. выполняется ли обращение к элементам в одной строке кэша примерно в одно и то же время.

- *Зависимости данных и диофантовы уравнения.* То, что два обращения к одному и тому же массиву работают с одной и той же его областью, еще не означает, что они в действительности обращаются к одним и тем же элементам. Дело в том, что обращения могут пропускать некоторые элементы; в качестве примера можно привести ситуацию, когда первое обращение работает с нечетными элементами, а второе — с четными. Чтобы убедиться в наличии зависимости данных между обращениями, следует решить целочисленное диофантово уравнение.

- *Решение диофантовых линейных уравнений.* Ключевым методом является вычисление наибольшего общего делителя коэффициентов при переменных. Целочисленное решение существует только в том случае, когда константный член делится на наибольший общий делитель.

- *Ограничения пространственного разбиения.* Для распараллеливания выполнения вложенности циклов мы должны отобразить итерации циклов на пространство процессоров, которое может иметь одно или несколько измерений. Ограничения пространственного разбиения гласят, что если два обращения в двух разных итерациях зависимы (т.е. обращаются к одному и тому же элементу массива), то они должны отображаться на один и тот же процессор. Если отображение итераций на процессоры аффинное, задачу можно сформулировать в матрично-векторном виде.

- *Примитивные преобразования кода.* Преобразования, использующиеся для распараллеливания программ с аффинными обращениями к массивам, представляют собой комбинации семи примитивов: слияния циклов, расщепления циклов, реиндексирования (добавления константы к индексам циклов), масштабирования (умножения индексов циклов на константу), реверса индекса цикла, перестановки порядка циклов и сдвига (переписывания циклов так, что линии обхода пространства итераций не лежат вдоль одной из осей).

- *Синхронизация параллельных операций.* Иногда больший параллелизм может быть получен путем вставки синхронизирующих операций между шагами программы. Например, последовательные вложенности циклов могут иметь зависимости данных, но синхронизация между циклами может позволить распараллелить эти циклы по отдельности.

- *Конвейеризация.* Этот метод распараллеливания позволяет процессорам совместно использовать данные, синхронно передавая определенные данные (обычно элементы массивов) от одного процессора к соседнему процессору в пространстве процессоров. Данный метод может повысить локальность данных, к которым обращается каждый из процессоров.

- *Ограничения временного разбиения.* При выяснении возможностей конвейеризации требуется найти решение ограничений временного разбиения, которые гласят, что если два обращения к массиву могут работать с одним и тем же элементом массива, то обращение в итерации, которое выполняется первым, должно быть назначено стадии конвейера, которая выполняется не позже стадии, которой назначено второе обращение.

- *Решение ограничений временного разбиения.* Лемма Фаркаша предоставляет метод поиска всех аффинных отображений временнóго разбиения, допустимых для данной вложенности циклов с обращениями к массиву. Этот метод по сути заменяет дуальной основную формулировку линейных неравенств, выражающих ограничения временнóго разбиения.

- *Блокирование.* Этот метод разбивает каждый из нескольких циклов во вложенности на два цикла. Преимущество этого метода состоит в том, что это может позволить нам работать с малыми частями (блоками) многомерного массива, по одному блоку за раз. Это, в свою очередь, повышает локальность программы, позволяя всем необходимым данным размещаться в кэше при выполнении одного блока.

- *Располосование.* Подобно блокированию, этот метод разбивает подмножество циклов вложенности на два. Возможное преимущество заключается в том, что обращения к многомерному массиву выполняются "полосами", что может привести к улучшенному использованию кэша.

11.13 Список литературы к главе 11

За детальным рассмотрением многопроцессорных архитектур мы отсылаем читателя к книге Хеннесси (Hennessy) и Паттерсона (Patterson) [9].

Концепция анализа зависимостей данных разработана Лампортом (Lamport) [13], а также Куком (Kuck), Мураокой (Muraoka) и Ченом (Chen) [6]. Ранние проверки наличия зависимостей данных использовали эвристики для доказательства независимости пар обращений путем определения отсутствия решений диофантовых уравнений и систем действительных линейных неравенств [5, 6, 26]. Майдан (Maydan), Хеннесси (Hennessy) и Лам (Lam) [18] сформулировали тест зависимости данных как задачу целочисленного линейного программирования и показали, что она может быть точно и эффективно решена на практике. Анализ зависимостей данных, описанный здесь, основан на работах Майдана (Maydan), Хеннесси (Hennessy) и Лама (Lam) [18], а также Паха (Pugh) и Воннакотта (Wonnacott) [23], которые, в свою очередь, используют методы исключения Фурье–Моцкина [7] и алгоритм Шостака [25].

К 70-м и началу 80-х годов относятся преобразования циклов для повышения векторизации и распараллеливания: слияния циклов [3], расщепления циклов [1], располосования [17] и обмена циклами [28]. К этому времени относятся три основных экспериментальных проекта распараллеливания/векторизации: Parafrase под руководством Кука (Kuck) в Университете Иллинойса [21], проект PFC под руководством Кеннеди (Kennedy) в Университете Райса [4] и проект PTRAN под руководством Аллена (Allen) из IBM Research [2].

Мак-Келлар (McKellar) и Коффман (Coffman) [19] первыми рассмотрели применение блокирования для повышения локальности данных. Лам (Lam), Ротберт (Rothbert) и Вольф (Wolf) [12] выполнили первый глубокий эмпирический анализ блокирования при использовании кэшей современных архитектур. Вольф (Wolf) и Лам (Lam) [27] воспользовались методами линейной алгебры для вычисления повторного использования данных в циклах. Саркар (Sarkar) и Гао (Gao) [24] открыли оптимизацию сжатия массивов.

Лампорт (Lamport) [13] первым смоделировал циклы как пространства итераций и использовал частный случай аффинного преобразования для поиска параллелизма в многопроцессорных системах. Корни аффинных преобразований произрастают из алгоритма проектирования систолических матриц [11]. Предназначавшиеся для параллельных алгоритмов, непосредственно реализованных в СБИС, систолические массивы требовали минимизации взаимодействия при распараллеливании. Для отображения вычислений на пространственные и временны́е координаты были разработаны специальные алгебраические методы. Концепция аффинного планирования и применение леммы Фаркаша в аффинных преобразованиях были введены Футриером (Feautrier) [8]. Алгоритм аффинного преобразования, описанный в данной книге, основан на работе Лима (Lim) и др. [14–16].

Портерфилд (Porterfield) [22] предложил один из первых алгоритмов компиляции для предвыборки данных. Моури (Mowry), Лам (Lam) и Гупта (Gupta) [20] применили анализ повторного использования для минимизации накладных расходов предвыборки и повышения общей производительности программы.

1. Abu-Sufah, W., D. J. Kuck, and D. H. Lawrie, “On the performance enhancement of paging systems through program analysis and transformations”, *IEEE Trans. on Computing* **C-30**:5 (1981), pp. 341–356.

2. Allen, F. E., M. Burke, P. Charles, R. Cytron, and J. Ferrante, “An overview of the PTRAN analysis system for multiprocessing”, *J. Parallel and Distributed Computing* **5**:5 (1988), pp. 617–640.

3. Allen, F. E. and J. Cocke, “A Catalogue of optimizing transformations”, in *Design and Optimization of Compilers* (R. Rustin, ed.), pp. 1–30, Prentice-Hall, 1972.

4. Allen, R. and K. Kennedy, “Automatic translation of Fortran programs to vector form”, *ACM Transactions on Programming Languages and Systems* **9**:4 (1987), pp. 491–542.

5. Banerjee, U., *Data Dependence in Ordinary Programs*, Master’s thesis, Department of Computer Science, University of Illinois Urbana-Champaign, 1976.

6. Banerjee, U., *Speedup of Ordinary Programs*, Ph.D. thesis, Department of Computer Science, University of Illinois Urbana-Champaign, 1979.

7. Dantzig, G. and B. C. Eaves, "Fourier-Motzkin elimination and its dual", *J. Combinatorial Theory*, **A(14)** (1973), pp. 288–297.

8. Feautrier, P., "Some efficient solutions to the affine scheduling problem: I. One-dimensional time", *International J. Parallel Programming* **21**:5 (1992), pp. 313–348.

9. Hennessy, J. L. and D. A. Patterson, *Computer Architecture: A Quantitative Approach*, Third Edition, Morgan Kaufman, San Francisco, 2003.

10. Kuck, D., Y. Muraoka, and S. Chen, "On the number of operations simultaneously executable in Fortran-like programs and their resulting speedup", *IEEE Transactions on Computers* **C-21**:12 (1972), pp. 1293–1310.

11. Kung, H. T. and C. E. Leiserson, "Systolic arrays (for VLSI)", in Duff, I. S. and G. W. Stewart (eds.), *Sparse Matrix Proceedings*, pp. 256–282. Society for Industrial and Applied Mathematics, 1978.

12. Lam, M. S., E. E. Rothberg, and M. E.Wolf, "The cache performance and optimization of blocked algorithms", *Proc. Sixth International Conference on Architectural Support for Programming Languages and Operating Systems* (1991), pp. 63–74.

13. Lamport, L., "The parallel execution of DO loops", *Comm. ACM* **17**:2 (1974), pp. 83–93.

14. Lim, A. W., G. I. Cheong, and M. S. Lam, "An affine partitioning algorithm to maximize parallelism and minimize communication", *Proc. 13th International Conference on Supercomputing* (1999), pp. 228–237.

15. Lim, A. W. and M. S. Lam, "Maximizing parallelism and minimizing synchronization with affine transforms", *Proc. 24th ACM SIGPLAN-SIG-ACT Symposium on Principles of Programming Languages* (1997), pp. 201–214.

16. Lim, A. W., S.-W. Liao, and M. S. Lam, "Blocking and array contraction across arbitrarily nested loops using affine partitioning", *Proc. ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (2001), pp. 103–112.

17. Loveman. D. B., "Program improvement by source-to-source transformation", *J. ACM* **24**:1 (1977), pp. 121–145.

18. Maydan, D. E., J. L. Hennessy, and M. S. Lam, "An efficient method for exact dependence analysis", *Proc. ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation*, pp. 1–14.

19. McKeller, A. C. and E. G. Coffman, "The organization of matrices and matrix operations in a paged multiprogramming environment", *Comm. ACM*, **12**:3 (1969), pp. 153–165.

20. Mowry, T. C., M. S. Lam, and A. Gupta, "Design and evaluation of a compiler algorithm for prefetching", *Proc. Fifth International Conference on Architectural Support for Programming Languages and Operating Systems* (1992), pp. 62–73.

21. Padua, D. A. and M. J. Wolfe, "Advanced compiler optimizations for supercomputers", *Comm. ACM*, **29**:12 (1986), pp. 1184–1201.

22. Porterfield, A., *Software Methods for Improving Cache Performance on Supercomputer Applications*, Ph.D. Thesis, Department of Computer Science, Rice University, 1989.

23. Pugh, W. and D.Wonnacott, "Eliminating false positives using the omega test", *Proc. ACM SIGPLAN 1992 Conference on Programming Language Design and Implementation*, pp. 140–151.

24. Sarkar, V. and G. Gao, "Optimization of array accesses by collective loop transformations", *Proc. 5th International Conference on Supercomputing* (1991), pp. 194–205.

25. R. Shostak, "Deciding linear inequalities by computing loop residues", *J. ACM*, **28**:4 (1981), pp. 769–779.

26. Towle, R. A., *Control and Data Dependence for Program Transformation*, Ph.D. thesis, Department of Computer Science, University of Illinois Urbana-Champaign, 1976.

27. Wolf, M. E. and M. S. Lam, "A data locality optimizing algorithm", *Proc. SIGPLAN 1991 Conference on Programming Language Design and Implementation*, pp. 30–44.

28. Wolfe, M. J., *Techniques for Improving the Inherent Parallelism in Programs*, Master's thesis, Department of Computer Science, University of Illinois Urbana-Champaign, 1978.

ГЛАВА 12

Межпроцедурный анализ

В этой главе в ходе рассмотрения ряда важных задач оптимизаций, которые не могут быть решены с помощью лишь внутрипроцедурного анализа, будет обоснована важность межпроцедурного анализа. Мы начнем с описания распространенных видов межпроцедурного анализа и сложностей их реализации, после чего перейдем к применениям межпроцедурного анализа. В широко используемых языках программирования наподобие C и Java ключом к любому межпроцедурному анализу является анализ псевдонимов указателей. Поэтому бóльшая часть главы будет посвящена именно технологиям, необходимым для вычисления псевдонимов указателей. Начнем мы с представления Datalog — системы обозначений, позволяющей скрывать сложность эффективного анализа указателей. Затем будет описан алгоритм анализа указателей и показано, как воспользоваться абстракцией диаграмм бинарного выбора (binary decision diagrams — BDD) для эффективной реализации алгоритма.

Большинство оптимизаций, включая описанные в главах 9–11, выполняются одновременно над одной процедурой. Такой анализ называется *внутрипроцедурным* (intraprocedural). Он консервативно предполагает, что вызываемая процедура может изменять состояние всех переменных, видимых процедуре, и что при этом могут осуществляться любые побочные действия, такие как изменение любой видимой процедуре переменной или генерация исключения, приводящая к развертыванию стека вызовов. Внутрипроцедурный анализ относительно прост, но при этом весьма неточен. Для ряда оптимизаций внутрипроцедурного анализа достаточно, в то время как другие оптимизации без межпроцедурного анализа не дают практически ничего.

Межпроцедурный анализ работает со всей программой, следуя за информацией, передаваемой от вызывающей процедуры вызываемой и обратно. Одним относительно простым, но полезным методом является *встраивание* (inlining) процедур, т.е. замена вызова процедуры ее телом с соответствующими изменениями для корректной передачи параметров и возврата значений. Этот метод применим, только если нам известен целевой код вызова процедуры.

Если процедуры вызываются косвенно, посредством указателя или через механизм диспетчеризации методов в объектно-ориентированном программировании, то в ряде случаев определить целевой код вызова позволяет анализ указателей

или ссылок программы. Если такой целевой код единственный, можно применить встраивание. Но даже в этом случае пользоваться встраиванием следует с осторожностью. В общем же случае невозможно, например, встраивание рекурсивных процедур, да и при отсутствии рекурсии встраивание, как минимум, приводит к существенному увеличению размера кода.

12.1 Базовые концепции

В этом разделе будут рассмотрены графы вызовов — графы, которые говорят нам о том, какие процедуры и из каких процедур могут быть вызваны. Мы также рассмотрим идею "контекстной чувствительности", когда анализу потока данных требуется информация о последовательности выполненных вызовов процедур. Иначе говоря, контекстно-чувствительный анализ при рассмотрении "местоположения" в программе наряду с текущей точкой программы включает и последовательность записей активации в стеке.

12.1.1 Графы вызовов

Граф вызовов (call graph) программы представляет собой множество узлов и ребер, такое, что

1. каждой процедуре программы соответствует один узел;

2. каждой *точке вызова* (call site), т.е. месту в программе, где осуществляется вызов процедуры, соответствует один узел графа;

3. если точка вызова c может вызывать процедуру p, в графе имеется ребро от узла c к узлу p.

Многие программы, написанные на таких языках программирования, как C и Fortran, осуществляют вызовы процедур непосредственно, так что целевой код каждого вызова может быть определен статически. В этом случае каждая точка вызова в графе имеет единственное ребро ровно к одной процедуре. Если же программа включает использование процедурных параметров или указателей на функции, то в общем случае до момента выполнения целевой код неизвестен и для разных вызовов может варьироваться. Таким образом, точка вызова может быть связана со многими (или даже всеми) процедурами графа вызовов.

Косвенные вызовы весьма распространены в объектно-ориентированных языках программирования. В частности, при перекрытии методов в подклассе использование метода m может означать любой из большого количества разных методов, зависящих от конкретного подкласса объекта. Использование вызовов таких *виртуальных* методов означает, что мы должны знать тип объекта до того, как сможем определить, какой именно метод будет вызван.

Пример 12.1. На рис. 12.1 показана программа на языке программирования C, в которой объявлен глобальный указатель pf на функцию, которая получает в качестве параметра и возвращает целое число. Имеются две функции данного типа — fun1 и fun2 — и функция main, тип которой не соответствует указателю pf. На рисунке указаны три точки вызова, обозначенные как c1, c2 и c3; эти метки не являются частью программы.

```
                int (*pf)(int);

                int fun1(int x) {
                    if (x < 10)
c1:                     return (*pf)(x+1);
                    else
                        return x;
                }

                int fun2(int y) {
                    pf = &fun1;
c2:                 return (*pf)(y);
                }

                void main() {
                    pf = &fun2;
c3:                 (*pf)(5);
                }
```

Рис. 12.1. Программа с указателем на функцию

Простейший анализ того, на что может указывать pf, состоит в исследовании типов функций. Функции fun1 и fun2 имеют тот же тип, что и указатель pf, в то время как функция main имеет другой тип. Таким образом, консервативный граф вызовов показан на рис. 12.2, *а*. При более аккуратном анализе программы обнаруживается, что указатель pf в функции main становится равным fun2, а затем в функции fun2 ему присваивается значение fun1. Никаких иных присваиваний указателю pf в программе нет, так что, в частности, указатель pf не может указывать на функцию main. В результате получается тот же граф вызовов, приведенный на рис. 12.2, *а*.

После еще более точного анализа можно сказать, что точка c3 — единственное место, где pf указывает на fun2, поскольку этому вызову непосредственно предшествует присваивание соответствующего значения указателю pf. Аналогично в точке c2 единственное значение, которое может принимать указатель pf, — это fun1. Поскольку единственный способ попасть в функцию fun1 — вызов

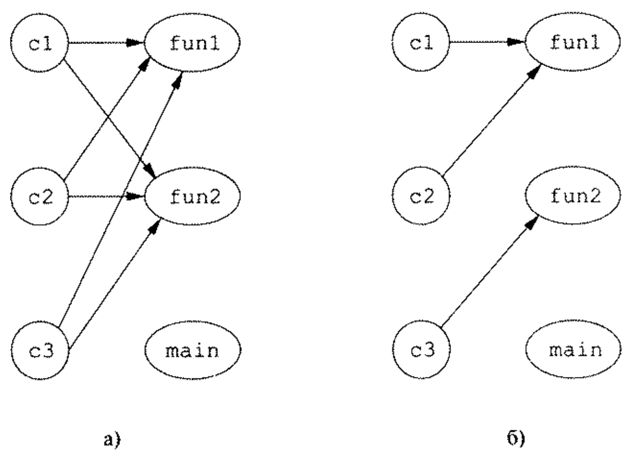


Рис. 12.2. Графы вызовов для кода на рис. 12.1

из функции fun2, а в самой функции fun1 значение pf не изменяется, в этой функции глобальный указатель pf всегда указывает на fun1. В частности, в точке c1 мы можем быть уверены в том, что pf указывает на fun1. Таким образом, на рис. 12.2, *б* приведен более точный, корректный граф вызовов. □

В общем случае наличие ссылок или указателей на функции или методы требует от нас статической аппроксимации потенциальных значений всех параметров процедур, указателей на функции и типов объектов, получающих сообщения. Для выполнения точной аппроксимации требуется применение межпроцедурного анализа. Этот анализ итеративен и начинается со статически определяемого целевого кода. При обнаружении нового целевого кода анализ добавляет в граф вызовов новые ребра и повторяет выявление нового целевого кода, пока данный процесс не сойдется.

12.1.2 Чувствительность к контексту

Межпроцедурный анализ очень сложен еще и потому, что поведение каждой процедуры зависит от контекста, в котором она вызвана. В примере 12.2 для иллюстрации важности чувствительности к контексту рассматривается задача межпроцедурного распространения констант в небольшой программе.

Пример 12.2. Рассмотрим фрагмент программы на рис. 12.3. Функция *f* вызывается в трех точках: c1, c2 и c3. На каждой итерации в точке c1 как фактический параметр передается константа 0, а в точках c2 и c3 — константа 243; возвращаются соответственно константы 1 и 244. Таким образом, функция *f* в каждом

контексте вызывается с передачей ей константного значения, но это значение зависит от контекста.

```
              for (i = 0; i < n; i++) {
c1:                t1 = f(0);
c2:                t2 = f(243);
c3:                t3 = f(243);
                   X[i] = t1+t2+t3;
              }

              int f (int v) {
                   return (v+1);
              }
```

Рис. 12.3. Фрагмент программы, иллюстрирующий необходимость контекстно-чувствительного анализа

Как мы увидим, сказать, что переменным t1, t2 и t3 (а значит, и $X[i]$) присваиваются константные значения, невозможно до тех пор, пока не будет выяснено, что при вызове в контексте c1 функция f возвращает значение 1, а в двух остальных контекстах – значение 244. После простейшего наивного анализа можно заключить, что f может возвращать в любом вызове либо 1, либо 244. □

Один простейший, но очень неточный подход к межпроцедурному анализу, известный как *контекстно-нечувствительный анализ* (context-insensitive analysis), заключается в рассмотрении каждой инструкции вызова и возврата как операции безусловного перехода. Мы создаем надграф потока управления, в котором, помимо обычных ребер внутрипроцедурных потоков управления, имеются дополнительные ребра, соединяющие

1. каждую точку вызова с началом вызываемой в ней процедуры;

2. инструкции возврата с точками вызова.[1]

Добавляются инструкции присваивания каждого фактического параметра соответствующему формальному параметру, а также присваивания возвращаемого значения переменной, получающей результат. Затем можно применить стандартный анализ, предназначающийся для использования в процедуре, к надграфу потока управления для поиска контекстно-нечувствительных межпроцедурных результатов. При своей простоте такая модель абстрагируется от важных взаимоотношений между входными и выходными значениями в вызовах процедур, что приводит к неточности такого анализа.

[1] Строго говоря, возврат выполняется в точку, непосредственно следующую за точкой вызова.

Пример 12.3. На рис. 12.4 показан надграф потока управления для программы на рис. 12.3. Блок B_6 представляет собой функцию f. Блок B_3 содержит точку вызова c1; в нем формальному параметру v присваивается значение 0 и выполняется переход в начало функции f, в блок B_6. Аналогично блоки B_4 и B_5 представляют точки вызовов c2 и c3 соответственно. В блоке B_4, который достигается из конца функции f (блок B_6), возвращаемое функцией значение присваивается переменной t1. Затем формальный параметр v устанавливается равным 243, и путем перехода к блоку B_6 выполняется новый вызов функции f. Обратите внимание на отсутствие ребер от блока B_3 к блоку B_4. Управление переходит от блока B_3 к блоку B_4 через функцию f.

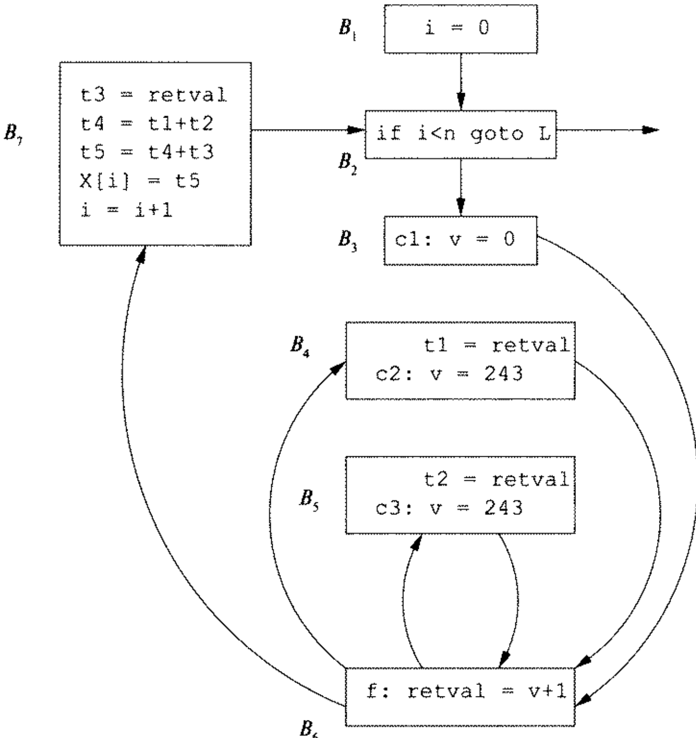


Рис. 12.4. Граф потока управления для кода на рис. 12.3, получающийся при рассмотрении вызовов функций в качестве потока управления

Блок B_5 подобен блоку B_4. Он получает возвращаемое значение от функции f, присваивает его переменной t2 и выполняет третий вызов функции f. Блок B_7 представляет возврат из третьего вызова функции f и присваивание значения элементу массива $X[i]$.

Если рассматривать рис. 12.4 как если бы это был граф потока единой процедуры, то можно заключить, что при переходе к блоку B_6 переменная v может иметь значение 0 или 243. Таким образом, наибольшее, что можно сказать о значении `retval`, — что оно может быть равным либо 1, либо 244. Значит, $X[i]$ должно принимать одно из следующих значений: 3, 246, 489 или 732. В отличие от контекстно-нечувствительного анализа, контекстно-чувствительный анализ разделяет результаты вызова в каждом контексте и дает интуитивный ответ, описанный в примере 12.2: переменная `t1` всегда равна 1, переменные `t2` и `t3` всегда равны 244, а $X[i]$ — 489. □

12.1.3 Строки вызовов

В примере 12.2 контексты отличались точками вызова процедуры f. В общем же случае контекст вызова определяется содержимым всего стека вызовов. Будем говорить о строке точек вызова в стеке как о *строке вызовов* (call string).

Пример 12.4. На рис. 12.5 приведен немного измененный код, показанный на рис. 12.3. Здесь вызовы функции f заменены вызовами функции g, которая вызывает f с тем же аргументом. Так у нас появляется еще одна точка вызова `c4`, в которой функция g вызывает функцию f.

Имеется три строки вызовов функции f: $(c1, c4)$, $(c2, c4)$ и $(c3, c4)$. Как видите, в данном примере значение v в функции f зависит не от последней точки вызова `c4` в строке вызовов; оно определяется первым элементом этой строки. □

```
            for (i = 0; i < n; i++) {
c1:             t1 = g(0);
c2:             t2 = g(243);
c3:             t3 = g(243);
                X[i] = t1+t2+t3;
            }

            int g (int v) {
c4:             return f(v);
            }

            int f (int v) {
                return (v+1);
            }
```

Рис. 12.5. Фрагмент программы, иллюстрирующий строки вызовов

В примере 12.4 демонстрируется, что важная для анализа информация может быть внесена ранними элементами цепочки вызовов. Фактически иногда для

получения максимально точного ответа необходимо рассматривать всю строку вызовов, как показано в примере 12.5.

Пример 12.5. В этом примере иллюстрируется, как возможность неограниченного рассмотрения строк вызовов позволяет получать более точные результаты. На рис. 12.6 мы видим, что если g вызывается с положительным значением c, то g рекурсивно вызывается c раз. Каждый раз при вызове функции g значение ее параметра v уменьшается на 1. Таким образом, значение параметра v функции g в контексте строки вызова $c2\,(c4)^n$ равно $243 - n$. Результатом работы функции g, таким образом, является увеличение нуля или любого отрицательного аргумента на 1 и возврат 2 для любого аргумента, равного 1 или больше.

```
            for (i = 0; i < n; i++) {
c1:             t1 = g(0);
c2:             t2 = g(243);
c3:             t3 = g(243);
                X[i] =  t1+t2+t3;
            }

            int g (int v) {
                if (v > 1) {
c4:                 return g(v-1);
                } else {
c5:                 return f(v);
            }

            int f (int v) {
                return (v+1);
            }
```

Рис. 12.6. Рекурсивная программа, требующая анализа всей строки вызовов

Для функции f имеются три возможные строки вызовов. Если работа начинается с вызова в точке c1, то функция g немедленно вызывает функцию f, так что одной из таких строк является строка $(c1, c5)$. Если начать с точки c2 или c3, то функция g вызывается 243 раза, после чего вызывается функция f. Соответствующие строки вызова — $(c2, c4, c4, \ldots, c5)$ и $(c3, c4, c4, \ldots, c5)$, в каждой из которых имеется по 242 точки c4. В первом из этих контекстов значение параметра v функции f равно 0, в то время как в остальных двух контекстах это значение — 1. □

При проектировании контекстно-чувствительного анализа главным фактором должна являться его точность. Например, вместо рассмотрения полных строк вы-

зовов для определения контекста мы можем ограничиться только k последними точками вызовов. Такой метод известен как k-ограниченный контекстный анализ. Контекстно-нечувствительный анализ представляет собой частный случай k-ограниченного контекстно-чувствительного анализа при $k = 0$. Все константы в примере 12.2 могут быть найдены с использованием 1-ограниченного анализа, а в примере 12.4 — при помощи 2-ограниченного анализа. Однако в примере 12.5 никакой k-ограниченный анализ не в состоянии найти все константы, если константы 243 будут заменены двумя разными произвольно большими константами.

Вместо выбора фиксированного значения k для всех *ациклических* строк вызовов (т.е. строк, в которых отсутствуют рекурсивные циклы) следует добиваться полной контекстной чувствительности. Чтобы ограничить количество различных анализируемых контекстов, в случае строк вызовов с рекурсией можно свернуть все рекурсивные циклы. В примере 12.5 вызовы из точки c2 можно аппроксимировать строкой вызовов $(c2, c4^*, c5)$. Заметим, что при такой схеме даже для программ без рекурсии количество разных контекстов вызова может экспоненциально зависеть от количества процедур в программе.

12.1.4 Контекстно-чувствительный анализ на основе клонирования

Еще один подход к контекстно-чувствительному анализу заключается в мысленном клонировании процедур, по одной для каждого уникального контекста, интересующего нас. Затем к клонированному графу вызовов можно применить контекстно-нечувствительный анализ. В примерах 12.6 и 12.7 показаны клонированные версии кода из примеров 12.4 и 12.5 соответственно. В реальности клонировать код не требуется — можно просто использовать эффективное внутреннее представление для отслеживания результатов анализа каждого клона.

Пример 12.6. Клонированная версия кода, представленного на рис. 12.5, показана на рис. 12.7. Поскольку каждый контекст вызова работает со своим клоном, никакой путаницы не возникает. Например, g1 получает в качестве аргумента 0 и возвращает 1; g2 и g3 получают в качестве аргументов 243 и возвращают 244. □

Пример 12.7. Клонированная версия кода из примера 12.5 показана на рис. 12.7. Для процедуры g мы создаем клон для представления всех экземпляров g, впервые вызываемых из точек вызова c1, c2 и c3. В этом случае анализ может определить, что вызов в точке c1 возвращает 1 (в предположении, что он в состоянии вывести, что при $v = 0$ результат проверки $v > 1$ отрицательный). Однако этот анализ недостаточно хорош, чтобы получить константы для точек вызовов c2 и c3. □

```
            for (i = 0; i < n; i++) {
c1:               t1 = g1(0);
c2:               t2 = g2(243);
c3:               t3 = g3(243);
                  X[i] =  t1+t2+t3;
            }
            int g1 (int v) {
c4.1:              return f1(v);
            }
            int g2 (int v) {
c4.2:              return f2(v);
            }
            int g3 (int v) {
c4.3:              return f3(v);
            }

            int f1 (int v) {
                return (v+1);
            }
            int f2 (int v) {
                return (v+1);
            }
            int f3 (int v) {
                return (v+1);
            }
```

Рис. 12.7. Клонированная версия кода, представленного на рис. 12.5

12.1.5 Контекстно-чувствительный анализ на основе резюме

Межпроцедурный анализ на основе резюме представляет собой расширение анализа на основе областей. По сути, в анализе на основе областей каждая процедура представлена кратким описанием ("резюме"), которое инкапсулирует некоторое наблюдаемое поведение процедуры. Основная цель резюме — избежать повторного анализа тела процедуры в каждой точке, где эта процедура может быть вызвана.

Рассмотрим сначала случай с отсутствием рекурсии. Каждая процедура моделируется как область с единой точкой входа, при этом все пары "вызывающая–вызываемая процедуры" описываются отношением "внешняя–внутренняя области". Единственное отличие от внутрипроцедурной версии заключается в том,

```
        for (i = 0; i < n; i++) {
c1:         t1 = g1(0);
c2:         t2 = g2(243);
c3:         t3 = g3(243);
            X[i] =  t1+t2+t3;
        }

        int g1 (int v) {
            if (v > 1) {
c4.1:           return g1(v-1);
            } else {
c5.1:           return f1(v);
        }

        int g2 (int v) {
            if (v > 1) {
c4.2:           return g2(v-1);
            } else {
c5.2:           return f2(v);
        }

        int g3 (int v) {
            if (v > 1) {
c4.3:           return g3(v-1);
            } else {
c5.3:           return f3(v);
        }

        int f1 (int v) {
            return (v+1);
        }
        int f2 (int v) {
            return (v+1);
        }
        int f3 (int v) {
            return (v+1);
        }
```

Рис. 12.8. Клонированная версия кода, представленного на рис. 12.6

что в межпроцедурном случае область процедуры может быть вложена внутрь нескольких различных внешних областей.

Анализ состоит из двух частей:

1. восходящая фаза, вычисляющая передаточную функцию для резюмирования действий процедуры;
2. нисходящая фаза, которая передает информацию вызывающей функции для вычисления результатов выполнения вызываемых процедур.

Для получения полностью контекстно-чувствительных результатов информация от разных контекстов вызова должна распространяться вниз к отдельным вызываемым функциям. Для повышения эффективности (с понижением точности) информация от всех вызывающих функций может объединяться с использованием оператора сбора, а затем передаваться вниз вызываемым функциям.

Пример 12.8. Для распространения констант каждая процедура резюмируется передаточной функцией, определяющей, как константы распространяются через тело этой процедуры. В примере 12.2 можно резюмировать f как функцию, которая для данной константы c в качестве фактического параметра v возвращает константу $c + 1$. На основе этой информации анализ может определить, что переменные `t1`, `t2` и `t3` получают значения 1, 244 и 244 соответственно. Заметим, что этот анализ свободен от неточности из-за нереализуемых строк вызовов.

Вспомним, что пример 12.4 расширяет пример 12.2 путем добавления функции g, которая вызывает функцию f. Таким образом, можно заключить, что передаточная функция для g такая же, как и передаточная функция для f. Так что мы вновь делаем вывод о том, что переменные `t1`, `t2` и `t3` получают значения 1, 244 и 244 соответственно.

Теперь рассмотрим значение параметра v функции f в примере 12.2. В качестве первого действия можно объединить результаты всех контекстов вызова. Поскольку v может иметь значение 0 или 243, можно просто заключить, что v не является константой. Это верный вывод, поскольку нет константы, которой можно бы было заменить v в коде.

Если мы хотим достичь более точных результатов, то можем вычислить их для интересующих нас контекстов. Для получения контекстно-чувствительного ответа информация должна передаваться вниз от рассматриваемого контекста. Этот шаг аналогичен нисходящему проходу в анализе на основе областей. Например, значение v равно 0 в точке вызова `c1` и 243 — в точках вызова `c2` и `c3`. Для получения преимуществ распространения констант внутри f следует зафиксировать это различие путем создания двух специализированных клонов: одного — для входного значения 0, а второго — для значения 243, как показано на рис. 12.9. □

В конце примера 12.8 мы видим, что если мы хотим скомпилировать код в разных контекстах по-разному, то мы должны клонировать его. Отличие заключается

```
          for (i = 0; i < n; i++) {
c1:             t1 = f0(0);
c2:             t2 = f243(243);
c3:             t3 = f243(243);
                X[i] = t1+t2+t3;
          }

          int f0 (int v) {
              return (1);
          }

          int f243 (int v) {
              return (244);
          }
```

Рис. 12.9. Результат распространения всех возможных константных аргументов в функции f

в том, что при подходе на основе клонирования оно выполняется до анализа, на основе строк вызовов. При использовании подхода на основе резюме клонирование выполняется после анализа, на основе его результатов. Даже если клонирование не применяется, выводы подхода на основе резюме о результатах выполнения вызываемой процедуры делаются точно, без проблемы нереализуемых путей.

Вместо клонирования функций можно прибегнуть к встраиванию кода. Встраивание обладает дополнительным эффектом устранения накладных расходов, связанных с вызовами процедур.

Справиться с рекурсией можно путем вычисления решения с фиксированной точкой. При наличии рекурсии мы сперва находим сильно связанные компоненты графа вызовов. В восходящей фазе мы не посещаем сильно связанные компоненты до тех пор, пока не будут посещены все их преемники. В случае нетривиального сильно связанного компонента мы итеративно вычисляем передаточную функцию для каждой процедуры компонента до достижения сходимости, т.е. итеративно обновляем передаточные функции до тех пор, пока не достигнем итерации с отсутствием изменений.

12.1.6 Упражнения к разделу 12.1

Упражнение 12.1.1. На рис. 12.10 приведена программа на языке программирования C с двумя указателями на функции — p и q. N — константа, которая может быть меньше или больше 10. Заметим, что в результате в программе получится бесконечная последовательность вызовов, но в данном случае нас это волновать не должно.

а) Укажите все точки вызова программы.

б) Укажите для каждой точки вызова, на что указывает p? На что указывает q?

в) Изобразите граф вызовов для данной программы.

! г) Опишите все строки вызовов для функций f и g.

```
int (*p)(int);
int (*q)(int);

int f(int i) {
    if (i < 10)
        {p = &g; return (*q)(i);}
    else
        {p = &f; return (*p)(i);}
}

int g(int j) {
    if (j < 10)
        {q = &f; return (*p)(j);}
    else
        {q = &g; return (*q)(j);}
}

void main() {
    p = &f;
    q = &g;
    (*p)((*q)(N));
}
```

Рис. 12.10. Программа к упражнению 12.1.1

Упражнение 12.1.2. На рис. 12.11 приведена "тождественная функция" `id`, возвращающая переданный ей аргумент. Приведен также фрагмент кода, включающий ветвление с последующим суммированием $x + y$.

а) Изучите код. Что вы можете сказать о значении z после выполнения данного кода?

б) Постройте граф потока для данного фрагмента кода, рассматривая вызовы функции `id` как поток управления.

в) Если применить к графу, построенному в предыдущем задании, анализ распространения констант, как в разделе 9.4, то какие константные значения будут найдены?

г) Перечислите все точки вызова на рис. 12.11.

д) Перечислите все контексты вызовов функции `id`.

е) Перепишите код на рис. 12.11 путем клонирования новой версии `id` для каждого контекста ее вызова.

ж) Постройте граф потока вашего кода из предыдущего пункта, рассматривая вызовы функции `id` как поток управления.

з) Примените анализ распространения констант к полученному в предыдущем пункте графу потока. Какие константные значения найдены в этот раз?

```
int id(int x) { return x;}

    ...
if (a == 1) { x = id(2); y = id(3); }
else        { x = id(3); y = id(2); }
z = x+y;
    ...
```

Рис. 12.11. Фрагмент кода к упражнению 12.1.2

12.2 Необходимость межпроцедурного анализа

Получив представление о том, насколько сложен межпроцедурный анализ, рассмотрим важную проблему — когда и зачем он может потребоваться. Хотя в качестве иллюстрации межпроцедурного анализа мы уже рассматривали распространение констант, он как не был легко применимым, так и не показал особых преимуществ при его выполнении. Большинство выгод распространения констант можно получить простым применением внутрипроцедурного анализа и встраиванием вызовов процедур в наиболее часто выполняемых разделах кода.

Однако существует множество причин, по которым межпроцедурный анализ оказывается очень существенным. Ниже будет описано несколько важных его применений.

12.2.1 Вызовы виртуальных методов

Как уже упоминалось, объектно-ориентированные программы включают множество небольших методов. Если оптимизировать по одному методу за раз, то у нас будет слишком мало возможностей получить реальную выгоду от оптимизации. Языки программирования типа Java динамически загружают свои классы. В результате во время компиляции мы не знаем, какой из множества методов с именем m будет реально вызван при использовании в коде вызова $x.m()$.

Многие реализации Java используют оперативную компиляцию (just-in-time compiler) для компиляции байт-кода в процессе выполнения. Распространенной оптимизацией является профилирование выполнения и определение основных типов объектов — получателей сообщений. После этого можно встроить в код наиболее часто использующиеся методы. Соответствующий код включает динамическую проверку типа и выполняет встроенные методы, если объекты времени выполнения имеют ожидаемый тип.

Еще один подход к разрешению использований метода с именем m возможен при доступности в процессе компиляции всего исходного кода. Тогда для определения типов объектов оказывается возможным выполнение межпроцедурного анализа. Если оказывается, что возможный тип объекта x единственный, то вызов $x.m()$ разрешим во время компиляции. Нам точно известно, какой именно метод m имеется в виду в данном контексте. В таком случае можно встроить код данного метода m без включения в код проверки типа x.

12.2.2 Анализ псевдонимов указателей

Даже если мы не хотим выполнять межпроцедурные версии распространенных анализов потоков данных наподобие достигающих определений, такие анализы все равно только выигрывают от применения межпроцедурного анализа указателей. Все представленные в главе 9 анализы применимы только к локальным скалярным переменным, которые не могут иметь псевдонимов. Однако применение указателей широко распространено, в особенности в С-образных языках программирования. Знание того, что у указателя может быть псевдоним (alias), может повысить точность методов, описанных в главе 9.

Пример 12.9. Рассмотрим последовательность из трех инструкций, которая может образовывать базовый блок:

```
*p = 1;
*q = 2;
 x = *p;
```

Без знания того, могут ли p и q указывать на одно и то же место в памяти, мы не в состоянии сказать, имеет ли **x** значение 1 при выходе из базового блока. □

12.2.3 Параллельность

Как говорилось в главе 11, наиболее эффективный способ распараллеливания приложения заключается в поиске наиболее крупнозернистой параллельности, такой как параллельность во внешних циклах программы. В этом случае межпроцедурный анализ приобретает особую важность. *Скалярная* оптимизация (описанная в главе 9 и основанная на значениях простых переменных) существенно отличается от распараллеливания. В случае распараллеливания даже одна ложная зависимость данных может сделать нераспараллеливаемым весь цикл, существенно снизив тем самым эффективность оптимизации. В случае скалярной оптимизации такое усиление неточностей не наблюдается. При скалярной оптимизации нам достаточно найти основные возможности для оптимизации, и одна или две упущенные возможности особой роли не играют.

12.2.4 Поиск программных ошибок и уязвимых мест

Межпроцедурный анализ важен не только при оптимизации кода. Те же методы могут использоваться и для анализа имеющегося программного обеспечения на наличие различных ошибок кодирования. Эти ошибки могут сделать программы ненадежными; ряд ошибок могут сделать программы уязвимыми для хакерских атак, позволяя хакерам взять управление вычислительной системой в свои руки.

Статический анализ полезен при обнаружении многих распространенных ошибок. Например, элемент данных должен быть защищен блокировкой. В качестве другого примера можно указать, что за запретом прерываний в операционной системе должно следовать их разрешение. Поскольку существенным источником ошибок являются несогласованности, пересекающие границы процедур, межпроцедурный анализ приобретает особую важность. Примерами инструментария, на практике эффективно использующего межпроцедурный анализ для поиска ошибок в больших программах, являются системы PREfix и Metal. Эти системы статически находят ошибки в программах и могут существенно повысить надежность последних. Однако обе указанные системы неполны и недостаточно надежны в том смысле, что они могут находить не все ошибки и не все их предупреждения соответствуют реальным ошибкам в программах. К сожалению, использованный межпроцедурный анализ достаточно неточен, так что большое количество ложных предупреждений делает упомянутые системы непригодными для практического использования. Тем не менее, хотя системы и далеки от совершенства, их систематическое использование может повысить надежность программного обеспечения.

Переходя к вопросу уязвимости программ, крайне желательно найти в программах все потенциальные ошибки. В 2006 году двумя "наиболее популярными" причинами вторжения, использовавшимися хакерами для взлома систем, были следующие.

1. Недостаточная проверка в Web-приложениях. Одним из наиболее популярных видов использования такого рода уязвимых мест является SQL-ввод, когда хакеры получают в свои руки управление базой данных при помощи манипулирования входными данными, передаваемыми Web-приложениям.

2. Переполнение буфера в программах на языках программирования C и C++. Поскольку языки программирования C и C++ не выполняют проверки выхода за границы массивов, хакеры могут записать специально подобранные строки в не предназначенные для этого области и получить контроль над выполнением программы.

В следующем разделе мы рассмотрим применение межпроцедурного анализа для защиты программ от таких вторжений.

12.2.5 SQL-ввод

Под данным названием подразумевается уязвимость, заключающаяся в подборе хакером пользовательского ввода Web-приложения и получение непредусмотренного доступа к базе данных. Например, банки предоставляют пользователям возможности выполнения транзакций в оперативном режиме при условии ввода корректного пароля. Обычно в такой системе пользователь вводит строки в Web-форме, после чего эти строки используются как часть запроса к базе данных в языке SQL. Если разработчики системы не были аккуратны и осторожны, то строки, вводимые пользователем, могут изменить смысл SQL-инструкции совершенно неожиданным способом.

Пример 12.10. Предположим, что банк предоставляет своим клиентам доступ к отношению

```
AcctData(name, password, balance)
```

Данное отношение представляет собой таблицу троек, каждая из которых состоит из имени клиента, пароля и баланса счета. Предполагается, что клиент может увидеть свой счет, только если он корректно введет свое имя и пароль. Если хакер сумеет увидеть эту информацию, это не самое худшее, что может произойти, но данный простой пример — типичная часть более сложных ситуаций, когда хакеры осуществляют платежи с чужих счетов.

Система может реализовывать запросы о состоянии счета следующим образом.

1. Пользователи получают Web-форму, в которой вводят свое имя и пароль.

2. Имя копируется в переменную n, а пароль — в переменную p.

3. Позже, вероятно, в некоторой иной процедуре выполняется следующая SQL-инструкция:

```
SELECT balance FROM AcctData
WHERE name = ':n' and password = ':p'
```

Для читателей, не знакомых с SQL, переведем этот запрос на обычный язык: "Найти в таблице `AcctData` строку, первый компонент которой (имя) равен строке, находящейся в настоящий момент в переменной n, а второй компонент которой (пароль) равен строке, находящейся в настоящий момент в переменной p; вывести третий компонент (баланс) этой строки". Обратите внимание, что SQL использует для строк одинарные, а не двойные кавычки, а двоеточия перед n и p указывают, что это переменные языка программирования, в программе на котором использован данный SQL-запрос.

Предположим, что хакер, желающий получить сведения о счете Чарльза Диккенса, передает переменным n и p следующие строки:

$n =$ `Charles Dickens' --`, $p =$ `who cares`.

Эти странные строки превращают запрос в

```
SELECT balance FROM AcctData
WHERE name = 'Charles Dickens' --' and password = 'who cares'
```

Во многих системах баз данных последовательность "--" представляет собой начало комментария и делает все, следующее за ней, комментарием. В результате получившийся запрос теперь просит вывести балансы всех клиентов по имени `'Charles Dickens'`, независимо от используемого пароля. Вот как выглядит запрос после удаления комментария:

```
SELECT balance FROM AcctData
WHERE name = 'Charles Dickens'
```

□

В примере 12.10 "плохие" строки располагаются в двух переменных, которые могут быть переданы между процедурами. Однако в более реалистических случаях такие строки могут быть много раз скопированы или объединены с другими для образования полного запроса. Мы не можем надеяться на обнаружение ошибок, приводящих к уязвимости посредством SQL-ввода, без полного межпроцедурного анализа всей программы в целом.

12.2.6 Переполнение буфера

Атака путем переполнения буфера осуществляется путем передачи пользователем специальным образом созданных данных за пределами предназначенного для них буфера и получения посредством этого доступа к управлению программой. Например, программа на языке программирования C может считывать предоставляемую пользователем строку s и копировать ее в буфер b с использованием вызова функции `strcpy(b,s);`. Если строка s длиннее буфера b, то будут изменены значения ячеек памяти, не являющиеся частью буфера b. Это может привести к некорректной работе программы или как минимум к получению неверных результатов, поскольку изменяются используемые программой данные.

Что еще хуже — хакер может выбрать строку s так, что она приведет к более серьезным последствиям, чем просто ошибка. Например, если буфер находится в стеке времени выполнения, то он располагается вблизи адреса возврата из функции. Специально подобранное значение s может переписать адрес возврата и при выходе из функции управление будет передано в место, выбранное хакером. Если хакер хорошо знает операционную систему и аппаратное обеспечение, то он сможет заставить программу выполнить команду, которая передаст ему управление над машиной. В некоторых случаях может даже иметься возможность так подменить адрес возврата, что управление будет передано коду, являющемуся частью строки s, позволяя, таким образом, вставить в выполняемый код любую хакерскую подпрограмму.

Для предотвращения переполнения буфера для любой операции, которая выполняет запись в массив, должно быть выполнено статическое доказательство невозможности выхода за границы массива либо соответствующая проверка должна выполняться динамически. Поскольку такая проверка должна вноситься в программы на языке программирования C или C++ вручную, очень легко либо забыть ее внести, либо ошибиться при ее написании. В настоящее время разработаны эвристические инструменты, которые позволяют выполнить проверку наличия как минимум некоторых тестов перед вызовом функции `strcpy`.

Избежать динамических проверок невозможно, поскольку размер пользовательского ввода не может быть определен статически. Все, чего можно добиться с помощью статического анализа, — убедиться в наличии динамических проверок. Таким образом, разумная стратегия состоит в том, чтобы заставить компилятор вставлять динамические проверки границ массивов при каждой записи и использовать статический анализ как средство для оптимизации и устранения по возможности как можно большего количества проверок. При этом нет необходимости отлавливать каждое потенциальное нарушение; более того, можно оптимизировать только часто выполняемые области кода.

Вставка проверки выхода за границы массива в программу на языке программирования C — задача нетривиальная, даже если забыть о стоимости таких

проверок. Указатель может указывать внутрь некоторого массива, причем мы можем не знать размер этого массива. В настоящее время разработаны методы динамического отслеживания размеров буферов, на которые указывает каждый из указателей. Такая информация позволяет компилятору вставить проверки выхода за границы массива для всех обращений. Достаточно интересно, что останавливать программу при обнаружении выхода за границу массива не рекомендуется. В действительности переполнение буферов встречается на практике, так что, если запретить все переполнения, программа может просто перестать работать. Решение заключается в динамическом изменении размера массива для предотвращения переполнения буфера.

Для снижения стоимости динамических проверок выхода за границы массива можно использовать межпроцедурный анализ. Предположим, например, что нас интересует только отлавливание переполнения буфера строками пользовательского ввода. Тогда можно использовать статический анализ для определения того, какие переменные могут хранить содержимое, предоставленное пользователем. Как и в случае SQL-ввода, отслеживание копирований входных данных между процедурами позволяет устранить излишние проверки выхода за границы массива.

12.3 Логическое представление потока данных

До этого момента наше представление задач потоков данных и их решений можно было охарактеризовать как представление с использованием теории множеств. Мы представляли информацию в виде массивов и вычисляли интересующие нас результаты с использованием операторов наподобие объединения и пересечения массивов. Например, в задаче достигающих определений в разделе 9.2.4 мы вычисляем множества IN $[B]$ и OUT $[B]$ для базового блока B и описываем их как множества определений. Содержимое блока B мы представляем множествами *gen* и *kill*.

Чтобы справиться со сложностями межпроцедурного анализа, требуется более обобщенное и лаконичное представление, основанное на логике. Вместо того чтобы говорить нечто наподобие “определение D входит в IN $[B]$”, для обозначения этого мы используем запись типа $in(B, D)$. Это позволяет нам записать краткие “правила” вывода логических заключений о программе. Кроме того, такие правила можно эффективно реализовать, как обобщение подхода с битовыми векторами для операций над множествами. Наконец, логический подход позволяет комбинировать несколько анализов в один интегрированный алгоритм. Например, в разделе 9.5 мы описали устранение частичной избыточности как последовательность из четырех анализов потоков данных и двух дополнительных промежуточных ша-

гов. В случае логического представления все эти шаги могут быть объединены в один набор логических правил, которые решаются одновременно.

12.3.1 Введение в Datalog

Datalog — это язык, использующий запись наподобие используемой в языке программирования Prolog, но с более простой по сравнению с Prolog семантикой. Начнем с того, что элементами Datalog являются *атомы* вида $p(X_1, X_2, \ldots, X_n)$. Здесь

1. p — *предикат*, символ, который представляет тип инструкции наподобие "определение достигает начала блока";

2. $X_1, X_2, \ldots, X_n$ — члены, такие как переменные или константы; в качестве аргументов предиката могут выступать простые выражения.[2]

Основной атом (ground atom) представляет собой предикат, аргументами которого являются только константы. Каждый основной факт является утверждением о некотором конкретном факте, и его значением является либо истина, либо ложь. Часто оказывается удобным представлять предикат *отношением* или таблицей его истинных основных атомов. Каждый основной атом представлен одной строкой, или *кортежем* (tuple), отношения. Столбцы отношения называются *атрибутами*, и каждый кортеж имеет компонент для каждого атрибута. Атрибуты соответствуют компонентам основных атомов, представленных отношением. Любой основной атом в отношении истинен, а основные атомы, не входящие в отношение, ложны.

Пример 12.11. Предположим, что предикат $in(B, D)$ означает "определение D достигает начала базового блока B". Мы можем предположить, что для некоторого графа потока истинными являются предикаты $in(b_1, d_1)$, $in(b_2, d_1)$ и $in(b_2, d_2)$. Предположим также, что все остальные in для данного графа потока ложны. Тогда отношение на рис. 12.12 представляет значение данного предиката для рассматриваемого графа потока.

Атрибутами отношения являются B и D, а тремя кортежами отношения — (b_1, d_1), (b_2, d_1) и (b_2, d_2). □

Мы также иногда будем встречаться с атомами, которые представляют собой сравнения переменных и констант. Примерами таких атомов могут служить $X \neq Y$ и $X = 10$. В этих примерах предикаты представляют собой операторы сравнения, т.е. мы можем рассматривать сравнение $X = 10$ как предикат вида

[2]Формально такие члены построены из символов функций и существенно усложняют Datalog. Однако мы будем использовать лишь несколько операторов, таких как сложение и вычитание констант, в контексте, который не усложнит наше рассмотрение.

B	D
b_1	d_1
b_2	d_1
b_2	d_2

Рис. 12.12. Представление значения предиката отношением

equals $(X, 10)$. Однако между предикатами сравнения и прочими имеется важное различие. Предикаты сравнения имеют стандартную интерпретацию, в то время как обычные предикаты наподобие *in* означают только то, что определено программой Datalog (об этом будет сказано ниже).

Литерал представляет собой либо атом, либо его отрицание. Отрицание обозначается с использованием слова NOT перед атомом. Таким образом, NOT $in(B, D)$ является утверждением, что определение не достигает начала базового блока B.

12.3.2 Правила Datalog

Правила представляют собой способ выражения логических выводов. В Datalog правила служат также для предложения того, каким способом должны быть вычислены истинные факты. Правила имеют вид

$$H \;:-\; B_1 \;\&\; B_2 \;\&\; \cdots \;\&\; B_n$$

Ниже перечислены компоненты правила.

- H является атомом, а $B_1, B_2, \ldots, B_n$ являются литералами — либо атомами, либо их отрицаниями.
- H — *заголовок* правила, а $B_1, B_2, \ldots, B_n$ образуют его *тело*.
- Каждый из литералов B_i называется также *подцелью* (subgoal) правила.

Символ ": -" должен читаться как "если". Смысл правила — "заголовок истинен, если истинно тело". Говоря более точно, мы *применяем* правило к данному множеству основных атомов следующим образом. Рассмотрим все возможные подстановки констант вместо переменных в правиле. Если подстановка делает каждую подцель тела истинной (т.е. истинны все данные основные атомы, и только они), то можно сделать вывод, что заголовок при такой подстановке констант вместо переменных является истинным фактом. Подстановки, которые не делают истинными все подцели, не несут для нас никакой информации — заголовок может быть, а может и не быть истинным.

Программа Datalog представляет собой набор правил. Такая программа применяется к "данным", т.е. к множеству основных атомов для некоторых предикатов. Результатом выполнения программы является множество основных атомов,

Соглашения Datalog

В программах Datalog мы будем использовать следующие соглашения.

1. Переменные начинаются с прописной буквы.
2. Все остальные элементы начинаются со строчных букв или иных символов, таких как цифры. Эти элементы включают предикаты и константы, являющиеся аргументами предикатов.

полученное путем применения правил до тех пор, пока не будут сделаны все возможные выводы.

Пример 12.12. Простым примером Datalog-программы является вычисление путей в графе, заданном его (ориентированными) ребрами, т.е. существует предикат $edge(X, Y)$, который означает "существует ребро от узла X к узлу Y". Еще один предикат — $path(X, Y)$ — означает, что существует путь от узла X к узлу Y. Правила, определяющие пути, имеют следующий вид:

$$\begin{array}{lll} 1) & path(X,Y) & :- \quad edge(X,Y) \\ 2) & path(X,Y) & :- \quad path(X,Z) \ \& \ path(Z,Y) \end{array}$$

Первое правило гласит, что отдельное ребро является путем. Иначе говоря, если мы заменим переменную X константой a, а переменную Y — константой b и предикат $edge(a, b)$ истинен (т.е. имеется ребро из узла a в узел b), то истинен и предикат $path(a, b)$ (т.е. существует путь из узла a в узел b). Второе правило гласит, что если существует путь из некоторого узла X в некоторый узел Z, а также путь из узла Z в узел Y, то существует путь из узла X в узел Y. Это правило выражает "транзитивное замыкание". Заметим, что любой путь можно сформировать, беря ребра вдоль пути и многократно применяя правило транзитивного замыкания.

Предположим, например, что истинны следующие факты (основные атомы): $edge(1, 2)$, $edge(2, 3)$ и $edge(3, 4)$. Тогда можно воспользоваться первым правилом с тремя разными подстановками для вывода $path(1, 2)$, $path(2, 3)$ и $path(3, 4)$. Так, подстановка $X = 1$ и $Y = 2$ приводит первое правило к виду $path(1, 2) \ :- \ edge(1, 2)$. Поскольку $edge(1, 2)$ истинно, мы получаем истинность $path(1, 2)$.

Имея указанные три факта о предикате *path*, можно несколько раз применить к ним второе правило. Если подставить $X = 1$, $Z = 2$ и $Y = 3$, второе правило дает нам $path(1, 3) \ :- \ path(1, 2) \ \& \ path(2, 3)$. Так как обе подцели тела уже были выведены, нам известно, что обе они истинны, так что мы выводим и заголовок

path $(1,3)$. Подстановка $X = 1$, $Z = 3$ и $Y = 4$ позволяет нам вывести заголовок *path* $(1,4)$, т.е. мы выясняем, что существует путь из узла 1 в узел 4. □

12.3.3 Интенсиональные и экстенсиональные предикаты

В программах Datalog удобно различать два типа предикатов.

1. Предикаты *экстенсиональной базы данных* (extensional database — EDB) — это предикаты, которые определяются априори, т.е. их истинные факты либо заданы отношением или таблицей, либо вытекают из смысла предиката (например, в случае предикатов сравнения).

2. Предикаты *интенсиональной базы данных* (intensional database — IDB) определяются только правилами.

Предикат должен быть либо EDB-, либо IDB-предикатом, причем может принадлежать только к одному из этих типов. Таким образом, любой предикат, находящийся в заголовке одного или нескольких правил, обязан быть IDB-предикатом. Предикаты, находящиеся в теле правил, могут быть как EDB-, так и IDB-предикатами. Так, в примере 12.12 *edge* является EDB-предикатом, а *path* — IDB-предикатом. Вспомним, что у нас имелось несколько фактов *edge*, таких как *edge* $(1,2)$, но факты *path* выводились при помощи правил.

При использовании Datalog-программ для выражения алгоритмов потоков данных EDB-предикаты вычисляются из самого графа потока. IDB-предикаты выражаются при помощи правил, и задача потока данных решается путем вывода всех возможных IDB-фактов на основании правил и имеющихся EDB-фактов.

Пример 12.13. Рассмотрим, каким образом в Datalog можно выразить достигающие определения. Во-первых, имеет смысл работать на уровне инструкций, а не на уровне блоков, т.е. построение множеств *gen* и *kill* из базовых блоков будет интегрировано с вычислением самих достигающих определений. На рис. 12.13 представлен типичный блок. Обратите внимание, что мы идентифицируем точки внутри блока, нумеруя их $0, 1, \ldots, n$, где n — количество инструкций в блоке. i-е определение находится "в точке" i; в точке 0 никаких определений нет.

```
        0
           x = y+z
  b1    1
           *p = u
        2
           x = v
        3
```

Рис. 12.13. Базовый блок с точками между инструкциями

Точка в программе должна быть представлена парой (b, n), где b – имя блока, а n – целое число от 0 до количества инструкций в базовом блоке b. Наша формулировка требует двух EDB-предикатов.

1. Предикат $def(B, N, X)$ истинен тогда и только тогда, когда N-я инструкция в базовом блоке B может определять переменную X. Например, на рис. 12.13 $def(b_1, 1, x)$ истинно, $def(b_1, 3, x)$ истинно и $def(b_1, 2, Y)$ истинно для каждой возможной переменной Y, на которую в этой точке может указывать указатель p.

2. Предикат $succ\,(B, N, C)$ истинен тогда и только тогда, когда базовый блок C является преемником базового блока B в графе потока и B содержит N инструкций, т.е. управление может быть передано из точки N базового блока B в точку 0 блока C. Предположим, например, что предшественником блока b_1 на рис. 12.13 является блок b_2, состоящий из пяти инструкций. Тогда предикат $succ\,(b_2, 5, b_1)$ истинен.

Имеется также один IDB-предикат $rd\,(B, N, C, M, X)$. Он истинен тогда и только тогда, когда определение переменной X в M-й инструкции базового блока C достигает точки N базового блока B. Правила, определяющие предикат rd, показаны на рис. 12.14.

$$
\begin{array}{lll}
1) & rd(B, N, B, N, X) & \text{:-} \quad def(B, N, X) \\
 & & \\
2) & rd(B, N, C, M, X) & \text{:-} \quad rd(B, N-1, C, M, X)\ \& \\
 & & \qquad def(B, N, Y)\ \& \\
 & & \qquad X \neq Y \\
 & & \\
3) & rd(B, 0, C, M, X) & \text{:-} \quad rd(D, N, C, M, X)\ \& \\
 & & \qquad succ(D, N, B)
\end{array}
$$

Рис. 12.14. Правила для предиката *rd*

Правило 1 гласит, что если N-я инструкция базового блока B определяет X, то это определение X достигает N-й точки B (т.е. точки, следующей непосредственно за инструкцией). Это правило соответствует концепции "gen" в нашей ранней формулировке достигающих определений, основанной на теории множеств.

Правило 2 представляет идею о том, что определение проходит через инструкцию, если оно не уничтожается ею, и что единственный способ уничтожения определения заключается в переопределении переменной со 100% гарантией. Де-

тальнее правило 2 гласит, что определение переменной X из M-й инструкции блока C достигает точки N базового блока B, если

а) оно достигает предыдущей точки $N-1$ базового блока B и

б) существует как минимум одна переменная Y, отличная от X, которая может быть определена в N-й инструкции B.

Наконец, правило 3 выражает поток управления в графе. Оно гласит, что определение X в M-й инструкции базового блока C достигает точки 0 базового блока B, если существует некоторый базовый блок D с N инструкциями, такой, что определение X достигает конца базового блока D, а B является преемником базового блока D. □

EDB-предикат *succ* из примера 12.13 может быть выведен из графа потока. Можно также получить из графа потока и предикат *def*, если консервативно считать, что указатель может указывать на что угодно. Если мы хотим ограничить область указателя переменными соответствующего типа, то можно получить информацию о типах из таблицы символов и использовать меньшее отношение *def*. Один из вариантов состоит в том, чтобы сделать *def* IDB-предикатом и определить его при помощи правил. Эти правила будут использовать более примитивные EDB-предикаты, которые могут быть определены из графа потока и таблицы символов.

Пример 12.14. Предположим, что мы вводим два новых EDB-предиката.

1. Предикат *assign* (B, N, X) истинен, если N-я инструкция базового блока B имеет X в левой части. Заметим, что X может быть переменной или простым выражением с l-значением наподобие `*p`.

2. Предикат *type* (X, T) истинен, если типом X является T. Здесь X, опять же, может быть переменной или простым выражением с l-значением, а T может быть любым выражением корректного типа.

Тогда мы можем записать правила для *def*, делая его IDB-предикатом. На рис. 12.15 приведено расширение рис. 12.14 с двумя из возможных правил для *def*. Правило 4 гласит, что N-я инструкция базового блока B определяет X, если X присваивается N-й инструкцией. Правило 5 говорит о том, что переменная X может также быть определена N-й инструкцией базового блока B, если эта инструкция выполняет присваивание $*P$, а X является любой переменной типа, на который указывает указатель P. Другие виды присваиваний требуют иных правил для *def*.

В качестве примера использования правил на рис. 12.15 для выводов еще раз рассмотрим базовый блок b_1 на рис. 12.13. Первая инструкция присваивает значение переменной x, так что *assign* $(b_1, 1, x)$ является EDB-фактом. Третья инструкция также присваивает значение переменной x, так что *assign* $(b_1, 3, x)$ является

$$
\begin{array}{lrcl}
1) & rd(B,N,B,N,X) & :- & def(B,N,X) \\
2) & rd(B,N.C,M,X) & :- & rd(B,N-1,C,M,X) \text{ \& } \\
 & & & def(B,N,Y) \text{ \& } \\
 & & & X \neq Y \\
3) & rd(B,0,C,M,X) & :- & rd(D,N,C,M,X) \text{ \& } \\
 & & & succ(D.N,B) \\
4) & def(B,N,X) & :- & assign(B,N,X) \\
5) & def(B,N,X) & :- & assign(B,N,*P) \text{ \& } \\
 & & & type(X,T) \text{ \& } \\
 & & & type(P,*T)
\end{array}
$$

Рис. 12.15. Правила для предикатов *rd* и *def*

еще одним EDB-фактом. Вторая инструкция выполняет косвенное присваивание через указатель p, так что третьим EDB-фактом является $assign\,(b_1, 2, *p)$. Правило 4 позволяет нам вывести $def(b_1, 1, x)$ и $def(b_1, 3, x)$.

Предположим, что p имеет тип указателя на целое число (`*int`), а x и y являются целочисленными переменными. Тогда мы можем использовать правило 5 с $B = b_1$, $N = 2$, $P = p$, $T = int$ и X, равным либо x, либо y, что позволяет вывести $def(b_1, 2, x)$ и $def(b_1, 2, y)$. Аналогично можно вывести такой же факт и для любой другой переменной, тип которой — либо `int`, либо приводимый к нему. □

12.3.4 Выполнение программы Datalog

Каждое множество правил Datalog определяет отношения для его IDB-предикатов как функции отношений, которые задаются для EDB-предикатов. Начнем с предположения о том, что множество IDB-предикатов пустое (т.е. для всех возможных аргументов IDB-предикаты ложны). Затем, многократно применяя правила, будем выводить новые факты, когда того от нас будут требовать правила программы. Когда этот процесс сойдется, выполнение программы завершится, а ее вывод образуют полученные IDB-отношения. Этот процесс формализуется следующим алгоритмом, подобным итеративным алгоритмам, рассматривавшимся в главе 9.

Алгоритм 12.15. Простое вычисление программ Datalog

ВХОД: программа Datalog и множества фактов для каждого EDB-предиката.

ВЫХОД: множества фактов для каждого IDB-предиката.

МЕТОД: пусть для каждого предиката p в программе R_p – отношение фактов, истинных для данного предиката. Если p – EDB-предикат, то R_p является множеством фактов, задаваемых этим предикатом. Если же p – IDB-предикат, мы вычисляем R_p, выполняя алгоритм, приведенный на рис. 12.16. □

```
for (каждый IDB-предикат p)
    R_p = ∅;
while (имеются изменения в R_p) {
    рассмотрим все возможные подстановки констант вместо
        переменных во всех правилах;
    определим для каждой подстановки, являются ли все подцели
        тела истинными, используя текущее множество R_p
        для определения истинности EDB- и IDB-предикатов;
    if (подстановка делает тело правила истинным)
            добавляем заголовок в R_q, где q – предикат заголовка;
}
```

Рис. 12.16. Вычисление программы Datalog

Пример 12.16. Программа в примере 12.12 вычисляет пути в графе. Чтобы применить к ней алгоритм 12.15, начнем с EDB-предиката *edge*, выполняющегося для всех ребер графа, и пустого отношения для предиката *path*. На первом шаге правило 2 ничего нам не дает, поскольку факты *path* пока что отсутствуют. Однако правило 1 заставляет все факты *edge* стать фактами *path*. Иначе говоря, после первого цикла мы знаем, что *path* (a, b) истинно тогда и только тогда, когда существует ребро от a до b.

На втором шаге правило 1 не дает новых фактов, поскольку EDB-отношение *edge* никогда не изменяется. Однако теперь правило 2 позволяет нам собрать по два пути длиной 1 для образования путей длиной 2, т.е. после второго цикла *path* (a, b) истинно тогда и только тогда, когда от a к b существует путь длиной 1 или 2. Аналогично на третьем шаге комбинируются пути длиной 2 или менее для получения всех путей длиной не более 4. На четвертом шаге выполняется поиск путей длиной до 8, и в общем случае после i-го шага *path* (a, b) истинно тогда и только тогда, когда существует путь от a в b длиной не более 2^{i-1}. □

Инкрементное вычисление множеств

Инкрементно могут быть решены и задачи потоков данных в формулировке с использованием теории множеств. Например, в случае достигающих определений определение может быть вновь открыто в $\text{IN}[B]$ на i-м цикле, только если оно было только что найдено в $\text{OUT}[P]$ некоторого предшественника P базового блока B. Причина, по которой мы не пытались инкрементно решать задачи потоков данных, заключается в эффективности реализации множеств при помощи векторов битов. В общем случае оказывается проще пройти по всем векторам, чем выяснять, является ли некоторый факт новым.

12.3.5 Инкрементное вычисление программ Datalog

Возможно повышение эффективности алгоритма 12.15. Заметим, что новый IDB-факт на шаге i может быть открыт, только если он является результатом подстановки констант в правило, такое, что как минимум одна из подцелей становится фактом, открытым на предыдущем шаге $i - 1$. Доказательство этого утверждения заключается в том, что если бы все факты подцелей были известны на шаге $i - 2$, то "новые" факты должны бы были быть открыты при помощи той же подстановки констант на шаге $i - 1$.

Чтобы воспользоваться этим наблюдением, введем для каждого IDB-предиката p предикат *newP*, который выполняется только для вновь открытых p-фактов из предыдущего шага. Каждое правило, которое содержит среди своих подцелей один или несколько IDB-предикатов, заменяется набором правил. Каждое правило в наборе образуется заменой на *newQ* ровно одного вхождения некоторого IDB-предиката q в тело. Наконец, для всех правил предикат заголовка h заменяется на *newH*. Получающиеся в результате правила назовем *инкрементной формой*.

Отношения для каждого IDB-предиката p накапливают все p-факты, как и в алгоритме 12.15. В одном цикле делается следующее.

1. Правила применяются для вычисления предикатов *newP*.
2. Из *newP* вычитается p, тем самым обеспечивается истинная новизна фактов в *newP*.
3. Добавляем в p все факты из *newP*.
4. Делаем для следующего цикла все отношения *newX* пустыми множествами.

Эти идеи формализованы в алгоритме 12.18, но перед тем как перейти к нему, рассмотрим еще один пример.

Пример 12.17. Вновь вернемся к программе Datalog из примера 12.12. Соответствующая инкрементная форма правил приведена на рис. 12.17. Правило 1 не претерпело изменений, за исключением заголовка, поскольку IDB-подцелей в теле этого правила нет. Однако правило 2 с двумя IDB-подцелями превратилось в два разных правила. В каждом из них одно из вхождений *path* в тело правила заменено на *newPath*. Вместе эти правила выражают идею, что как минимум один из двух соединяемых правилом путей должен быть открыт на предыдущем шаге. □

$$\begin{array}{llll} 1) & newPath(X,Y) & :- & edge(X,Y) \\ \\ 2a) & newPath(X,Y) & :- & path(X,Z)\ \& \\ & & & newPath(Z,Y) \\ \\ 2b) & newPath(X,Y) & :- & newPath(X,Z)\ \& \\ & & & path(Z,Y) \end{array}$$

Рис. 12.17. Инкрементные правила для программы Datalog для путей в графе

Алгоритм 12.18. Инкрементное вычисление программы Datalog

Вход: программа Datalog и множества фактов для каждого EDB-предиката.

Выход: множества фактов для каждого IDB-предиката.

Метод: пусть для каждого предиката p в программе R_p — отношение фактов, истинных для данного предиката. Если p — EDB-предикат, то R_p является множеством фактов, задаваемых этим предикатом. Если же p — IDB-предикат, мы вычисляем R_p. Кроме того, пусть R_{newP} для каждого IDB-предиката p является отношением из "новых" фактов для предиката p.

1. Преобразуем правила в описанную ранее инкрементную форму.

2. Выполняем алгоритм, приведенный на рис. 12.18. □

12.3.6 Проблематичные правила Datalog

Имеются как некоторые правила Datalog, так и программы, которые технически не имеют смысла и не должны использоваться. Две наиболее важные опасности заключаются в следующем.

1. *Небезопасные правила.* Это правила, в заголовках которых находятся переменные, не появляющиеся в теле таким образом, чтобы они могли принимать только значения из EDB.

```
for (каждый IDB-предикат p) {
        R_p = ∅;
        R_newP = ∅;
}
repeat {
        рассмотрим все возможные подстановки констант вместо
                переменных во всех правилах;
        определим для каждой подстановки, являются ли все подцели
                тела истинными, используя текущие множества R_p и R_newP
                для определения истинности EDB- и IDB-предикатов;
        if (подстановка делает тело правила истинным)
                        добавляем заголовок в R_newH, где h — предикат заголовка;
        for (каждый предикат p) {
                R_newP = R_newP − R_p;
                R_p = R_p ∪ R_newP;
        }
} until (все R_newP — пустые);
```

Рис. 12.18. Вычисление программы Datalog

2. *Нестратифицированные программы.* Это множества правил, в которых имеется рекурсия, включающая отрицание.

Далее мы рассмотрим обе эти опасности.

Безопасность правил

Любая переменная, которая появляется в заголовке правила, должна иметься и в его теле. Более того, она должна находиться в подцели, которая является либо обычным IDB, либо EDB-атомом. Неприемлемо, если переменная находится только в атоме-отрицании или только в операторе сравнения. Данная стратегия применяется для того, чтобы избежать правил, которые приведут нас к выводу бесконечного количества фактов.

Пример 12.19. Правило

$$p(X,Y) \text{ :- } q(Z) \text{ \& NOT } r(X) \text{ \& } X \neq Y$$

небезопасно по двум причинам. Переменная X находится только в отрицании подцели $r(X)$ и в сравнении $X \neq Y$. Y находится только в сравнении. Следствием этого является то, что предикат p истинен для бесконечного количества пар (X, Y), лишь бы $r(X)$ было ложно, а Y отличалось от X. □

Стратифицированный Datalog

Чтобы программа имела смысл, рекурсия и отрицание должны быть разделены. Формальное требование имеет следующий вид. Мы должны иметь возможность разделить IDB-предикаты на *страты*, так что, если имеется правило с заголовком p и подцелью вида NOT $q(\ldots)$, q — либо EDB-, либо IDB-предикат в более низкой по сравнению с p страте. Если этот принцип выполняется, то страты могут быть вычислены в восходящем порядке при помощи алгоритма 12.15 или 12.18, после чего можно рассматривать отношения IDB-предикатов данной страты, как будто это EDB-предикаты для вычисления более высоких страт. Однако при нарушении указанного принципа итеративный алгоритм может не достичь сходимости, что проиллюстрировано следующим примером.

Пример 12.20. Рассмотрим программу Datalog, состоящую из одного правила:

$$p(X) \ :\text{-}\ e(X) \ \& \ \texttt{NOT}\, p(X)$$

Предположим, что e — EDB-предикат, причем истинно только $e(1)$. Истинно ли $p(1)$?

Данная программа не стратифицирована. В какую бы страту мы не поместили предикат p, его правило будет содержать подцель, которая представляет собой отрицание и включает IDB-предикат (сам предикат p), располагающийся, само собой разумеется, не в более низкой страте, чем p.

Если применить итеративный алгоритм, начиная с $R_p = \emptyset$, то первым ответом будет "Нет; $p(1)$ не истинно". Но первая итерация позволит нам вывести $p(1)$, поскольку и $e(1)$, и NOT $p(1)$ истинны. Вторая итерация вновь приведет нас к выводу, что $p(1)$ ложно (на этот раз подцель NOT $p(1)$ будет ложна). Третья итерация вновь делает $p(1)$ истинным, четвертая — ложным, и т.д. Мы вынуждены сделать вывод, что данная нестратифицированная программа не имеет смысла, и не рассматривать ее как корректную. □

12.3.7 Упражнения к разделу 12.3

! **Упражнение 12.3.1.** Здесь мы рассмотрим более простой по сравнению с примером 12.13 анализ достигающих определений. Предположим, что каждая инструкция сама по себе является блоком и изначально определяет ровно одну переменную. EDB-предикат *pred* (I, J) означает, что инструкция I является предшественницей инструкции J. EDB-предикат *defines* (I, X) означает, что переменная, определяемая инструкцией I, — X. Мы будем использовать IDB-предикаты *in* (I, D) и *out* (I, D), означающие, что определение D достигает соответственно начала или конца инструкции I. Заметим, что определение фактически является номером инструкции. На рис. 12.19 приведена Datalog-программа, которая выражает обычный алгоритм вычисления достигающих определений.

$$
\begin{array}{llll}
1) & kill(I, D) & :- & defines(I, X) \ \& \ defines(D, X) \\
2) & out(I, I) & :- & defines(I, X) \\
3) & out(I, D) & :- & in(I, D) \ \& \ \text{NOT}\ kill(I, D) \\
4) & in(I, D) & :- & out(J, D) \ \& \ pred(J, I)
\end{array}
$$

Рис. 12.19. Datalog-программа для простого анализа достигающих определений

Правило 1 гласит, что инструкция уничтожает саму себя, правило 2 — что инструкция входит в собственное "выходное множество". Правило 3 является обычной передаточной функцией, а правило 4 позволяет слияние, так как I может иметь несколько предшественников.

Ваша задача — модифицировать правила для обработки распространенной ситуации неоднозначных определений, например присваиваний посредством указателя. В этой ситуации *defines* (I, X) может быть истинно для нескольких различных X и одного I. Определение лучше всего представить парой (D, X), где D — инструкция, а X — одна из переменных, которая может быть определена в D. В результате *in* и *out* станут трехаргументными предикатами; например, *in* (I, D, X) означает, что (возможное) определение X в инструкции D достигает начала инструкции I.

Упражнение 12.3.2. Напишите Datalog-программу, аналогичную приведенной на рис. 12.19, для вычисления доступных выражений. В дополнение к предикату *defines* используйте предикат *eval* (I, X, O, Y), который гласит, что инструкция I вычисляет выражение XOY. Здесь O — оператор в выражении, например +.

Упражнение 12.3.3. Напишите Datalog-программу, аналогичную приведенной на рис. 12.19, для вычисления активных переменных. В дополнение к предикату *defines* воспользуйтесь предикатом *use* (I, X), который гласит, что инструкция I использует переменную X.

Упражнение 12.3.4. В разделе 9.5 мы определили вычисления потока данных, включающие шесть концепций: ожидаемого, доступного, самого раннего, откладываемого, самого позднего и используемого выражений. Предположим, что мы написали Datalog-программу для определения каждой из них в терминах EDB-концепций, выводимых из программы (например, из информации gen и kill), и остальных из данных шести концепций. Какие из концепций зависимы от других? Какие из зависимостей используют отрицания? Является ли полученная Datalog-программа стратифицированной?

Упражнение 12.3.5. Предположим, что EDB-предикат $edge(X, Y)$ состоит из следующих фактов:

$$edge(1,2) \quad edge(2,3) \quad edge(3,4)$$
$$edge(4,1) \quad edge(4,5) \quad edge(5,6)$$

а) Смоделируйте Datalog-программу из примера 12.12 для этих данных с использованием простой стратегии вычисления из алгоритма 12.15. Укажите, какие факты *path* выявляются на каждом цикле вычислений.

б) Смоделируйте Datalog-программу из примера 12.12 для этих данных с использованием инкрементной стратегии вычисления из алгоритма 12.18. Укажите, какие факты *path* выявляются на каждом цикле вычислений.

Упражнение 12.3.6. Правило

$$p(X,Y) \ :- \ q(X,Z) \ \& \ r(Z,W) \ \& \ \texttt{NOT} \ p(W,Y)$$

является частью большей Datalog-программы P.

а) Укажите заголовок, тело и подцели данного правила.

б) Какие предикаты определенно являются IDB-предикатами программы P?

! в) Какие предикаты определенно являются EDB-предикатами программы P?

г) Является ли это правило безопасным?

д) Стратифицирована ли программа P?

Упражнение 12.3.7. Преобразуйте правила на рис. 12.14 в инкрементную форму.

12.4 Простой алгоритм анализа указателей

В этом разделе мы начнем рассмотрение очень простого, нечувствительного к потоку, анализа псевдонимов указателей в предположении отсутствия вызовов процедур. В последующих разделах мы покажем, как работать с процедурами — сначала контекстно-нечувствительно, а затем контекстно-чувствительно. Чувствительность к потоку добавляет массу сложностей; и она не слишком важна для языков программирования наподобие Java, в которых наблюдается тенденция к наличию методов небольшого размера.

Фундаментальным вопросом в анализе псевдонимов указателей является вопрос о том, может ли пара данных указателей быть псевдонимом. Один из способов ответить на этот вопрос состоит в вычислении каждого указателя для получения ответа на вопрос "На какой объект может указывать данный указатель?" Если два указателя могут указывать на один и тот же объект, указатели могут быть псевдонимами.

12.4.1 Сложность анализа указателей

Особенно сложен анализ указателей для С-программ, поскольку в них над указателями могут выполняться произвольные вычисления. Фактически в С-программе указателю может быть присвоено считанное целочисленное значение, что делает указатель потенциальным псевдонимом для всех указателей в программе. Указатели в Java, известные как ссылки, существенно проще. Над ними не могут выполняться никакие арифметические операции, и указатели могут указывать только на начало объекта.

Анализ псевдонимов указателей должен быть межпроцедурным. Без этого следует полагать, что любой вызванный метод может изменить содержимое всех доступных переменных-указателей, делая тем самым внутрипроцедурный анализ неэффективным.

Языки, в которых разрешены косвенные вызовы процедур, вносят дополнительный вклад в сложность анализа указателей. В языке программирования С можно вызвать функцию косвенно, путем вызова разыменованного указателя на функцию. Перед тем как приступить к анализу вызываемой функции, мы должны знать, на что именно может указывать данный указатель на функцию. Понятно, что после анализа вызываемой функции могут обнаружиться новые функции, на которые может указывать рассматриваемый указатель, так что процесс анализа должен быть итеративным.

В то время как в С большинство функций все же вызываются непосредственно, виртуальные методы Java приводят к тому, что многие вызовы оказываются косвенными. В случае некоторого вызова `x.m()` в Java-программе может иметься множество классов, в которых есть метод m и которым может принадлежать объект x. Чем более точны наши знания о типе x, тем точнее оказывается граф вызовов. В идеальном случае во время компиляции можно определить точный тип x, а значит, точно указать, какой метод m будет вызван.

Пример 12.21. Рассмотрим последовательность инструкций Java:

```
Object o;
o = new String();
n = o.hashCode();
```

Здесь o объявлено как `Object`. Без анализа, на что именно ссылается o, в качестве возможных целевых методов должны рассматриваться все методы с именем `hashCode` для всех классов. Знание же о том, что o указывает на объект типа `String`, сузит результаты межпроцедурного анализа до единственного метода `length`, объявленного в классе `String`. □

Для сокращения количества целевых методов можно применить аппроксимацию. Например, можно статически определить, какие типы объектов создаются,

и ограничиться только их анализом. Можно быть и более точным, если получится строить граф вызовов "на лету" на основе выполняемого одновременно анализа, на что именно указывают указатели. Более точные графы вызовов приводят не только к более точным результатам, но и существенно сокращают время, затрачиваемое на выполнение анализа.

Такой анализ целей указателей достаточно сложен. Это не одна из тех "простых" задач потока данных, когда достаточно смоделировать проход по циклу инструкций. При обнаружении новой цели указателя все инструкции, присваивающие значение этого указателя другому, должны быть проанализированы заново.

Для простоты мы сосредоточимся, в основном, на языке программирования Java. Начнем с анализа, нечувствительного к потоку и контексту, полагая для начала, что в программе не вызываются никакие методы. Затем опишем, как можно строить граф вызовов "на лету" при вычислении результатов анализа целей указателей. И наконец мы опишем один из контекстно-чувствительных методов.

12.4.2 Модель указателей и ссылок

Предположим, что наш язык программирования имеет следующие способы представления и работы со ссылками.

1. Некоторые переменные в программе имеют тип "указатель на T" или "ссылка на T", где T – тип языка программирования. Эти переменные либо статические, либо активные в стеке времени выполнения. Далее мы будем называть их просто *переменными*.

2. Имеется куча объектов. Все переменные указывают на объекты кучи, но не на иные переменные. Такие объекты мы будем называть *объектами кучи* (heap objects).

3. Объект кучи может иметь *поля*, а значение поля может быть ссылкой на объект кучи (но не на переменную).

Такая структура хорошо моделирует язык программирования Java, и в примерах мы будем использовать синтаксис Java. Заметим, что язык программирования C моделируется существенно хуже, поскольку переменные-указатели в C могут указывать на другие переменные-указатели, и, в принципе, любое значение C может быть преобразовано в указатель.

Поскольку мы выполняем нечувствительный анализ, достаточно утверждения, что данная переменная v может указывать на данный объект кучи h; нас не интересует вопрос, где именно в программе v может указывать на h или в каком контексте v может указывать на h. Заметим, однако, что переменные могут быть именованы при помощи их полных имен. В Java такое полное имя может включать модуль, класс, метод и блок в методе наряду с самим именем переменной.

Таким образом, мы можем различать несколько переменных с одним и тем же идентификатором.

Объекты кучи не имеют имен. Динамически может быть создано неограниченное количество объектов. Мы будем ссылаться на объекты с использованием инструкций, в которых они были созданы. Поскольку инструкция может выполняться многократно и каждый раз создавать новый объект кучи, утверждение наподобие "v может указывать на h" на самом деле означает "v может указывать на один или несколько объектов, созданных в инструкции с меткой h".

Цель анализа состоит в определении того, на что может указывать каждая переменная и каждое поле каждого объекта кучи. Будем называть такой анализ *анализом целей указателей* (points-to analysis); два указателя являются псевдонимами, если их целевые множества пересекаются. Здесь мы опишем анализ, *основанный на включении* (inclusive-based); т.е. инструкция вида `v = w` приводит к тому, что переменная v указывает на все объекты, на которые указывает w, но не наоборот. Хотя этот подход и кажется очевидным, существуют и иные альтернативы определения анализа целей указателей. Например, можно определить анализ, *основанный на эквивалентности* (equivalence-based), такой, что инструкция вида `v = w` помещает переменные v и w в один класс эквивалентности, указывающий на все переменные, на которые может указывать каждый из указателей. Хотя эта формулировка и не обеспечивает хорошего приближения при вычислении псевдонимов, она предоставляет быстрый, а часто и достаточно хороший ответ на вопрос о том, какие переменные указывают на один и тот же вид объектов.

12.4.3 Нечувствительность к потоку

Начнем с очень простого примера для иллюстрации влияния игнорирования потока управления в анализе целей указателей.

Пример 12.22. На рис. 12.20 создаются три объекта, h, i и j, которые присваиваются соответственно переменным a, b и c. Таким образом, по окончании строки 3 a указывает на h, b — на i, c — на j.

```
1) h:  a = new Object();
2) i:  b = new Object();
3) j:  c = new Object();
4)     a = b;
5)     b = c;
6)     c = a;
```

Рис. 12.20. Код Java к примеру 12.22

Если проследовать по строкам 4–6, то после строки 4 выяснится, что a указывает только на i. После строки 5 b указывает на j, а после строки 6 c указывает на i. □

Приведенный выше анализ чувствителен к потоку, поскольку мы следуем за потоком управления и после каждой инструкции выясняем, на что может указывать каждая переменная. Другими словами, наряду с выяснением, какая информация о целях указателей "генерируется" каждой инструкцией, мы также рассматриваем, какая информация о целях указателей "уничтожается". Например, инструкция `b = c;` уничтожает предыдущий факт "b указывает на i" и генерирует новое соотношение "b указывает на то же, на что и c".

Нечувствительный к потоку анализ игнорирует поток управления, что по сути эквивалентно предположению, что все инструкции программы могут выполняться в произвольном порядке. При этом анализе вычисляется только одно глобальное отображение, указывающее, на что может указывать каждая переменная в любой точке выполнения программы. Если переменная может указывать на два разных объекта после двух разных инструкций программы, мы просто записываем, что она может указывать на два объекта. Другими словами, в анализе, нечувствительном к потоку, присваивание не уничтожает отношения указывания, а только их генерирует. Для вычисления нечувствительных к потоку результатов мы многократно добавляем цели указателей для каждой инструкции, пока новые цели не перестанут обнаруживаться. Понятно, что отсутствие чувствительности к потоку существенно ослабляет результаты анализа, но при таком подходе снижается размер представления результатов и быстрее достигается сходимость алгоритма.

Пример 12.23. Вернемся к примеру 12.22. Строки 1–3 говорят нам, что a может указывать на h, b может указывать на i, а c может указывать на j. После строк 4 и 5 a может указывать на h и i, а b — на i и j. Анализ строки 6 дает нам, что c может указывать на h, i и j. Эта информация воздействует на строку 5, которая, в свою очередь, воздействует на строку 4. В конце концов мы делаем бесполезный вывод, что любой указатель может указывать на любой объект. □

12.4.4 Формулировка с применением Datalog

Формализуем нечувствительный к потоку анализ целей указателей, основанный на рассмотренном выше материале. Пока что мы проигнорируем вызовы процедур и сконцентрируемся на четырех видах инструкций, которые могут влиять на значения указателей.

1. *Создание объекта.* `h: T v = new T();`. Данная инструкция создает новый объект кучи, и переменная v может указывать на него.

2. *Инструкция копирования.* `v = w;`. Здесь v и w — переменные. Инструкция делает переменную v указывающей на тот же объект кучи, на который указывает переменная w, т.е. значение w копируется в v.

3. *Сохранение в поле.* `v.f = w;`. Тип объекта, на который указывает v, должен иметь поле f некоторого ссылочного типа. Пусть v указывает на объект кучи h, а w указывает на g. Данная инструкция делает поле f объекта h указывающим на g. Заметим, что переменная v остается при этом неизменной.

4. *Загрузка поля.* `v = w.f;`. Здесь w — переменная, указывающая на некоторый объект в куче, который имеет поле f, и f указывает на некоторый объект кучи h. Инструкция делает переменную v указывающей на h.

Заметим, что обращение к составному полю в исходном коде наподобие `v = w.f.g` разбивается на две примитивные инструкции загрузки поля:

```
v1 = w.f;
v  = v1.g;
```

Выразим наш анализ формально при помощи правил Datalog. Имеется только два IDB-предиката, которые мы должны вычислять.

1. *pts* (V, H) означает, что переменная V может указывать на объект кучи H.

2. *hpts* (H, F, G) означает, что поле F объекта кучи H может указывать на объект кучи G.

EDB-отношения строятся на основе исходного текста программы. Поскольку положение инструкций в программе при нечувствительном к потоку анализе не имеет значения, в EDB достаточно указать существование инструкций, имеющих определенный вид. Далее мы сделаем удобное упрощение. Вместо определения EDB-отношений для хранения информации, полученной из исходного текста программы, мы будем использовать заключенные в кавычки инструкции, указывающие EDB-отношение или отношения, которые представляет данная инструкция. Например, "$H\ :\ T\ V\ =\ \text{new}\ T\,()$" является EDB-фактом, утверждающим, что в инструкции H имеется присваивание, которое делает переменную V указывающей на новый объект типа T. Мы полагаем, что на практике будет иметься соответствующее EDB-отношение, которое будет заполнено основными атомами, по одному для каждой инструкции данного вида в программе.

При таком соглашении все, что нам надо для написания Datalog-программы, — это по одному правилу для каждого из четырех типов инструкций. Такая программа показана на рис. 12.21. Правило 1 гласит, что переменная V может указывать

на объект кучи H, если инструкция H представляет собой присваивание нового объекта переменной V. Правило 2 гласит, что если существует инструкция копирования `V = W` и W может указывать на H, то V может указывать на H.

$$\begin{array}{llll}
1) & pts(V,H) & :- & \text{“}H:\ T\ V\ = \texttt{new}\ T\text{”} \\
 & & & \\
2) & pts(V,H) & :- & \text{“}V = W\text{”}\ \& \\
 & & & pts(W,H) \\
 & & & \\
3) & hpts(H,F,G) & :- & \text{“}V.F = W\text{”}\ \& \\
 & & & pts(W,G)\ \& \\
 & & & pts(V,H) \\
 & & & \\
4) & pts(V,H) & :- & \text{“}V = W.F\text{”}\ \& \\
 & & & pts(W,G)\ \& \\
 & & & hpts(G,F,H)
\end{array}$$

Рис. 12.21. Datalog-программа для анализа указателей, нечувствительного к потоку

Правило 3 гласит, что если существует инструкция вида `V.F = W` и W может указывать на G, а V может указывать на H, то поле F объекта H может указывать на G. Наконец, правило 4 гласит, что если существует инструкция вида `V = W.F` и W может указывать на G, а поле F объекта G может указывать на H, то V может указывать на H. Обратите внимание на взаимную рекурсию *pts* и *hpts*; при этом данная Datalog-программа может быть вычислена любым из итеративных алгоритмов, рассматривавшихся в разделе 12.3.4.

12.4.5 Использование информации о типе

Поскольку Java — язык программирования, безопасный с точки зрения типов, переменные могут указывать только на типы, совместимые с объявленными. Например, присваивание объекта, принадлежащего к надклассу объявленного типа переменной, приведет к исключению времени выполнения. Рассмотрим простой пример, показанный на рис. 12.22, где S является подклассом T. Если в данной программе значение p истинно, то будет сгенерировано исключение времени выполнения, поскольку a нельзя присваивать объект класса T. Таким образом, статически можно заключить, что в силу ограничений, связанных с типами, a может указывать только на h, но не на g.

```
        S a;
        T b;
        if (p) {
g:          b = new T();
        } else
h:          b = new S();
        }
        a = b;
```

Рис. 12.22. Java-программа с ошибкой, связанной с типами

Таким образом, мы вводим в наш анализ три EDB-предиката, которые отражают важную информацию о типах в анализируемом коде.

1. *vType* (V, T) гласит, что переменная V объявлена как имеющая тип T.
2. *hType* (H, T) говорит о том, что тип созданного объекта кучи $H - T$. Тип создаваемого объекта может не быть точно известен, если, например, объект возвращается методом. В таком случае консервативно предполагается, что тип объекта может быть любым.
3. *assignable* (T, S) означает, что объект типа S может быть присвоен переменной с типом T. Эта информация в общем случае собирается из объявлений подтипов в программе, но сюда входит и информация о предопределенных классах языка. *assignable* (T, T) всегда истинно.

Правила на рис. 12.21 могут быть изменены таким образом, чтобы можно было делать выводы, только если переменной присваивается объект кучи совместимого типа. Модифицированные правила показаны на рис. 12.23.

Первое изменение внесено в правило 2. Последние три подцели говорят нам, что заключение о том, что V может указывать на H, можно сделать, только если типы T и S, которые могут иметь переменная V и объект кучи H, таковы, что объект типа S может быть присвоен переменной, которая является ссылкой на T. Подобное дополнительное ограничение добавлено и к правилу 4. Заметим, что в правиле 3 дополнительных ограничений нет, поскольку все присваивания должны выполняться через переменные, типы которых уже проверены. Любое ограничение, связанное с типами, перехватывает один дополнительный случай, когда базовый объект представляет собой нулевую константу.

12.4.6 Упражнения к разделу 12.4

Упражнение 12.4.1. На рис. 12.24 h и g представляют собой метки, использующиеся для представления вновь создаваемых объектов, и не являются частью

$$\begin{array}{rrcl}
1) & pts(V,H) & :- & \text{“}H:\ T\ V\ = \texttt{new}\ T\text{”} \\
\\
2) & pts(V,H) & :- & \text{“}V = W\text{”}\ \& \\
 & & & pts(W,H)\ \& \\
 & & & vType(V,T)\ \& \\
 & & & hType(H,S)\ \& \\
 & & & assignable(T,S) \\
\\
3) & hpts(H,F,G) & :- & \text{“}V.F = W\text{”}\ \& \\
 & & & pts(W,G)\ \& \\
 & & & pts(V,H) \\
\\
4) & pts(V,H) & :- & \text{“}V = W.F\text{”}\ \& \\
 & & & pts(W,G)\ \& \\
 & & & hpts(G,F,H)\ \& \\
 & & & vType(V,T)\ \& \\
 & & & hType(H,S)\ \& \\
 & & & assignable(T,S)
\end{array}$$

Рис. 12.23. Добавление к нечувствительному к потоку анализу указателей ограничений, связанных с типами

кода. Вы можете считать, что объект типа T имеет поле f. Воспользуйтесь Datalog-правилами из данного раздела для вывода всех возможных фактов *pts* и *hpts*.

```
h: T a = new T();
g: T b = new T();
   T c = a;
   a.f = b;
   b.f = c;
   T d = c.f;
```

Рис. 12.24. Код к упражнению 12.4.1

! Упражнение 12.4.2. Применение алгоритма из данного раздела к коду

```
g: T a = new T();
h:   a = new T();
   T c = a;
```

приводит к выводу, что и a, и c могут указывать и на h, и на g. Если код будет иметь вид

```
g: T a = new T();
h: T b = new T();
   T c = b;
```

то внимательное рассмотрение позволяет сделать вывод о том, что a не может указывать на h. Предложите внутрипроцедурный анализ потока данных, который может избежать неточности такого вида.

! **Упражнение 12.4.3.** Анализ из данного раздела можно сделать межпроцедурным, если моделировать вызов и возврат из процедуры так, как если бы это были операции копирования, как в правиле 2 на рис. 12.21. Иными словами, вызов копирует фактические параметры в соответствующие формальные, а возврат копирует переменную, в которой хранится возвращаемое значение, в переменную, которой присваивается результат вызова. Рассмотрите программу на рис. 12.25.

а) Выполните нечувствительный анализ этого кода.

б) Некоторые из сделанных в части *а* выводов оказываются "фальшивыми" в том смысле, что не представляют ни одного события, происходящего во время выполнения. Проблема связана с множественными присваиваниями переменной *b*. Перепишите код, показанный на рис. 12.25, так, чтобы ни одна переменная не присваивалась более одного раза. Выполните анализ заново и покажите, что каждый выведенный факт *pts* и *hpts* осуществляется во время выполнения программы.

```
t p(t x) {
    h: T a = new T;
       a.f = x;
       return a;
}

void main() {
    g: T b = new T;
         b = p(b);
         b = b.f;
}
```

Рис. 12.25. Пример кода для анализа указателей

12.5 Контекстно-нечувствительный межпроцедурный анализ

Теперь рассмотрим вызовы методов. Сначала поясним, как анализ целей указателей может использоваться для вычисления точного графа вызовов, который применяется при вычислении точных результатов целей указателей. Затем мы формализуем построение графа вызовов "на лету" и покажем, как для краткого описания анализа может использоваться Datalog.

12.5.1 Влияние вызовов методов

Влияние вызовов методов в Java, таких как `x = y.n(z)`, на отношения указания может быть вычислено следующим образом.

1. Определяется тип вызываемого объекта, который является объектом, на который указывает y. Предположим, что его тип — t. Пусть m — метод с именем n в последнем надклассе t, в котором имеется метод с именем n. Заметим, что в общем случае выяснить, какой метод будет вызван, можно только динамически.

2. Формальным параметрам m присваиваются объекты, на которые указывают фактические параметры. Фактические параметры включают не только параметры, передаваемые непосредственно, но и сам вызываемый объект. Каждый вызов метода присваивает вызываемый объект переменной `this`.[3] Эти переменные `this` будут рассматриваться нами как нулевые формальные параметры методов. В вызове `x = y.n(z)` два формальных параметра: объект, на который указывает переменная y и который присвоен переменной `this`, и объект, на который указывает z и который присваивается первому объявленному формальному параметру m.

3. Объект, возвращаемый m, присваивается переменной, находящейся слева в инструкции присваивания.

В контекстно-нечувствительном анализе параметры и возвращаемые значения моделируются инструкциями копирования. Остается вопрос, как определить тип вызываемого объекта. Можно консервативно определить тип в соответствии с объявлением переменной; например, если объявленная переменная имеет тип t, то могут быть вызваны методы с именем n в подтипах t. К сожалению, если объявленная переменная имеет тип `Object`, то потенциальными целевыми методами являются все методы с именем n. В реальных программах, использующих

[3]Вспомним, что переменные различаются методами, к которым они принадлежат, так что имеется много переменных `this`, но в каждом методе программы — только одна такая переменная.

иерархии объектов и включающих много больших библиотек, такой подход может привести к множеству ложных целей вызова, замедляя анализ и делая его менее точным.

Чтобы вычислить цели вызова, нам надо знать, на что именно может указывать переменная; но пока мы не знаем цели вызова, мы не можем найти все, на что указывают переменные. Такое рекурсивное соотношение требует обнаружения целей вызовов "на лету" при вычислении множества целей указателя. Анализ продолжается до тех пор, пока на очередной итерации мы не сможем обнаружить ни новые цели вызовов, ни новые отношения указания.

Пример 12.24. В коде на рис. 12.26 r является подтипом s, который, в свою очередь, является подтипом t. Если использовать только информацию из объявлений типов, `a.n()` может вызывать любой из трех объявленных методов с именами n, поскольку s и r являются подтипами объявленного типа a — t. Кроме того, после строки 5 a может указывать на объекты g, h и i.

```
            class t {
1) g:          t n() { return new r(); }
            }
            class s extends t {
2) h:          t n() { return new s(); }
            }
            class r extends s {
3) i:          t n() { return new r(); }
            }

            main () {
4) j:          t a = new t();
5)                a = a.n();
            }
```

Рис. 12.26. Вызов виртуального метода

Анализируя отношение указывания, мы сначала определяем, что a может указывать на j — объект типа t. Таким образом, целью вызова является метод, объявленный в строке 1. Анализируя строку 1, мы определяем, что a может также указывать на g, объект типа r. Таким образом, целью вызова может быть метод, объявленный в строке 3, а a может указывать на i — другой объект типа r. Поскольку новых целей вызова нет, анализ завершается без рассмотрения метода, объявленного в строке 2, и без заключения о том, что a может указывать на h. □

12.5.2 Построение графа вызовов в Datalog

Чтобы сформулировать правила Datalog для контекстно-нечувствительного межпроцедурного анализа, мы вводим три EDB-предиката, каждый из которых легко получается из исходного кода.

1. *actual* (S, I, V) гласит, что V — I-й фактический параметр, использованный в точке вызова S.
2. *formal* (M, I, V) гласит, что V — I-й формальный параметр, объявленный в методе M.
3. *cha* (T, N, M) гласит, что M — метод, вызываемый, когда вызывается метод N вызываемого объекта типа T (*cha* — сокращение от "class hierarchy analysis", "анализ иерархии классов").

Каждое ребро графа вызовов представлено IDB-предикатом *invokes*. При обнаружении новых ребер графа вызовов создаются новые отношения указания в качестве передаваемых параметров и возвращаемых значений. Все это подытожено правилами, приведенными на рис. 12.27.

$$
\begin{array}{lrcl}
1) & invokes(S, M) & :- & \text{“}S:\ V.N(...)\text{”}\ \& \\
 & & & pts(V, H)\ \& \\
 & & & hType(H, T)\ \& \\
 & & & cha(T, N, M) \\
 & & & \\
2) & pts(V, H) & :- & invokes(S, M)\ \& \\
 & & & formal(M, I, V)\ \& \\
 & & & actual(S, I, W)\ \& \\
 & & & pts(W, H)
\end{array}
$$

Рис. 12.27. Datalog-программа для построения графа вызовов

Первое правило вычисляет цель точки вызова, т.е. "$S:\ V.N\ (\ldots)$" гласит, что существует точка вызова с меткой S, которая вызывает метод с именем N объекта, на который указывает V. Подцели говорят нам, что если V может указывать на объект кучи H типа T, а M — метод, использованный при вызове N для объекта типа T, то точка вызова S может вызывать метод M.

Второе правило гласит, что если точка S может вызывать метод M, то каждый формальный параметр M может указывать на то, на что указывает соответствующий фактический параметр в данной точке вызова. Правило для обработки возвращаемых значений остается читателю в качестве упражнения.

Объединение этих двух правил с правилами, поясненными в разделе 12.4, создает контекстно-нечувствительный анализ целей указателей, который использует вычисляемый "на лету" граф вызовов. Этот анализ имеет побочный эффект, заключающийся в построении графа вызовов с применением контекстно-нечувствительного и чувствительного к потоку анализов целей указателей. Этот граф вызовов значительно более точен, чем граф, вычисленный на основе только лишь объявлений типов и синтаксического анализа.

12.5.3 Динамическая загрузка и отражение

Языки программирования наподобие Java позволяют выполнять динамическую загрузку классов. Проанализировать все возможные выполнения кода невозможно, а следовательно, невозможно статически обеспечить консервативное приближение графов вызовов или псевдонимов указателей. Статический анализ в состоянии предоставить лишь приближение, основанное на анализе кода. Вспомним, что все описанные здесь анализы могут быть применены на уровне байт-кода Java и, таким образом, не требуют изучения исходного кода. Эта возможность особенно важна в связи с использованием Java-программами множества библиотек.

Даже если считать, что анализируется весь выполнимый код, все равно имеется еще одна сложность, делающая консервативный анализ невозможным: отражение (reflection). Отражение позволяет программе динамически определять типы создаваемых объектов, имена вызываемых методов, а также имена полей, к которым выполняется обращение. Имена типа, метода и поля могут быть вычислены или получены из пользовательского ввода, так что в общем случае единственная возможность приближения — считать возможным все.

Пример 12.25. Приведенный ниже код показывает распространенное применение отражения:

```
1)  String className = ...;
2)  Class c = Class.forName(className);
3)  Object o = c.newInstance();
4)  T t = (T) o;
```

Метод `forName` в библиотеке `Class` принимает строку, содержащую имя класса, и возвращает класс. Метод `newInstance` возвращает экземпляр этого класса. Вместо того чтобы оставить объект o типа `Object`, этот объект приводится к надклассу всех ожидаемых классов T. □

Многие большие приложения Java применяют отражения, при этом обычно используя распространенные идиомы наподобие показанной в примере 12.25. Если приложение не переопределяет загрузчик класса, то сказать, какой класс имеет объект, можно при знании значения `className`. Если значение `className`

определяется в программе, то, поскольку строки в Java неизменяемы, знание того, на что указывает `className`, дает нам имя класса. Этот метод представляет собой еще одно использование анализа целей указателей. Если значение `className` основано на пользовательском вводе, то анализ целей указателей может помочь выяснить, где именно вводится это значение, так что разработчик может ограничить его область видимости.

Аналогично можно воспользоваться инструкцией приведения типа (строка 4 в примере 12.25) для аппроксимации типа динамически создаваемых объектов. В предположении, что обработчик исключений приведения типа не переопределен, объект должен принадлежать подклассу класса T.

12.5.4 Упражнения к разделу 12.5

Упражнение 12.5.1. Для кода на рис. 12.26

а) постройте EDB-отношения *actual*, *formal* и *cha*;

б) сделайте все возможные выводы о фактах *pts* и *hpts*.

! **Упражнение 12.5.2.** Каким образом добавить дополнительные предикаты и правила к EDB-предикатам и правилам из раздела 12.5.2, чтобы учесть тот факт, что если вызов метода возвращает объект, то переменная, которой присваивается результат вызова, может указывать на то, на что указывает переменная, хранящая возвращаемое значение?

12.6 Контекстно-чувствительный анализ указателей

Как говорилось в разделе 12.1.2, контекстная чувствительность может существенно повысить точность межпроцедурного анализа. Мы говорили о двух подходах к межпроцедурному анализу, основанных на клонировании (раздел 12.1.4) и на резюме (раздел 12.1.5). Какой из них мы должны использовать?

При вычислении резюме информации о целях указателей имеется ряд трудностей. Резюме имеют большой размер. Резюме каждого метода должно включать результат действий функции и всех функций, вызываемых ею, на входные параметры. Иначе говоря, метод может изменить множество целей всех данных, достижимых через статические переменные, входные параметры и все объекты, создаваемые методом и вызываемыми им функциями. До сих пор не имеется решения, которое можно было бы масштабировать для больших программ. Даже если резюме сможет быть вычислено при восходящем проходе, то вычисление множеств целей для всего экспоненциального количества контекстов при нисходящем проходе представляет собой еще большую проблему. Такая информация

необходима для глобальных запросов наподобие поиска всех точек в коде, которые обращаются к некоторому объекту.

В этом разделе мы рассмотрим контекстно-чувствительный анализ, основанный на клонировании. Такой анализ просто клонирует методы, по одному для каждого интересующего нас контекста. Затем мы применяем к клонированному графу вызовов контекстно-нечувствительный анализ. Хотя этот подход и кажется простым, но проблема заключается в деталях обработки больших количеств клонов. Сколько всего существует контекстов? Даже если воспользоваться идеей свертки всех рекурсивных циклов, рассматривавшейся в разделе 12.1.3, вряд ли удастся найти все 10^{14} контекстов приложений Java. Представление результатов такого количества контекстов — задача не для слабонервных.

Разделим рассмотрение контекстно-чувствительного анализа на две части.

1. Как логически работать с контекстной чувствительностью? Эта часть проста, так как мы просто применяем контекстно-нечувствительный алгоритм к клонированному графу потока.

2. Как представить экспоненциальное количество контекстов? Один из способов состоит в представлении информации в виде диаграмм бинарного выбора (binary decision diagrams — BDD) — высокооптимизированной структуры данных, используемой во многих других приложениях.

Этот подход к контекстной чувствительности представляет собой прекрасный пример важности абстракции. Как мы покажем, устранение алгоритмической сложности возможно благодаря многим годам работы, вылившимся в BDD-абстракцию. Можно определить контекстно-чувствительный анализ целей указателей всего лишь несколькими строками Datalog, которые, в свою очередь, используют тысячи строк существующего кода для работы с BDD. Этот подход имеет несколько важных преимуществ. Во-первых, он позволяет легко выразить различные дальнейшие анализы, которые используют результаты анализа целей указателей. В конце концов, сам по себе анализ целей указателей мало интересен. Во-вторых, он облегчает написание корректного анализа, позволяя воспользоваться большим количеством готового хорошо отлаженного кода.

12.6.1 Контексты и строки вызовов

В контекстно-чувствительном анализе целей указателей, описанном ниже, предполагается, что граф вызовов уже вычислен. Этот шаг помогает получить компактное представление многих контекстов вызовов. Для получения графа вызовов сначала выполняется рассматривавшийся в разделе 12.5 контекстно-нечувствительный анализ целей указателей, который строит граф вызовов "на лету". Сейчас мы опишем, как создать клонированный граф вызовов.

Контекст является представлением строки вызовов, которая формирует историю активных вызовов функций. Другой взгляд на контекст рассматривает его как резюме последовательности вызовов, записи активации которых находятся в настоящее время в стеке времени выполнения. Если в стеке нет рекурсивных функций, то строка вызовов — последовательность точек, из которых выполнялись вызовы в стеке — является полным представлением. Это приемлемое представление в том смысле, что в нем содержится конечное количество различных контекстов, хотя это количество может экспоненциально зависеть от количества функций в программе.

Однако если в программе имеются рекурсивные функции, то количество возможных строк вызовов становится бесконечным, и мы не можем позволить всем возможным строкам вызовов представлять различные контексты. Существуют разные способы ограничения количества различных контекстов. Например, можно записать регулярное выражение, которое описывает все возможные строки вызовов, и преобразовать это регулярное выражение в детерминированный конечный автомат с применением методов из раздела 3.7. После этого контекст может идентифицироваться состояниями этого автомата.

Здесь мы применим более простую схему, которая собирает историю нерекурсивных вызовов, а рекурсивные вызовы рассматривает как "слишком сложные". Мы начнем с поиска всех множеств взаимно рекурсивных функций в программе. Этот процесс прост и детально рассматриваться здесь не будет. Представим граф, узлами которого являются функции, а наличие ребра от p к q означает, что функция p вызывает функцию q. Сильно связанные компоненты такого графа представляют собой множества взаимно рекурсивных функций. Распространенный частный случай — функция, вызывающая саму себя и не входящая в сильно связанный компонент с другими функциями. Такая функция образует сильно связанный компонент сама по себе. Нерекурсивные функции также образуют сильно связанные компоненты сами по себе. Назовем сильно связанный компонент *нетривиальным*, если он либо состоит более чем из одного члена (случай взаимной рекурсии), либо имеет единственный рекурсивный член. Сильно связанный компонент, состоящий из единственной нерекурсивной функции, будем называть тривиальным сильно связанным компонентом.

Модифицируем правило, что любая строка вызовов представляет собой контекст, следующим образом. Для данной строки вызовов удаляем точку вызова s, если выполняется следующее.

1. s находится в функции p.

2. В точке s вызывается функция q (возможно, что $q = p$).

3. p и q находятся в одном и том же сильно связанном компоненте (т.е. p и q взаимно рекурсивны, или $q = p$ и p — рекурсивная функция).

В результате, когда вызывается член нетривиального сильно связанного компонента S, соответствующая точка вызова становится частью контекста, но вызовы внутри S других функций из того же сильно связанного компонента не являются частью контекста. Наконец, когда из S делается вызов "вовне", эта точка вызова записывается как часть контекста.

Пример 12.26. На рис. 12.28 приведен набросок из пяти функций с некоторыми точками вызова и вызовами друг друга. Изучение вызовов показывает, что q и r являются взаимно рекурсивными, а p, s и t – не рекурсивными. Таким образом, наши контексты представляют собой списки всех точек вызова за исключением s3 и s5, в которых имеют место рекурсивные вызовы между q и r.

```
void p() {
          h: a   = new T;
         s1: T b = q(a);
         s2:       s(b);
     }

T    q(T w) {
         s3: c   = r(w);
          i: T d = new T;
         s4:       t(d);
             return d;
     }

T    r(T x) {
         s5: T e = q(x);
         s6:       s(e);
             return e;
     }

void s(T y) {
         s7: T f = t(y);
         s8:   f = t(f);
     }

T    t(T z) {
          j: T g = new T;
             return g;
     }
```

Рис. 12.28. Функции и точки вызова к примеру 12.26

Рассмотрим все способы достижения p из t, т.е. все контексты, в которых осуществляется вызов t.

1. p может вызвать s в точке s2, а затем s может вызвать t в точке s7 или s8. Таким образом, две возможные строки вызовов — $(s2, s7)$ и $(s2, s8)$.

2. p может вызвать q в точке s1. Затем q и r могут несколько раз рекурсивно вызывать друг друга. Этот цикл вызовов может быть остановлен следующими способами.

   а) В точке s4, где t вызывается непосредственно из q. Это дает нам только один контекст $(s1, s4)$.

   б) В точке s6, где r вызывает s. Здесь мы можем достичь t вызовом либо в точке s7, либо в точке s8. Это дает нам еще два контекста — $(s1, s6, s7)$ и $(s1, s6, s8)$.

Таким образом, всего имеется пять контекстов вызова t. Обратите внимание, что все эти контексты не включают точки рекурсивных вызовов s3 и s5. Например, контекст $(s1, s4)$ на самом деле представляет бесконечное множество строк вызовов $(s1, s3, (s5, s3)^n, s4)$ для всех $n \geqslant 0$. □

Опишем теперь, как строится клонированный граф. Каждый клонированный метод идентифицируется методом M программы и контекстом C. Ребра можно получить путем добавления соответствующих контекстов к каждому из ребер исходного графа вызовов. Вспомним, что если предикат *invokes* (S, M) истинен, то в исходном графе вызовов имеется ребро от точки вызова S к методу M. Чтобы добавить контекст для идентификации методов в клонированном графе вызовов, можно определить соответствующий предикат *CSinvokes*, такой, что *CSinvokes* (S, C, M, D) истинно, если точка вызова S в контексте C вызывает контекст D метода M.

12.6.2 Добавление контекста в правила Datalog

Чтобы найти контекстно-чувствительные отношения указывания, можно просто применить контекстно-нечувствительный анализ целей указателей к клонированному графу вызовов. Поскольку метод в клонированном графе вызовов представлен исходным методом и контекстом, мы должны соответствующим образом изменить правила Datalog. Для простоты приведенные на рис. 12.29 правила не включают ограничений, связанных с типами; символы _ являются любыми новыми переменными.

Дополнительный аргумент, представляющий контекст, должен быть передан IDB-предикату *pts*. *pts* (V, C, H) гласит, что переменная V в контексте C может указывать на объект H. Все правила самоочевидны, за исключением, возможно,

$$
\begin{array}{llll}
1) & pts(V,C,H) & :- & \text{“}H:\ T\ V\ = \texttt{new}\ T()\text{”}\ \& \\
 & & & CSinvokes(H,C,_,_) \\
 & & & \\
2) & pts(V,C,H) & :- & \text{“}V = W\text{”}\ \& \\
 & & & pts(W,C,H) \\
 & & & \\
3) & hpts(H,F,G) & :- & \text{“}V.F = W\text{”}\ \& \\
 & & & pts(W,C,G)\ \& \\
 & & & pts(V,C,H) \\
 & & & \\
4) & pts(V,C,H) & :- & \text{“}V = W.F\text{”}\ \& \\
 & & & pts(W,C,G)\ \& \\
 & & & hpts(G,F,H) \\
 & & & \\
5) & pts(V,D,H) & :- & CSinvokes(S,C,M,D)\ \& \\
 & & & formal(M,I,V)\ \& \\
 & & & actual(S,I,W)\ \& \\
 & & & pts(W,C,H)
\end{array}
$$

Рис. 12.29. Программа Datalog для контекстно-чувствительного анализа целей указателей

правила 5. Это правило гласит, что если точка вызова S в контексте C вызывает метод M контекста D, то формальные параметры в методе M контекста D могут указывать на объекты, на которые указывают соответствующие фактические параметры в контексте C.

12.6.3 Дополнительные наблюдения о чувствительности

Описанное нами — одна из формулировок контекстной чувствительности, которая достаточно практична для обработки больших реальных программ на языке программирования Java, с применением приемов, которые вкратце будут описаны в следующем разделе. Тем не менее данный алгоритм не в состоянии обрабатывать приложения Java очень большого размера.

Объекты кучи в данной формулировке именуются их точками вызовов, но без учета контекста. Такое упрощение может привести к проблемам. Рассмотрим идиому фабрики объектов, в которой все объекты одного и того же типа создаются одной и той же подпрограммой. Рассматриваемая схема приведет к тому, что все

эти объекты будут носить одно и то же имя. Эту ситуацию относительно просто обработать путем встраивания кода создания нового объекта. В общем случае желательно повысить чувствительность к контексту при именовании объектов. Но хотя добавить чувствительность объектов к контексту в Datalog-формулировку несложно, обеспечить масштабируемость данного анализа для работы с большими программами — совершенно другое дело.

Еще одним важным видом чувствительности является чувствительность к объектам. Объектно-чувствительный метод может различать методы разных вызываемых объектов. Рассмотрим сценарий точки вызова в контексте, где переменная, как выясняется, может указывать на два разных вызываемых объекта одного и того же класса. Их поля могут указывать на разные объекты. Если нет возможности различить эти объекты, копирование среди полей объекта, на который ссылается `this`, будет создавать ложные отношения — если только мы не сможем разделить анализ в соответствии с вызываемыми объектами. В некоторых случаях объектная чувствительность оказывается более полезной, чем контекстная.

12.6.4 Упражнения к разделу 12.6

Упражнение 12.6.1. Какие контексты будут найдены при применении методов из данного раздела к коду на рис. 12.30?

! **Упражнение 12.6.2.** Выполните контекстно-чувствительный анализ кода на рис. 12.30.

! **Упражнение 12.6.3.** Следуя подходу из раздела 12.5, измените Datalog-правила из данного раздела так, чтобы они включали информацию о типах и подтипах.

12.7 Реализация Datalog с применением BDD

Диаграммы бинарного выбора (binary decision diagrams — BDD) представляют собой метод представления булевых функций графами. Поскольку всего существует 2^{2^n} булевых функций от n переменных, нет метода представления всех булевых функций, который был бы достаточно лаконичным. Однако булевы функции, встречающиеся на практике, обычно достаточно регулярны. Таким образом, обычно для них удается найти достаточно краткие BDD.

Оказывается, что булевы функции, описываемые Datalog-программами, разработанными для анализа исходных текстов программ, не являются исключением. Хотя краткие BDD-представления информации о программах зачастую требуют применения эвристик и/или методов, используемых в коммерческих BDD-пакетах, на практике BDD-подход зарекомендовал себя как вполне успешный. В частности, он превосходит методы, основанные на обычных системах баз данных, поскольку

```
class T {
    void p() {
          h: T a = new T();
          i: T b = new T();
         c1: T c = a.q(b);
    }

    T    q(T y) {
          j: T d = new T();
         c2:   d = this.q(d);
         c3:   d = d.q(y);
         c4:   d = d.r();
               return d;
    }

    T    r() {
               return this;
    }
}
```

Рис. 12.30. Код к упражнениям 12.6.1 и 12.6.2

последние разработаны для более нерегулярных данных, обычно встречающихся в коммерческих приложениях.

Рассмотрение всей разрабатываемой годами BDD-технологии выходит за пределы нашей книги. Здесь мы только вкратце рассмотрим концепцию BDD: каким образом представить связанные данные в виде BDD и как работать с BDD для отражения операций, выполняющих Datalog-программы при помощи алгоритмов, таких как алгоритм 12.18. Наконец, мы опишем, как представить в BDD экспоненциальное количество контекстов, что является ключом к успешному использованию BDD в контекстно-чувствительном анализе.

12.7.1 Диаграммы бинарного выбора

Диаграммы бинарного выбора представляют булевы функции при помощи ориентированного ациклического графа с корнем. Каждый внутренний узел ориентированного ациклического графа помечен одной из переменных представляемой функции. В нижней части ориентированного ациклического графа находятся два листа, помеченные 0 и 1. Каждый внутренний узел имеет два ребра к дочерним узлам; эти ребра называются верхним (high) и нижним (low). Нижнее ребро связано со случаем, когда переменная в узле имеет значение 0, а верхнее — когда ее значение равно 1.

Чтобы вычислить функцию для заданных значений переменных, мы можем начать с корня и в каждом узле, скажем, в узле x, следовать по нижнему или верхнему ребру в зависимости от значения переменной x. Если в конечном счете мы достигаем листа с меткой 1, то для данного набора значений переменных функция возвращает истинное значение, в противном случае она возвращает ложное значение.

Пример 12.27. На рис. 12.31 приведен пример BDD. Ниже мы познакомимся с функцией, которую представляет это BDD. Все нижние ребра на рисунке имеют метку 0, а верхние – метку 1. Рассмотрим вычисление функции для набора переменных $w = x = y = 0$ и $z = 1$. Начнем с корня. Поскольку $w = 0$, мы выбираем левое ребро, которое приводит нас к левому узлу x. Поскольку $x = 0$, мы вновь выбираем левое ребро, которое приводит нас в левый узел y, где мы также выбираем левое ребро в силу $y = 0$. Это ребро приводит нас в левый узел z, в котором, поскольку $z = 1$, мы выбираем правое ребро, ведущее нас в лист 1. Таким образом, мы делаем заключение, что для данного набора значений переменных функция возвращает значение "истина".

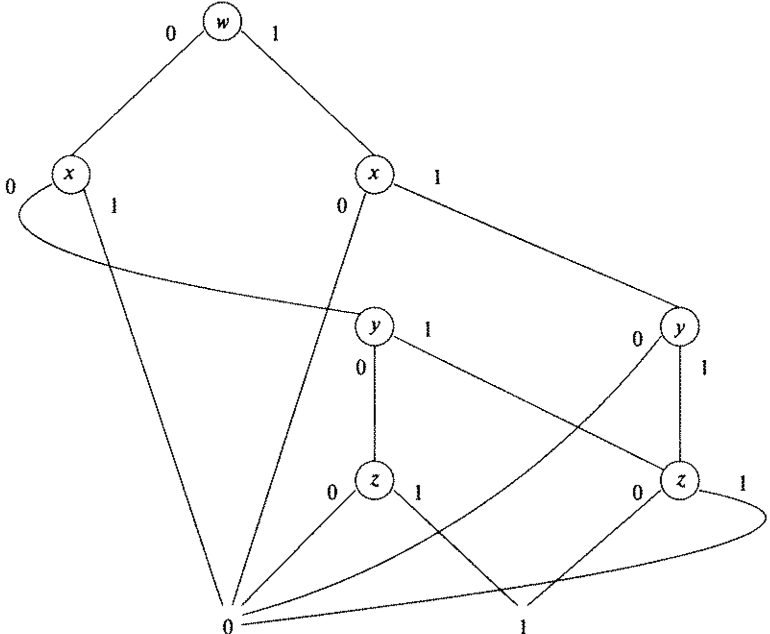


Рис. 12.31. Диаграмма бинарного выбора

Рассмотрим теперь набор значений $wxyz = 0101$, т.е. $w = y = 0$ и $x = z = 1$. Мы снова начинаем с корня BDD. Поскольку $w = 0$, мы опять идем по левому ребру в левый узел x. Но теперь, поскольку $x = 1$, мы выходим из этого узла

по верхнему ребру, которое ведет нас непосредственно в лист 0. Значит, для набора значений 0101 функция возвращает значение "ложь". Но поскольку мы не использовали при этом значений y и z, можно заключить, что функция ложна для всех наборов значений $01yz$. Такое "сокращенное вычисление" — одна из причин, по которой BDD обычно является лаконичным представлением булевых функций. □

На рис. 12.31 узлы распределены по рангам — каждый ранг содержит узлы с меткой определенной переменной. Хотя это не обязательное требование, удобно самоограничиться *упорядоченными диаграммами бинарного выбора*. В упорядоченной BDD существует порядок $x_1, x_2, \ldots, x_n$ переменных, и, если существует ребро от родительского узла с меткой x_i к дочернему узлу с меткой x_j, то $i < j$. Мы увидим, что работать с упорядоченной BDD проще, поэтому далее все рассматриваемые BDD предполагаются упорядоченными.

Заметим также, что диаграммы бинарного выбора являются ориентированными ациклическими графами, но не деревьями. В них не только листья 0 и 1 обычно имеют по нескольку родительских узлов, но и внутренние узлы также могут иметь по нескольку родительских. Например, крайний справа узел z на рис. 12.31 имеет два родительских узла. Это объединение узлов — вторая причина, по которой BDD обычно достаточно кратки.

12.7.2 Преобразования диаграмм бинарного выбора

Выше мы упоминали, что два упрощения BDD могут ее сократить.

1. *Сокращенные вычисления.* Если у узла N и верхнее, и нижнее ребра идут в один и тот же узел M, то узел N можно удалить. Ребра, входящие в N, перенаправляются в узел M.

2. *Слияние узлов.* Если у двух узлов, N и M, нижние ребра идут в один и тот же узел и верхние ребра также идут в один и тот же узел, то узлы N и M можно объединить в один. Ребра, входящие в узлы N и M, теперь идут в объединенный узел.

Эти преобразования могут быть выполнены и в обратном направлении. В частности, в ребро от N к M можно добавить новый узел. Оба ребра из этого нового узла ведут в M, а ребро из N, которое ранее шло в узел M, теперь идет в новый узел. Заметим, однако, что переменная, назначаемая новому узлу, должна быть одной из переменных, лежащих между N и M в порядке, используемом BDD. На рис. 12.32 схематически представлены описанные преобразования.

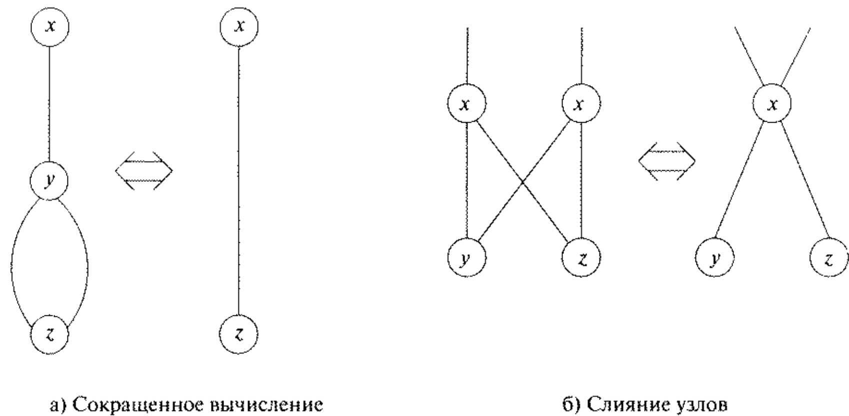


Рис. 12.32. Преобразования BDD

12.7.3 Представление отношений при помощи BDD

Отношения, с которыми мы работаем, имеют компоненты, взятые из "областей определений". Область определения компонента отношения — множество возможных значений, которые в этом компоненте могут принимать кортежи. Например, отношение *pts* (V, H) имеет в качестве области определения первого компонента все переменные программы, а в качестве второго — все инструкции, создающие новые объекты. Если домен имеет более 2^{n-1} возможных значений, но не более 2^n, то для представления значений этого домена требуется n бит или булевых переменных.

Таким образом, кортеж в отношении может рассматриваться как присваивание истинности переменным, представляющим значения из области определения для каждого компонента кортежа. Отношение можно рассматривать как булеву функцию, которая возвращает значение "истинно" для всех тех присваиваний, которые представляют кортежи отношения. Пример 12.28 должен пояснить сказанное.

Пример 12.28. Рассмотрим отношение $r\,(A, B)$, такое, что область определения и A, и B — $\{a, b, c, d\}$. Закодируем a битами 00, b — 01, c — 10 и d — 11. Пусть кортежи отношения r представляют собой

A	B
a	b
a	c
d	c

Воспользуемся булевыми переменными wx для кодирования первого (A) компонента и переменными yz — для кодирования второго (B) компонента. Тогда отношение r превращается в

w	x	y	z
0	0	0	1
0	0	1	0
1	1	1	0

Иначе говоря, отношение r преобразовано в булеву функцию, истинную для трех наборов параметров — $wxyz$ = 0001, 0010 и 1110. Заметим, что именно эти три последовательности битов встречаются в качестве меток на путях от корня к листу 1 на рис. 12.31, т.е. BDD на этом рисунке представляет описанное выше отношение r. □

12.7.4 Операции отношений и BDD-операции

Теперь посмотрим, как представить отношения в виде диаграмм бинарного выбора. Однако для реализации алгоритма наподобие алгоритма 12.18 (инкрементное вычисление Datalog-программ) нам надо уметь работать с BDD так, чтобы наши действия отражали работу с самими отношениями. Вот основные операции, которые мы должны выполнять над отношениями.

1. *Инициализация.* Мы должны создавать BDD, которые представляют единственный кортеж отношения. Мы будем затем связывать их в BDD, представляющие большие отношения при помощи объединения.

2. *Объединение.* Чтобы получить объединение отношений, мы применяем логическое ИЛИ к булевым функциям, представляющим отношения. Эта операция нужна не только для построения начальных отношений, но и для объединения результатов нескольких правил для одного и того же заголовка предиката и для накопления новых фактов в множествах старых фактов, как в случае инкрементного алгоритма 12.18.

3. *Проекция.* При вычислении тела правила мы должны строить заголовок отношения, который вытекает из истинных кортежей тела. В терминах BDD, которая представляет отношение, мы должны удалить узлы, помеченные теми булевыми переменными, которые не представляют компонентов заголовка. Может также потребоваться переименовать переменные в BDD для соответствия булевым переменным компонентов отношения заголовка.

4. *Соединение.* Чтобы найти значения переменных, которые делают тело правила истинным, нам надо "соединить" отношения, соответствующие каждой из подцелей. Предположим, например, что у нас есть две подцели — $r(A, B)$ & $s(B, C)$. Соединение отношений для этих подцелей представляет собой множество троек (a, b, c), таких, что (a, b) является кортежем

в отношении для r, а (b, c) — кортежем в отношении для s. Мы увидим, что после переименования булевых переменных в BDD, такого, что компоненты для двух B согласуются по именам переменных, данная операция над BDD аналогична операции логического И, которая, в свою очередь, похожа на операцию ИЛИ, реализующую объединение.

Диаграммы бинарного выбора для отдельных кортежей

Чтобы инициализировать отношение, нам нужен способ построить BDD для функции, которая истинна для единственного набора значений. Предположим, что у нас есть булевы переменные $x_1, x_2, \ldots, x_n$, имеющие значения $a_1, a_2, \ldots, a_n$, где каждое a_i равно 0 или 1. Для каждого x_i BDD будет иметь один узел N_i. Если $a_i = 0$, то верхнее ребро из N_i ведет в лист 0, а нижнее ведет в N_{i+1} или, если $i = n$, в лист 1. Если $a_i = 1$, то справедливо то же самое, но с заменой верхнего ребра нижним, и наоборот.

Такая стратегия дает нам BDD, которая проверяет, имеет ли каждая переменная x_i $(i = 1, 2, \ldots, n)$ корректное значение. Если обнаруживается некорректное значение, мы переходим непосредственно в лист 0. Лист 1 достигается только в том случае, когда все переменные имеют корректное значение.

В качестве примера взгляните на рис. 12.33, *б*. Показанная здесь BDD представляет функцию, которая истинна тогда и только тогда, когда $x = y = 0$, т.е. ее истинный набор значений — 00.

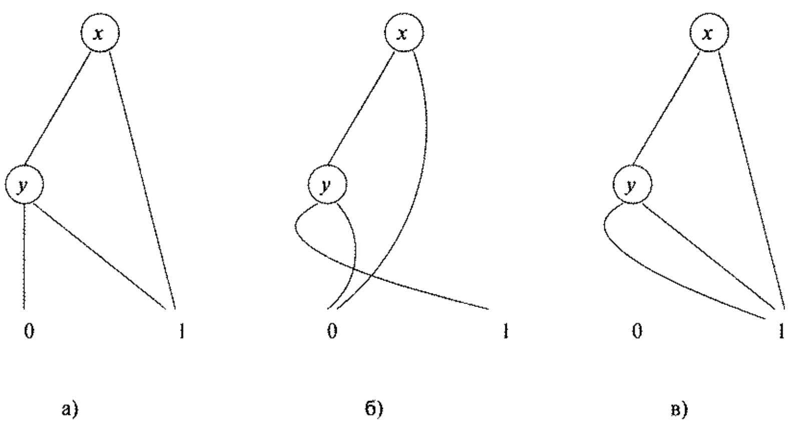


Рис. 12.33. Построение BDD для логического ИЛИ

Объединение

Приведем подробное описание алгоритма, дающего логическое ИЛИ двух BDD, т.е. объединения отношений, представленных этими BDD.

Алгоритм 12.29. Объединение двух BDD

Вход: две упорядоченные BDD с одним и тем же множеством переменных в одном и том же порядке.

Выход: BDD, представляющая функцию, которая является логическим ИЛИ двух булевых функций, представленных входными BDD.

Метод: ниже описана рекурсивная процедура объединения двух BDD. Применяется индукция по размеру множества переменных, участвующих в обеих BDD.

Базис: нуль переменных. Обе BDD представляют собой листья, помеченные 0 либо 1. Выходом является лист 1, если хотя бы один из входов равен 1, и 0, если оба входа равны 0.

Индукция: предположим, что в двух BDD найдено k переменных, $y_1, y_2, \ldots, y_k$. Выполняем следующее.

1. При необходимости используем обращение сокращенного вычисления для добавления нового корня, с тем чтобы оба BDD имели корень с меткой y_1.

2. Пусть корнями являются узлы N и M, их нижними дочерними узлами — узлы N_0 и M_0, а верхними дочерними узлами — N_1 и M_1. Рекурсивно применяем данный алгоритм к диаграммам с корнями N_0 и M_0, а также к диаграммам с корнями N_1 и M_1. Первая из этих диаграмм представляет функцию, которая возвращает значение "истина" для тех истинных наборов данных, у которых $y_1 = 0$, а вторая — для $y_1 = 1$.

3. Создаем новый корневой узел с меткой y_1. Его нижним дочерним узлом является корень первой рекурсивно построенной BDD, а верхним дочерним узлом является корень второй рекурсивно построенной BDD.

4. Выполняем слияние листьев с метками 0 и с метками 1 в построенной объединенной BDD.

5. Применяем, где это возможно, сокращенные вычисления и слияние для упрощения BDD. □

Пример 12.30. На рис. 12.33, *а* и *б* приведены две простые BDD. Первая представляет функцию x OR y, а вторая — функцию NOT x AND NOT y. Заметим, что логическое ИЛИ, примененное к этим функциям, дает функцию 1, т.е. функцию, всегда возвращающую значение "истина". Чтобы применить алгоритм 12.29 к этим диаграммам, рассмотрим нижние и верхние дочерние узлы двух корней, начав с верхних.

Верхний дочерний узел корня на рис. 12.33, *а* — 1, а на рис. 12.33, *б* — 0. Поскольку оба дочерних узла находятся на уровне листьев, мы не должны вставлять узлы с меткой y вдоль каждого ребра, хотя результат при этом получился бы

тот же самый. Базисный случай для объединения 0 и 1 заключается в получении листа с меткой 1, который становится верхним дочерним узлом нового корня.

Нижние дочерние узлы на рис. 12.33, *а* и *б* имеют метки y, так что мы можем вычислить объединение BDD рекурсивно. Нижними дочерними узлами узлов y являются 0 и 1, так что комбинацией их нижних дочерних узлов является лист 1. Аналогично верхними дочерними узлами узлов y являются 0 и 1, так что комбинацией их верхних дочерних узлов также является лист 1. При добавлении нового корня x мы получаем BDD, показанную на рис. 12.33, *в*.

Но это не все, поскольку BDD на рис. 12.33, *в* можно упростить. У узла с меткой y оба дочерних узла представляют собой лист 1, так что узел y можно удалить и сделать лист 1 нижним дочерним узлом корня. Но тогда оба дочерних узла корня оказываются листом 1, так что можно удалить и корень. Таким образом, простейшей BDD объединения является лист 1 сам по себе. □

12.7.5 Использование диаграмм бинарного выбора для анализа целей указателей

Выполнение контекстно-нечувствительного анализа целей указателей — задача нетривиальная. Упорядочение переменных BDD может существенно влиять на размер представления. Чтобы получить упорядочение, позволяющее быстро выполнить анализ, требуется учесть множество факторов, а также воспользоваться методом проб и ошибок.

Еще труднее выполнить контекстно-чувствительный анализ целей указателей, что связано с экспоненциальным количеством контекстов в программе. В частности, при произвольном назначении номеров для представления контекстов в графе вызовов мы не сможем справиться даже с небольшой программой на языке программирования Java. Очень важно пронумеровать контексты таким образом, чтобы бинарное кодирование анализа целей указателей могло быть очень компактным. Два контекста одного и того же метода с похожими путями вызова имеют очень много общего, так что желательно пронумеровать n контекстов метода последовательно. Аналогично, поскольку пары вызываемых и вызывающих методов в одной точке вызова также имеют очень много общего, желательна такая нумерация контекстов, чтобы числовая разность между каждой парой вызываемых и вызывающих методов в точке вызова была константой.

Даже при удовлетворяющей всем условиям схеме контекстов вызовов эффективно проанализировать программу на языке программирования Java все равно сложно. Для получения упорядочения переменных, достаточно эффективного для обработки больших приложений, можно попробовать применить активное машинное обучение.

12.7.6 Упражнения к разделу 12.7

Упражнение 12.7.1. Используя кодирование символов из примера 12.28, разработайте BDD, которая представляет отношение, состоящее из кортежей (b, b), (c, a) и (b, a). Булевы переменные упорядочьте любым способом, который даст вам наиболее краткую BDD.

! **Упражнение 12.7.2.** Выразите количество узлов в наиболее краткой BDD, представляющей функцию ИСКЛЮЧАЮЩЕГО ИЛИ от n переменных в виде функции от n. Такая функция ИСКЛЮЧАЮЩЕГО ИЛИ от n переменных возвращает значение "истинно", если среди n переменных нечетное количество истинных, и "ложно", если таких переменных четное число.

Упражнение 12.7.3. Модифицируйте алгоритм 12.29 так, чтобы он давал нам пересечение (логическое И) двух BDD.

!! **Упражнение 12.7.4.** Разработайте алгоритмы для выполнения следующих операций над отношениями, представленными упорядоченными диаграммами бинарного выбора.

а) Удаление некоторых из булевых переменных. Представляющая функция должна быть истинна для данного набора значений переменных α, если исходная функция истинна для набора значений переменных α и любого значения удаленной переменной.

б) Соединения двух отношений r и s путем объединения кортежа из r с кортежем из s, когда эти кортежи согласуются по общим атрибутам из r и s. Достаточно рассмотреть случай, когда отношения состоят только из двух компонентов, один из которых присутствует в обоих отношениях, т.е. отношения представляют собой $r\,(A, B)$ и $s\,(B, C)$.

12.8 Резюме к главе 12

- *Межпроцедурный анализ.* Анализ потока данных, который отслеживает информацию через границы процедур, называется межпроцедурным анализом. Многие анализы, такие как анализ целей указателей, могут дать полезные результаты, только если являются межпроцедурными.

- *Точки вызова.* Программа вызывает процедуры в определенных точках, известных как точки вызова. Вызываемая процедура может быть очевидна, но может быть и неоднозначна, если вызывается косвенно через указатель или представляет собой вызов виртуального метода, имеющего несколько реализаций.

- *Графы вызовов.* Граф вызова программы представляет собой двудольный граф, узлами которого являются точки вызовов и процедуры. Ребра графа идут от узла точки вызова к узлу процедуры, если данная процедура может быть вызвана в данной точке.

- *Встраивание.* При отсутствии в программе рекурсии, в принципе, можно заменить все вызовы процедур копиями их кода и применить к получившейся программе внутрипроцедурный анализ. Такой анализ по сути является межпроцедурным.

- *Чувствительность к потоку и контекстная чувствительность.* Анализ потока данных называется чувствительным к потоку, если он генерирует факты, зависящие от местоположения в программе. Если эти факты зависят от истории вызовов процедур, то такой анализ называется контекстно-чувствительным. Анализ потока данных может быть чувствителен к потоку, контекстно-чувствителен, может обладать обоими этими свойствами или ни одним из них.

- *Контекстно-чувствительный анализ, основанный на клонировании.* В принципе, установив различные контексты, в которых может быть вызвана процедура, можно представить, что имеется клон каждой процедуры для каждого контекста. Таким образом контекстно-нечувствительный анализ работает в качестве контекстно-чувствительного.

- *Контекстно-чувствительный анализ, основанный на резюме.* Еще один подход к межпроцедурному анализу заключается в распространении ранее описанной методики анализа на основе областей для межпроцедурного анализа. Каждая процедура имеет свою передаточную функцию и в каждом месте вызова процедуры рассматривается как область.

- *Применение межпроцедурного анализа.* Важным приложением, требующим межпроцедурного анализа, является обнаружение уязвимых мест программного обеспечения. Они зачастую характеризуются чтением данных из ненадежного источника одной процедурой с последующим использованием их другой.

- *Datalog.* Язык Datalog предоставляет возможность простой записи правил “если-то”, которые могут использоваться для описания анализа потока данных на высоком уровне. Наборы Datalog-правил, или Datalog-программы, могут вычисляться при помощи одного из нескольких стандартных алгоритмов.

- *Datalog-правила.* Правило Datalog состоит из тела (посылки) и заголовка (следствия). Тело представляет собой один или несколько атомов, а заголовок — ровно один атом. Атомы — это предикаты, применяемые к аргументам, которые могут быть переменными или константами. Атомы тела соединяются при помощи логического И; кроме того, в теле может использоваться отрицание атома.

- *IDB- и EDB-предикаты.* Истинные факты EDB-предикатов в Datalog-программе известны априори. В анализе потоков данных эти предикаты соответствуют фактам, которые могут быть получены из анализируемого кода. IDB-предикаты определяются самими правилами и в анализе потоков данных соответствуют информации, которую мы пытаемся получить из анализируемого кода.

- *Вычисление Datalog-программ.* Мы применяем правила путем подстановки вместо переменных констант, которые делают тело истинным. При этом мы выводим заголовок, который также истинен при данной подстановке. Эта операция повторяется до тех пор, пока не останется невыведенных фактов.

- *Инкрементное вычисление Datalog-программ.* Повышение эффективности достигается за счет инкрементных вычислений. Мы выполняем ряд итераций. В одной итерации мы рассматриваем только те подстановки констант вместо переменных, которые делают как минимум один атом тела фактом, открытым на предыдущей итерации.

- *Анализ указателей Java.* Анализ указателей в Java можно моделировать схемой, в которой существуют ссылочные переменные, указывающие на объекты кучи, которые, в свою очередь, могут иметь поля, указывающие на другие объекты кучи. Нечувствительный анализ указателей можно записать в виде Datalog-программы, которая выводит два вида фактов: что переменная может указывать на объект кучи или что поле объекта кучи может указывать на другой объект кучи.

- *Использование информации о типах.* Более точного анализа указателей можно добиться, если воспользоваться тем фактом, что ссылочные переменные могут указывать только на те объекты кучи, которые принадлежат тому же типу, что и переменная, или его подтипу.

- *Межпроцедурный анализ указателей.* Чтобы сделать анализ межпроцедурным, требуется добавить правила, которые отражают то, как передаются параметры и как возвращаемые значения присваиваются переменным. Эти правила по сути те же, что и правила для копирования одной ссылочной переменной в другую.

- *Построение графа вызовов.* Поскольку в Java имеются виртуальные методы, межпроцедурный анализ требует, в первую очередь, ограничиться процедурами, которые могут быть вызваны в данной точке вызова. Главным способом поиска границ того, что и где может быть вызвано, является анализ типа объектов и использование того факта, что фактический метод, на который ссылается вызов виртуального метода, должен принадлежать соответствующему классу.

- *Контекстно-чувствительный анализ.* В случае рекурсивных процедур мы должны уплотнить информацию, содержащуюся в строках вызова, в конечное количество контекстов. Эффективный способ добиться этого — опустить из строки вызова все точки вызова, где процедура вызывает другую процедуру (возможно, саму себя), взаимно рекурсивную с вызывающей. Используя такое представление, можно модифицировать правила внутрипроцедурного анализа указателей так, чтобы контекст участвовал в предикатах. Этот подход имитирует анализ на основе клонирования.

- *Диаграммы бинарного выбора.* BDD является компактным представлением булевых функций ориентированным ациклическим графом с корнем. Внутренние узлы ориентированного ациклического графа соответствуют булевым переменным и имеют по два дочерних узла — нижний (представляющий значение 0) и верхний (представляющий 1). Имеется также два листа с метками 0 и 1. Набор значений переменных делает представляемую функцию истинной тогда и только тогда, когда путь от корня, на котором мы переходим к нижнему дочернему узлу при равенстве переменной 0, и к верхнему — при 1, приводит нас в лист 1.

- *BDD и отношения.* BDD может служить компактным представлением одного из предикатов Datalog-программы. Константы кодируются наборами значений булевых переменных, а функция, представляемая BDD, истинна тогда и только тогда, когда булевы переменные представляют истинный факт для данного предиката.

- *Реализация анализа потока данных с использованием BDD.* Любой анализ потока данных, который может быть выражен Datalog-правилами, может быть реализован при помощи манипуляций с диаграммами бинарного выбора, представляющими предикаты, включенные в эти правила. Зачастую такое представление приводит к более эффективной реализации анализа потока данных, чем любые другие подходы.

12.9 Список литературы к главе 12

Некоторые базовые концепции межпроцедурного анализа можно найти в [1, 6, 7 и 21]. Каллагэн (Callahan) и другие [11] описывают межпроцедурный алгоритм распространения констант.

Стинсгард (Steensgaard) [22] опубликовал первый масштабируемый анализ псевдонимов указателей, являющийся контекстно-нечувствительным, чувствительным к потоку и основанным на эквивалентности. Контекстно-нечувствительная версия основанного на включении анализа целей указателей была разработана Андерсеном (Andersen) [2]. Позже Хайнце (Heintz) и Тардью (Tardieu) [15] описали эффективный алгоритм этого анализа. Фяндрих (Fahndrich), Рехоф (Rehof) и Дас (Das) [14] представили контекстно-чувствительный, нечувствительный к потоку, основанный на эквивалентности анализ, масштабируемый до больших программ наподобие gcc. Среди попыток создать контекстно-чувствительный анализ целей указателей на основе включения следует отметить работу Эмами (Emami), Гия (Ghiya) и Хендрена (Hendren) [13], разработавших основанный на клонировании и включении контекстно-чувствительный, чувствительный к потоку алгоритм анализа целей указателей.

Диаграммы бинарного выбора впервые появились в работе Брянта (Bryant) [9]. Впервые они были применены для анализа потока данных Берндлом (Berndl) с коллегами в работе [4]. О применении BDD к нечувствительному анализу указателей сообщили Жу (Zhu) [25], Берндл (Berndl) и другие [8]. Вэли (Whaley) и Лам (Lam) [24] описали первый контекстно-чувствительный, нечувствительный к потоку, основанный на включении алгоритм, который, как было показано, применим к реальным приложениям. В статье описывается инструмент под названием bddbddb, который автоматически преобразует анализ, описанный Datalog-программой, в BDD-код. Объектная чувствительность введена Миланова (Milanova), Рунтевом (Rountev) и Райдером (Ryder) [18].

Рассмотрение Datalog можно найти в работе Ульмана (Ullman) и Видома (Widom) [23]. Кроме того, в работе Лама (Lam) с коллегами [16] описывается связь анализа потока данных с Datalog.

Инструментарий проверки кода Metal описан Энглером (Engler) и другими [12], а программа проверки PREfix разработана Бушем (Bush), Пинкусом (Pincus) и Силаффом (Sielaff) [10]. Болл (Ball) и Раджамани (Rajamani) [4] разработали анализатор SLAM, использующий модель проверки и символьного выполнения для имитации возможного поведения системы. На основе SLAM путем применения BDD Болл (Ball) и другие [5] разработали статический анализатор SDV для поиска ошибок использования API в программах драйверов устройств, написанных на C.

Лившиц (Livshits) и Лам (Lam) [17] описали, как можно использовать контекстно-чувствительный анализ целей указателей для поиска уязвимых мест в Web-

приложениях Java, связанных с SQL. Рувейз (Ruwase) и Лам (Lam) [20] описали, как отслеживать расширения массивов и автоматически вставлять динамические проверки границ. Ринард (Rinard) и другие [19] описали, как динамически расширять массивы для предотвращения переполнения. Авоц (Avots) и другие [3] распространили контекстно-чувствительный анализ целей указателей Java на язык программирования C и показали, каким образом его можно использовать для снижения стоимости динамического обнаружения переполнений буферов.

1. Allen, F. E., "Interprocedural data flow analysis", *Proc. IFIP Congress 1974*, pp. 398–402, North Holland, Amsterdam, 1974.

2. Andersen, L., *Program Analysis and Specialization for the C Programming Language*, Ph.D. thesis, DIKU, Univ. of Copenhagen, Denmark, 1994.

3. Avots, D., M. Dalton, V. B. Livshits, and M. S. Lam, "Improving software security with a C pointer analysis", *ICSE 2005: Proc. 27th International Conference on Software Engineering*, pp. 332–341.

4. Ball, T. and S. K. Rajamani, "A symbolic model checker for boolean programs", *Proc. SPIN 2000 Workshop on Model Checking of Software*, pp. 113–130.

5. Ball, T., E. Bounimova, B. Cook, V. Levin, J. Lichtenberg, C. McGarvey, B. Ondrusek, S. Rajamani, and A. Ustuner, "Thorough static analysis of device drivers", *EuroSys* (2006), pp. 73–85.

6. Banning, J. P., "An efficient way to find the side effects of procedural calls and the aliases of variables", *Proc. Sixth Annual Symposium on Principles of Programming Languages* (1979), pp. 29–41.

7. Barth, J. M., "A practical interprocedural data flow analysis algorithm", *Comm. ACM* **21**:9 (1978), pp. 724–736.

8. Berndl, M., O. Lohtak, F. Qian, L. Hendren, and N. Umanee, "Points-to analysis using BDD's", *Proc. ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, pp. 103–114.

9. Bryant, R. E., "Graph-based algorithms for Boolean function manipulation", *IEEE Trans. on Computers* **C-35**:8 (1986), pp. 677–691.

10. Bush, W. R., J. D. Pincus, and D. J. Sielaff, "A static analyzer for finding dynamic programming errors", *Software — Practice and Experience*, **30**:7 (2000), pp. 775–802.

11. Callahan, D., K. D. Cooper, K. Kennedy, and L. Torczon, "Interprocedural constant propagation", *Proc. SIGPLAN 1986 Symposium on Compiler Construction, SIGPLAN Notices*, **21**:7 (1986), pp. 152–161.

12. Engler, D., B. Chelf, A. Chou, and S. Hallem, "Checking system rules using system-specific, programmer-written compiler extensions", *Proc. Sixth USENIX Conference on Operating Systems Design and Implementation* (2000). pp. 1–16.

13. Emami, M., R. Ghiya, and L. J. Hendren, "Context-sensitive interprocedural points-to analysis in the presence of function pointers", *Proc. SIGPLAN Conference on Programming Language Design and Implementation* (1994), pp. 224–256.

14. Fahndrich, M., J. Rehof, and M. Das, "Scalable context-sensitive flow analysis using instantiation constraints", *Proc. SIGPLAN Conference on Programming Language Design and Implementation* (2000), pp. 253–263.

15. Heintze, N. and O. Tardieu, ""Ultra-fast aliasing analysis using CLA: a million lines of C code in a second", *Proc. of the SIGPLAN Conference on Programming Language Design and Implementation* (2001), pp. 254–263.

16. Lam, M. S., J. Whaley, V. B. Livshits, M. C. Martin, D. Avots, M. Carbin, and C. Unkel, "Context-sensitive program analysis as database queries", *Proc. 2005 ACM Symposium on Principles of Database Systems*, pp. 1–12.

17. Livshits, V. B. and M. S. Lam, "Finding security vulnerabilities in Java applications using static analysis", *Proc. 14th USENIX Security Symposium* (2005), pp. 271–286.

18. Milanova, A., A. Rountev, and B. G. Ryder, "Parameterized object sensitivity for points-to and side-effect analyses for Java", *Proc. 2002 ACM SIGSOFT International Symposium on Software Testing and Analysis*, pp. 1–11.

19. Rinard, M., C. Cadar, D. Dumitran, D. Roy, and T. Leu, "A dynamic technique for eliminating buffer overflow vulnerabilities (and other memory errors)", *Proc. 2004 Annual Computer Security Applications Conference*, pp. 82–90.

20. Ruwase, O. and M. S. Lam, "A practical dynamic buffer overflow detector", *Proc. 11th Annual Network and Distributed System Security Symposium* (2004), pp. 159–169.

21. Sharir, M. and A. Pnueli, "Two approaches to interprocedural data flow analysis", in S. Muchnick and N. Jones (eds.) *Program Flow Analysis: Theory and Applications*, Chapter 7, pp. 189–234. Prentice-Hall, Upper Saddle River NJ, 1981.

22. Steensgaard, B., "Points-to analysis in linear time", *Twenty-Third ACM Symposium on Principles of Programming Languages* (1996).

23. Ullman, J. D. and J. Widom, *A First Course in Database Systems*, Prentice-Hall, Upper Saddle River NJ, 2002.

24. Whaley, J. and M. S. Lam, "Cloning-based context-sensitive pointer alias analysis using binary decision diagrams", *Proc. ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation*, pp. 131–144.

25. Zhu, J., "Symbolic Pointer Analysis", *Proc. International Conference in Computer-Aided Design* (2002), pp. 150–157.

Приложение А

Завершенный пример начальной стадии компилятора

В данном приложении представлен завершенный пример начальной стадии компилятора, основанный на простом компиляторе, неформально описанном в разделах 2.5–2.8. Основное отличие от компилятора из главы 2 заключается в том, что приведенная здесь начальная стадия генерирует для булевых выражений безусловные переходы, как описано в разделе 6.6. Мы начнем с синтаксиса исходного языка программирования, описываемого грамматикой, которая должна быть адаптирована для нисходящего синтаксического анализа.

Код транслятора на языке программирования Java состоит из пяти пакетов: `main`, `lexer`, `symbols`, `parser` и `inter`. Пакет `inter` содержит классы для конструкций языка в абстрактном синтаксисе. Поскольку код синтаксического анализатора взаимодействует с остальными пакетами, он будет описан ниже. Каждый пакет хранится в виде отдельного каталога с файлами, по одному для каждого класса.

Исходная программа, поступающая в синтаксический анализатор, представляет собой поток токенов, так что код синтаксического анализатора далек от объектной ориентированности. Выход синтаксического анализатора представляет собой исходную программу в виде синтаксического дерева, причем конструкции и узлы реализованы как объекты. Эти объекты выполняют следующие задачи: строят узлы синтаксического дерева, проверяют типы и генерируют трехадресный промежуточный код (см. пакет `inter`).

А.1 Исходный язык

Программа на исходном языке программирования состоит из блока с необязательными объявлениями и инструкциями. Токен **basic** представляет базовые типы.

Объектно-ориентированный и фазо-ориентированный подходы

В случае объектно-ориентированного подхода весь код конструкции собирается в одном классе, соответствующем этой конструкции. Альтернативным является фазо-ориентированный подход, при котором код группируется по фазам, так что процедура проверки типов может иметь по ветви для каждой конструкции, как и процедура генерации кода и т.д.

Объектно-ориентированный подход упрощает изменение или добавление конструкции, такой, например, как инструкция for; фазо-ориентированный же подход делает проще изменение или добавление фазы, такой, например, как проверка типов. При работе с объектами новая конструкция может быть добавлена путем написания самодостаточного класса, но изменение фазы, такое как, например, добавление кода для преобразования типов, требует внесения изменений во все затрагиваемые классы. В случае фаз добавление новой конструкции может привести к необходимости внесения изменений во все процедуры для данной фазы.

$$
\begin{array}{rcl}
\mathit{program} & \rightarrow & \mathit{block} \\
\mathit{block} & \rightarrow & \{\ \mathit{decls}\ \mathit{stmts}\ \} \\
\mathit{decls} & \rightarrow & \mathit{decls}\ \mathit{decl}\ \mid\ \epsilon \\
\mathit{decl} & \rightarrow & \mathit{type}\ \mathbf{id}\ ; \\
\mathit{type} & \rightarrow & \mathit{type}\ [\ \mathbf{num}\ ]\ \mid\ \mathbf{basic} \\
\mathit{stmts} & \rightarrow & \mathit{stmts}\ \mathit{stmt}\ \mid\ \epsilon
\end{array}
$$

Рассмотрение присваиваний как инструкций, а не как операторов в выражениях упрощает трансляцию.

$$
\begin{array}{rcl}
\mathit{stmt} & \rightarrow & \mathit{loc} = \mathit{bool}\ ; \\
& \mid & \mathbf{if}\ (\ \mathit{bool}\ )\ \mathit{stmt} \\
& \mid & \mathbf{if}\ (\ \mathit{bool}\ )\ \mathit{stmt}\ \mathbf{else}\ \mathit{stmt} \\
& \mid & \mathbf{while}\ (\ \mathit{bool}\ )\ \mathit{stmt} \\
& \mid & \mathbf{do}\ \mathit{stmt}\ \mathbf{while}\ (\ \mathit{bool}\ )\ ; \\
& \mid & \mathbf{break}\ ; \\
& \mid & \mathit{block} \\
\mathit{loc} & \rightarrow & \mathit{loc}\ [\ \mathit{bool}\ ]\ \mid\ \mathbf{id}
\end{array}
$$

Продукции для выражений обрабатываются с учетом ассоциативности и приоритета операторов. Для каждого уровня приоритета используется свой нетер-

минал, а также нетерминал *factor* для выражений в скобках, идентификаторов, обращений к массивам и констант.

$$
\begin{array}{rcl}
bool & \rightarrow & bool \; \texttt{||} \; join \mid join \\
join & \rightarrow & join \; \texttt{\&\&} \; equality \mid equality \\
equality & \rightarrow & equality == rel \mid equality \; != rel \mid rel \\
rel & \rightarrow & expr < expr \mid expr <= expr \mid expr >= expr \mid \\
 & & \quad expr > expr \mid expr \\
expr & \rightarrow & expr + term \mid expr - term \mid term \\
term & \rightarrow & term * unary \mid term \, / \, unary \mid unary \\
unary & \rightarrow & ! \; unary \mid - \, unary \mid factor \\
factor & \rightarrow & (\; bool \;) \mid loc \mid \textbf{num} \mid \textbf{real} \mid \textbf{true} \mid \textbf{false}
\end{array}
$$

A.2 Main

Выполнение начинается с метода `main` в классе `Main`. Метод `main` создает лексический анализатор и синтаксический анализатор, а затем вызывает метод `program` синтаксического анализатора.

```
 1) package main;                      // Файл Main.java
 2) import java.io.*; import lexer.*; import parser.*;
 3) public class Main {
 4)    public static void main(String[] args) throws IOException {
 5)       Lexer lex = new Lexer();
 6)       Parser parse = new Parser(lex);
 7)       parse.program();
 8)       System.out.write('\n');
 9)    }
10) }
```

A.3 Лексический анализатор

Пакет `lexer` представляет собой расширение кода лексического анализатора, представленного в разделе 2.6.5. Класс `Tag` определяет константы для токенов.

```
1) package lexer;                      // Файл Tag.java
2) public class Tag {
3)    public final static int
4)       AND   = 256, BASIC = 257, BREAK = 258, DO    = 259,
5)       ELSE  = 260, EQ    = 261, FALSE = 262, GE    = 263,
6)       ID    = 264, IF    = 265, INDEX = 266, LE    = 267,
7)       MINUS = 268, NE    = 269, NUM   = 270, OR    = 271,
8)       REAL  = 272, TEMP  = 273, TRUE  = 274, WHILE = 275;
9) }
```

Три константы — INDEX, MINUS и TEMP — не являются лексическими токенами; они будут использованы в синтаксических деревьях.

Классы Token и Num те же, что и в разделе 2.6.5, но с добавлением метода toString.

```
1) package lexer;                    // Файл Token.java
2) public class Token {
3)    public final int tag;
4)    public Token(int t) { tag = t; }
5)    public String toString() {return "" + (char)tag;}
6) }
```

```
1) package lexer;                    // Файл Num.java
2) public class Num extends Token {
3)    public final int value;
4)    public Num(int v) { super(Tag.NUM); value = v; }
5)    public String toString() { return "" + value; }
6) }
```

Класс Word работает с лексемами для зарезервированных слов, идентификаторов и составных токенов наподобие &&. Он также используется для работы с прописными формами операторов в промежуточном коде — наподобие унарного минуса; например, исходный текст -2 имеет промежуточный вид minus 2.

```
 1) package lexer;                    // Файл Word.java
 2) public class Word extends Token {
 3)    public String lexeme = "";
 4)    public Word(String s, int tag) { super(tag); lexeme = s; }
 5)    public String toString() { return lexeme; }
 6)    public static final Word
 7)       and    = new Word( "&&",    Tag.AND ),
 8)       or     = new Word( "||",    Tag.OR  ),
 9)       eq     = new Word( "==",    Tag.EQ  ),
10)       ne     = new Word( "!=",    Tag.NE  ),
11)       le     = new Word( "<=",    Tag.LE  ),
12)       ge     = new Word( ">=",    Tag.GE  ),
13)       minus  = new Word( "minus", Tag.MINUS ),
14)       True   = new Word( "true",  Tag.TRUE  ),
15)       False  = new Word( "false", Tag.FALSE ),
16)       temp   = new Word( "t",     Tag.TEMP  );
17) }
```

Класс Real предназначен для работы с числами с плавающей точкой.

```
1) package lexer;                    // Файл Real.java
2) public class Real extends Token {
3)    public final float value;
4)    public Real(float v) { super(Tag.REAL); value = v; }
5)    public String toString() { return "" + value; }
6) }
```

Основной метод класса `Lexer` – функция `scan` – распознает числа, идентификаторы и зарезервированные слова (см. раздел 2.6.5).

В строках 9–13 класса `Lexer` резервируются выбранные ключевые слова. В строках 14–16 резервируются лексемы для объектов, определенных в других местах. Объекты `Word.True` и `Word.False` определены в классе `Word`; объекты базовых типов `int`, `char`, `bool` и `float` определены в классе `Type`, подклассе `Word`. Класс `Type` находится в пакете `symbols`.

```
 1) package lexer;                       // Файл Lexer.java
 2) import java.io.*; import java.util.*; import symbols.*;
 3) public class Lexer {
 4)    public static int line = 1;
 5)    char peek = ' ';
 6)    Hashtable words = new Hashtable();
 7)    void reserve(Word w) { words.put(w.lexeme, w); }
 8)    public Lexer() {
 9)       reserve( new Word("if",    Tag.IF)    );
10)       reserve( new Word("else",  Tag.ELSE)  );
11)       reserve( new Word("while", Tag.WHILE) );
12)       reserve( new Word("do",    Tag.DO)    );
13)       reserve( new Word("break", Tag.BREAK) );
14)       reserve( Word.True );  reserve( Word.False );
15)       reserve( Type.Int  );  reserve( Type.Char  );
16)       reserve( Type.Bool );  reserve( Type.Float );
17)    }
```

Функция `readch()` (строка 18) используется для чтения очередного входного символа в переменную `peek`. Имя `readch` повторно используется или перегружается (строки 19–25) для распознавания составных токенов. Например, если во входном потоке встречается символ <, вызов `readch('=')` считывает очередной символ в переменную `peek` и проверяет, является ли он символом =.

```
18)    void readch() throws IOException {
19)       peek = (char)System.in.read(); }
20)    boolean readch(char c) throws IOException {
21)       readch();
22)       if( peek != c ) return false;
23)       peek = ' ';
24)       return true;
25)    }
```

Функция `scan` начинается с пропускания всех пробельных символов (строки 27–31). Она распознает составные токены наподобие <= (строки 32–51) и числа наподобие `365` и `3.14` (строки 52–65), после чего переходит к сбору слов (строки 66–77).

```
26)    public Token scan() throws IOException {
27)       for( ; ; readch() ) {
```

```
28)         if( peek == ' ' || peek == '\t' ) continue;
29)         else if( peek == '\n' ) line = line + 1;
30)         else break;
31)      }
32)      switch( peek ) {
33)      case '&':
34)         if( readch('&') ) return Word.and;
35)            else return new Token('&');
36)      case '|':
37)         if( readch('|') ) return Word.or;
38)            else return new Token('|');
39)      case '=':
40)         if( readch('=') ) return Word.eq;
41)            else return new Token('=');
42)      case '!':
43)         if( readch('=') ) return Word.ne;
44)            else return new Token('!');
45)      case '<':
46)         if( readch('=') ) return Word.le;
47)            else return new Token('<');
48)      case '>':
49)         if( readch('=') ) return Word.ge;
50)            else return new Token('>');
51)      }
52)      if( Character.isDigit(peek) ) {
53)         int v = 0;
54)         do {
55)            v = 10*v + Character.digit(peek, 10); readch();
56)         } while( Character.isDigit(peek) );
57)         if( peek != '.' ) return new Num(v);
58)         float x = v; float d = 10;
59)         for(;;) {
60)            readch();
61)            if( ! Character.isDigit(peek) ) break;
62)            x = x + Character.digit(peek, 10) / d; d = d*10;
63)         }
64)         return new Real(x);
65)      }
66)      if( Character.isLetter(peek) ) {
67)         StringBuffer b = new StringBuffer();
68)         do {
69)            b.append(peek); readch();
70)         } while( Character.isLetterOrDigit(peek) );
71)         String s = b.toString();
72)         Word w = (Word)words.get(s);
73)         if( w != null ) return w;
74)         w = new Word(s, Tag.ID);
75)         words.put(s, w);
76)         return w;
77)      }
```

Наконец, все остальные символы возвращаются как токены (строки 78–81).

```
78)         Token tok = new Token(peek); peek = ' ';
79)         return tok;
80)     }
81) }
```

А.4 Таблицы символов и типы

Пакет symbols реализует таблицы символов и типы.

Класс Env по сути остался неизменным, таким как на рис. 2.37. В то время как класс Lexer отображает строки на слова, класс Env отображает токены на объекты класса Id, определенного в пакете inter вместе с классами для выражений и инструкций.

```
 1) package symbols;                    // Файл Env.java
 2) import java.util.*; import lexer.*; import inter.*;
 3) public class Env {
 4)    private Hashtable table;
 5)    protected Env prev;
 6)    public Env(Env n) { table = new Hashtable(); prev = n; }
 7)    public void put(Token w, Id i) { table.put(w, i); }
 8)    public Id get(Token w) {
 9)       for( Env e = this; e != null; e = e.prev ) {
10)          Id found = (Id)(e.table.get(w));
11)          if( found != null ) return found;
12)       }
13)       return null;
14)    }
15) }
```

Класс Type определен как подкласс Word, поскольку имена фундаментальных типов наподобие int являются зарезервированными словами, лексемы которых отображаются лексическим анализатором на соответствующие объекты. Объектами для фундаментальных типов являются Type.Int, Type.Float, Type.Char и Type.Bool (строки 8–11). У всех них наследованное поле tag устанавливается равным Tag.BASIC, так что синтаксический анализатор обрабатывает их одинаково.

```
 1) package symbols;                    // Файл Type.java
 2) import lexer.*;
 3) public class Type extends Word {
 4)    public int width = 0;       // Используется при выделении памяти
 5)    public Type(String s, int tag, int w)
 6)       { super(s, tag); width = w; }
 7)    public static final Type
 8)       Int   = new Type( "int",   Tag.BASIC, 4 ),
```

```
 9)         Float = new Type( "float", Tag.BASIC, 8 ),
10)         Char  = new Type( "char",  Tag.BASIC, 1 ),
11)         Bool  = new Type( "bool",  Tag.BASIC, 1 );
```

Функции `numeric` (строки 12–16) и `max` (строки 17–25) используются при преобразовании типов.

```
12)     public static boolean numeric(Type p) {
13)        if (p == Type.Char || p == Type.Int ||
14)            p == Type.Float) return true;
15)        else return false;
16)     }
17)     public static Type max(Type p1, Type p2 ) {
18)        if ( ! numeric(p1) || ! numeric(p2) ) return null;
19)        else if ( p1 == Type.Float || p2 == Type.Float )
20)            return Type.Float;
21)        else if ( p1 == Type.Int   || p2 == Type.Int   )
22)            return Type.Int;
23)        else return Type.Char;
24)     }
25) }
```

Преобразования разрешены между "числовыми" типами `Type.Char`, `Type.Int` и `Type.Float`. При применении арифметического оператора к двум числовым типам результат имеет "максимальный" из этих двух типов.

Массивы – единственный конструируемый тип в исходном языке программирования. Вызов функции `super` в строке 7 устанавливает поле `width`, которое очень важно при вычислении адресов. Он также устанавливает `lexeme` и `tok` равными не используемым значениям по умолчанию.

```
 1) package symbols;                     // Файл Array.java
 2) import lexer.*;
 3) public class Array extends Type {
 4)     public Type of;                       // Массив типа *of*
 5)     public int size = 1;                  // Количество элементов
 6)     public Array(int sz, Type p) {
 7)        super("[]", Tag.INDEX, sz*p.width);
 8)        size = sz;  of = p;
 9)     }
10)     public String toString() { return "[" + size +
11)                                "] " + of.toString(); }
12) }
```

А.5 Промежуточный код выражений

Пакет `inter` содержит иерархию классов `Node`. Этот класс имеет два подкласса: `Expr` для узлов, представляющих выражения, и `Stmt` для узлов, представляющих инструкции. В этом разделе мы рассмотрим класс `Expr` и его подклассы.

Некоторые из методов `Expr` предназначены для работы с условными выражениями и безусловными переходами; они будут рассмотрены в разделе А.6 вместе с остальными подклассами `Expr`.

Узлы в синтаксическом дереве реализованы как объекты класса `Node`. Для сообщения об ошибках в поле `lexline` (строка 4 в файле *Node.java*) сохраняется номер строки в исходном тексте, соответствующий данной конструкции. Строки 8–13 предназначены для вывода трехадресного кода.

```
 1) package inter;                    // Файл Node.java
 2) import lexer.*;
 3) public class Node {
 4)    int lexline = 0;
 5)    Node() { lexline = Lexer.line; }
 6)    void error(String s) {
 7)       throw new Error("near line "+lexline+": "+s); }
 8)    static int labels = 0;
 9)    public int newlabel() { return ++labels; }
10)    public void emitlabel(int i) {
11)       System.out.print("L" + i + ":"); }
12)    public void emit(String s) {
13)       System.out.println("\t" + s); }
14) }
```

Конструкции выражений реализованы подклассами `Expr`. Класс `Expr` имеет поля `op` и `type` (строки 4 и 5 файла *Expr.java*), которые представляют в узле соответственно оператор и тип.

```
1) package inter;                    // Файл Expr.java
2) import lexer.*; import symbols.*;
3) public class Expr extends Node {
4)    public Token op;
5)    public Type type;
6)    Expr(Token tok, Type p) { op = tok; type = p; }
```

Метод `gen` (строка 7) возвращает "член", который может находиться в правой части трехадресной команды. Для выражения $E = E_1 + E_2$ метод `gen` возвращает член $x_1 + x_2$, где x_1 и x_2 — адреса значений E_1 и E_2 соответственно. Возвращаемое значение `this` корректно, если этот объект является адресом; подклассы `Expr` обычно заново реализуют метод `gen`.

Метод `reduce` (строка 8) вычисляет, или "сворачивает", выражение в единственный адрес; т.е. он возвращает константу, идентификатор или временное имя. Для данного выражения E метод `reduce` возвращает временную переменную t, хранящую значение E. Здесь также возвращаемое значение `this` корректно, если этот объект является адресом.

Рассмотрение методов `jumping` и `emitjumps` (строки 9–21) мы отложим до раздела А.6. Они предназначены для генерации кода логических выражений.

```
 7)    public Expr gen() { return this; }
 8)    public Expr reduce() { return this; }
 9)    public void jumping(int t, int f) {
10)       emitjumps(toString(), t, f); }
11)    public void emitjumps(String test, int t, int f) {
12)       if( t != 0 && f != 0 ) {
13)          emit("if " + test + " goto L" + t);
14)          emit("goto L" + f);
15)       }
16)       else if( t != 0 )
17)          emit("if " + test + " goto L" + t);
18)       else if( f != 0 )
19)          emit("iffalse " + test + " goto L" + f);
20)       else ; // nothing since both t and f fall through
21)    }
22)    public String toString() { return op.toString(); }
23) }
```

Класс `Id` наследует реализацию по умолчанию методов `gen` и `reduce` из класса `Expr`, поскольку идентификатор представляет собой адрес.

```
1) package inter;                    // Файл Id.java
2) import lexer.*; import symbols.*;
3) public class Id extends Expr {
4)    public int offset;     // Относительный адрес
5)    public Id(Word id, Type p, int b) {
6)       super(id, p); offset = b; }
7) }
```

Узел для идентификатора класса `Id` является листом. Вызов `super(id,p)` (строка 6 файла *Id.java*) сохраняет `id` и `p` в наследованных полях `op` и `type` соответственно. Поле `offset` (строка 4) хранит относительный адрес идентификатора.

Класс `Op` реализует метод `reduce` (строки 5–10 файла *Op.java*), который наследуется подклассами `Arith` для арифметических операций, `Unary` – для унарных и `Access` – для обращения к массивам. В каждом случае `reduce` вызывает `gen` для генерации члена, выводит команду присваивания члена новой временной переменной и возвращает эту временную переменную.

```
 1) package inter;                    // Файл Op.java
 2) import lexer.*; import symbols.*;
 3) public class Op extends Expr {
 4)    public Op(Token tok, Type p)  { super(tok, p); }
 5)    public Expr reduce() {
 6)       Expr x = gen();
 7)       Temp t = new Temp(type);
 8)       emit( t.toString() + " = " + x.toString() );
 9)       return t;
10)    }
11) }
```

Класс Arith реализует бинарные операторы наподобие + и *. Конструктор Arith начинается с вызова super(tok,null) (строка 6), где tok — токен, представляющий оператор, а null — заполняющее значение для типа. Тип определяется в строке 7 путем использования метода Type.max, который определяет, могут ли два операнда быть приведены к одному общему числовому типу; код Type.max приведен в разделе А.4. Если приведение возможно, type устанавливается равным результирующему типу; в противном случае выводится сообщение об ошибке типа (строка 8). Наш простой компилятор проверяет типы, но не вносит операции преобразования типа.

```
 1) package inter;                    // Файл Arith.java
 2) import lexer.*; import symbols.*;
 3) public class Arith extends Op {
 4)    public Expr expr1, expr2;
 5)    public Arith(Token tok, Expr x1, Expr x2)  {
 6)       super(tok, null); expr1 = x1; expr2 = x2;
 7)       type = Type.max(expr1.type, expr2.type);
 8)       if (type == null ) error("type error");
 9)    }
10)    public Expr gen() {
11)       return new Arith(op, expr1.reduce(),
12)                             expr2.reduce());
13)    }
14)    public String toString() {
15)       return expr1.toString()+" "+op.toString()+
16)              " "+expr2.toString();
17)    }
18) }
```

Метод gen конструирует правую часть трехадресной команды путем свертки подвыражений в адреса и применения к ним оператора (строки 11 и 12 файла *Arith.java*). Предположим, например, что gen вызывается в корне для a+b*c. Вызовы reduce вернут a как адрес подвыражения a и t как адрес b*c. Одновременно reduce генерирует команду t=b*c. Метод gen возвращает новый узел Arith с оператором * и адресами a и t в качестве операндов.[1]

Стоит заметить, что временные переменные типизированы, как и все выражения. Следовательно, конструктор Temp вызывается с типом в качестве параметра (строка 6, файл *Temp.java*).[2]

[1] Для сообщения об ошибке поле lexline класса Node хранит номер текущей лексической строки при построении узла. Отслеживание номеров строк при построении узлов в процессе генерации промежуточного кода остается читателю в качестве упражнения.

[2] Другой подход заключается в передаче в качестве параметра конструктора узла выражения, чтобы конструктор мог скопировать тип и лексическое положение узла выражения.

```
1) package inter;                  // Файл Temp.java
2) import lexer.*; import symbols.*;
3) public class Temp extends Expr {
4)    static int count = 0;
5)    int number = 0;
6)    public Temp(Type p) {
7)       super(Word.temp, p); number = ++count; }
8)    public String toString() { return "t" + number; }
9) }
```

Класс Unary — однооперандный аналог класса Arith.

```
1) package inter;                  // Файл Unary.java
2) import lexer.*; import symbols.*;
3) public class Unary extends Op {
4)    public Expr expr;
5)    public Unary(Token tok, Expr x) {    // Обрабатывает
6)                              // унарный минус; обработку
7)                              //оператора ! см. в классе Not
8)       super(tok, null);  expr = x;
9)       type = Type.max(Type.Int, expr.type);
10)      if (type == null ) error("type error");
11)   }
12)   public Expr gen() {
13)      return new Unary(op, expr.reduce()); }
14)   public String toString() {
15)      return op.toString()+" "+expr.toString(); }
16) }
```

А.6 Переходы для булевых выражений

Код переходов для булева выражения B генерируется методом jumping, который получает в качестве параметров две метки, t и f, именуемые соответственно истинным и ложным выходами B. Код содержит переход к t, если вычисление B дает значение "истина", и к f, если вычислено значение "ложь". По соглашению специальная метка 0 означает, что управление передается следующей команде после кода B.

Начнем с класса Constant. Конструктор Constant в строке 4 получает в качестве параметра токен tok и тип p. Он строит лист синтаксического дерева с меткой tok и типом p. Для удобства конструктор Constant перегружается (строка 5) для создания константного объекта из целого числа.

```
1) package inter;                  // Файл Constant.java
2) import lexer.*; import symbols.*;
3) public class Constant extends Expr {
4)    public Constant(Token tok, Type p) { super(tok, p); }
5)    public Constant(int i) { super(new Num(i), Type.Int); }
```

```
 6)    public static final Constant
 7)       True  = new Constant(Word.True,  Type.Bool),
 8)       False = new Constant(Word.False, Type.Bool);
 9)    public void jumping(int t, int f) {
10)       if ( this == True && t != 0 )
11)          emit("goto L" + t);
12)       else if ( this == False && f != 0)
13)          emit("goto L" + f);
14)    }
15) }
```

Метод `jumping` (строки 9–14, файл *Constant.java*) получает два параметра, метки `t` и `f`. Если эта константа является статическим объектом `True` (определенным в строке 7), а `t` не является специальной меткой `0`, то генерируется переход к метке `t`. В противном случае, если это объект `False` (определенный в строке 8), а `f` не равно `0`, генерируется переход к метке `f`.

Класс `Logical` предоставляет некоторую общую функциональность для классов `Or`, `And` и `Not`. Поля `expr1` и `expr2` (строка 4) соответствуют операндам логического оператора. (Хотя класс `Not` и реализует унарный оператор, для удобства он сделан подклассом класса `Logical`.) Конструктор `Logical(tok,a,b)` (строки 5–10) строит синтаксический узел с оператором `tok` и операндами `a` и `b`. При этом для проверки, что и `a`, и `b` имеют логический тип, используется функция `check`. Метод `gen` будет рассмотрен в конце этого раздела.

```
 1) package inter;                    // Файл Logical.java
 2) import lexer.*; import symbols.*;
 3) public class Logical extends Expr {
 4)    public Expr expr1, expr2;
 5)    Logical(Token tok, Expr x1, Expr x2) {
 6)       super(tok, null);                  // Для начала нулевой тип
 7)       expr1 = x1; expr2 = x2;
 8)       type = check(expr1.type, expr2.type);
 9)       if (type == null ) error("type error");
10)    }
11)    public Type check(Type p1, Type p2) {
12)       if ( p1 == Type.Bool && p2 == Type.Bool )
13)          return Type.Bool;
14)       else return null;
15)    }
16)    public Expr gen() {
17)       int f = newlabel(); int a = newlabel();
18)       Temp temp = new Temp(type);
19)       this.jumping(0,f);
20)       emit(temp.toString() + " = true");
21)       emit("goto L" + a);
22)       emitlabel(f); emit(temp.toString() + " = false");
23)       emitlabel(a);
24)       return temp;
```

```
25)     }
26)     public String toString() {
27)        return expr1.toString()+" "+op.toString()
28)               +" "+expr2.toString();
29)     }
30) }
```

В классе `Or` метод `jumping` (строки 6–11) генерирует код перехода для булева выражения $B = B_1 \| B_2$. Предположим на минуту, что ни истинный выход `t`, ни ложный выход `f` выражения B не является специальной меткой 0. Поскольку B истинно, если истинно B_1, истинный выход B_1 должен быть `t`, а ложный выход соответствует первой команде B_2. Истинный и ложный выходы B_2 те же, что и для B в целом.

```
 1) package inter;                    // Файл Or.java
 2) import lexer.*; import symbols.*;
 3) public class Or extends Logical {
 4)    public Or(Token tok, Expr x1, Expr x2) {
 5)       super(tok, x1, x2); }
 6)    public void jumping(int t, int f) {
 7)       int label = t != 0 ? t : newlabel();
 8)       expr1.jumping(label, 0);
 9)       expr2.jumping(t,f);
10)      if( t == 0 ) emitlabel(label);
11)    }
12) }
```

В общем случае `t`, истинный выход B, может быть специальной меткой 0. Переменная `label` (строка 7, файл *Or.java*) гарантирует корректную установку истинного выхода B_1 на конец кода B. Если `t` равно 0, то `label` устанавливается равной новой метке, которая выводится после генерации кода для B_1 и B_2.

Код класса `And` подобен коду класса `Or`.

```
 1) package inter;                    // Файл And.java
 2) import lexer.*; import symbols.*;
 3) public class And extends Logical {
 4)    public And(Token tok, Expr x1, Expr x2) {
 5)       super(tok, x1, x2); }
 6)    public void jumping(int t, int f) {
 7)       int label = f != 0 ? f : newlabel();
 8)       expr1.jumping(0, label);
 9)       expr2.jumping(t,f);
10)       if( f == 0 ) emitlabel(label);
11)    }
12) }
```

Класс `Not` имеет достаточно много общего с другими логическими операторами, так что мы делаем его подклассом `Logical`, несмотря на то что `Not`

реализует унарный оператор. Надкласс ожидает два операнда, так что при вызове `super` в строке 5 ему передаются два `x2`. В методах в строках 6–9 используется только переменная `expr2` (объявленная в строке 4 файла *Logical.java*). В строке 7 метод `jumping` просто вызывает `expr2.jumping` с обращенными истинным и ложным выходами.

```
 1) package inter;                     // Файл Not.java
 2) import lexer.*; import symbols.*;
 3) public class Not extends Logical {
 4)    public Not(Token tok, Expr x2) {
 5)       super(tok, x2, x2); }
 6)    public void jumping(int t, int f) {
 7)       expr2.jumping(f, t); }
 8)    public String toString() {
 9)       return op.toString()+" "+expr2.toString(); }
10) }
```

Класс `Rel` реализует операторы <, <=, ==, !=, >= и >. Функция `check` (строки 6–11) проверяет, что два операнда имеют один и тот же тип и что они не являются массивами. Для простоты приведение типов не допускается.

```
 1) package inter;                     // Файл Rel.java
 2) import lexer.*; import symbols.*;
 3) public class Rel extends Logical {
 4)    public Rel(Token tok, Expr x1, Expr x2) {
 5)   super(tok, x1, x2); }
 6)    public Type check(Type p1, Type p2) {
 7)       if ( p1 instanceof Array || p2 instanceof Array )
 8)           return null;
 9)       else if( p1 == p2 ) return Type.Bool;
10)       else return null;
11)    }
12)    public void jumping(int t, int f) {
13)       Expr a = expr1.reduce();
14)       Expr b = expr2.reduce();
15)       String test = a.toString() + " " + op.toString()
16)          + " " + b.toString();
17)       emitjumps(test, t, f);
18)    }
19) }
```

Метод `jumping` (строки 12–18, файл *Rel.java*) начинается с генерации кода для подвыражений `expr1` и `expr2` (строки 13 и 14). Затем вызывается метод `emitjump`, определенный в строках 11–21 файла *Expr.java* в разделе А.5. Если ни `t`, ни `f` не является специальной меткой 0, метод `emitjumps` выполняет следующее:

```
13)        emit("if " + test + " goto L" + t);           // Файл Expr.java
14)        emit("goto L" + f);
```

Если либо t, либо f является специальной меткой 0, то генерируется не более одной команды:

```
16)     else if( t != 0 )              // Файл Expr.java
17)        emit("if " + test + " goto L" + t);
18)     else if( f != 0 )
19)        emit("iffalse " + test + " goto L" + f);
20)     else ; // Ничего: и t, и f неуспешны
```

Рассмотрим еще одно применение emitjumps в коде класса Access. Исходный язык позволяет присваивать булевы значения идентификаторам и элементам массива, так что булево выражение может представлять собой обращение к массиву. Класс Access имеет метод gen для генерации "нормального" кода и метод jumping для генерации кода переходов. Метод jumping (строки 13 и 14) вызывает emitjumps после свертки данного обращения к массиву во временную переменную. Конструктор (строки 6–10) вызывается с передачей выровненного массива a, индекса i и типа p элемента в выровненном массиве. Проверка типа выполняется в процессе вычисления адреса массива.

```
 1) package inter;                    // Файл Access.java
 2) import lexer.*; import symbols.*;
 3) public class Access extends Op {
 4)    public Id array;
 5)    public Expr index;
 6)    public Access(Id a, Expr i, Type p) {
 7)       // p - тип элемента после выравнивания массива
 8)       super(new Word("[]", Tag.INDEX), p);
 9)       array = a; index = i;
10)    }
11)    public Expr gen() {
12)       return new Access(array, index.reduce(), type); }
13)    public void jumping(int t,int f) {
14)       emitjumps(reduce().toString(),t,f); }
15)    public String toString() {
16)       return array.toString() + " [ " +
17)          index.toString() + " ]";
18)    }
19) }
```

Код с переходами может использоваться и для возврата булева значения. Класс Logical, описанный в этом разделе выше, имеет метод gen (строки 16–25), который возвращает временную переменную temp, значение которой определяется потоком управления, проходящим по коду с переходами, сгенерированному для этого выражения. На истинном выходе этого булева выражения переменной temp присваивается значение true; на ложном выходе – значение false. Временная переменная объявлена в строке 18. Код переходов для выражения генерируется в строке 19; истинный выход при этом представляет собой следующую команду,

а ложным выходом является новая метка f. Следующая команда присваивает переменной temp значение true (строка 20), после чего следует переход к новой метке a (строка 21). Код в строке 22 генерирует метку f и команду, которая присваивает переменной temp значение false. Фрагмент кода завершается меткой a, генерируемой в строке 23. Наконец, метод gen возвращает temp (строка 24).

А.7 Промежуточный код для инструкций

Каждая конструкция реализуется подклассом класса Stmt. Поля для компонентов конструкции находятся в соответствующих подклассах; например, класс While содержит поля для проверяемого выражения и подынструкции.

В строках 3 и 4 приведенного далее кода класса Stmt выполняется работа по построению синтаксического дерева. Конструктор Stmt() не делает ничего, поскольку вся работа выполняется подклассами. Статический объект Stmt.Null (строка 4) представляет пустую последовательность инструкций.

```
 1) package inter;                    // Файл Stmt.java
 2) public class Stmt extends Node {
 3)    public Stmt() { }
 4)    public static Stmt Null = new Stmt();
 5)    // Вызывается с метками начала и после конструкции:
 6)    public void gen(int b, int a) {}
 7)    int after = 0;          // Сохраняет метку после конструкции
 8)                            // Используется для инструкций break:
 9)    public static Stmt Enclosing = Stmt.Null;
10) }
```

Строки 6–9 предназначены для генерации трехадресного кода. Метод gen вызывается с передачей ему двух методов, b и a, где b маркирует начало кода инструкции, а a — первую команду после кода для данной инструкции. Метод gen (строка 6) представляет собой заполнитель для методов gen в подклассах. Подклассы While и Do сохраняют метку a в поле after (строка 7), так что она может использоваться любыми включаемыми инструкциями для перехода за пределы охватывающей конструкции. Объект Stmt.Enclosing используется в процессе синтаксического анализа для отслеживания охватывающей конструкции. (В исходном языке с инструкциями continue можно использовать тот же подход для отслеживания конструкции, охватывающей инструкцию continue.)

Конструктор класса If строит узел для инструкции **if** (E) S. Поля expr и stmt хранят соответственно узлы для E и S. Заметим, что expr представляет собой имя поля типа Expr, записанного строчными буквами, как stmt — имя поля типа Stmt, представляющего собой название типа, записанного строчными буквами.

```
 1) package inter;                // Файл If.java
 2) import symbols.*;
 3) public class If extends Stmt {
 4)    Expr expr; Stmt stmt;
 5)    public If(Expr x, Stmt s) {
 6)       expr = x;  stmt = s;
 7)       if( expr.type != Type.Bool )
 8)          expr.error("boolean required in if");
 9)    }
10)    public void gen(int b, int a) {
11)       int label = newlabel(); // Метка кода инструкции
12)       expr.jumping(0, a);     // Проходим при значении
13)          // true, переход к метке a при значении false
14)       emitlabel(label); stmt.gen(label, a);
15)    }
16) }
```

Код объекта `If` состоит из кода с переходами для выражения `expr`, за которым следует код для `stmt`. Как говорилось в разделе А.6, вызов `expr.jumping(0,a)` в строке 12 указывает, что управление должно пройти через код для `expr`, если вычисления дают для этой переменной значение `true`, в противном случае управление переходит к метке `a`.

Реализация класса `Else`, который обрабатывает часть **else** конструкции **if-else**, аналогична реализации класса `If`.

```
 1) package inter;                // Файл Else.java
 2) import symbols.*;
 3) public class Else extends Stmt {
 4)    Expr expr; Stmt stmt1, stmt2;
 5)    public Else(Expr x, Stmt s1, Stmt s2) {
 6)       expr = x; stmt1 = s1; stmt2 = s2;
 7)       if( expr.type != Type.Bool )
 8)          expr.error("boolean required in if");
 9)    }
10)    public void gen(int b, int a) {
11)       int label1 = newlabel();   // Метка для stmt1
12)       int label2 = newlabel();   // Метка для stmt2
13)       expr.jumping(0,label2);    // При true - к stmt1
14)       emitlabel(label1); stmt1.gen(label1, a);
15)       emit("goto L" + a);
16)       emitlabel(label2); stmt2.gen(label2, a);
17)    }
18) }
```

Построение объекта `While` разделено между конструктором `While()`, который создает узел с нулевым дочерним узлом (строка 5), и функцией инициализации `init(x,s)`, которая устанавливает поле `expr` равным `x`, а поле `stmt` равным `s` (строки 6–10). Функция `gen(b,a)` для генерации трехадресного кода (строки 11–17) написана в духе соответствующей функции `gen()` класса `If`.

Отличие между ними заключается в том, что метка a сохраняется в поле after (строка 12) и что за кодом для stmt следует переход к метке b (строка 16) для выполнения следующей итерации цикла while.

```
 1) package inter;                 // Файл While.java
 2) import symbols.*;
 3) public class While extends Stmt {
 4)    Expr expr; Stmt stmt;
 5)    public While() { expr = null; stmt = null; }
 6)    public void init(Expr x, Stmt s) {
 7)       expr = x;  stmt = s;
 8)       if( expr.type != Type.Bool )
 9)          expr.error("boolean required in while");
10)    }
11)    public void gen(int b, int a) {
12)       after = a;               // Сохранение метки a
13)       expr.jumping(0, a);
14)       int label = newlabel();   // Метка для stmt
15)       emitlabel(label); stmt.gen(label, b);
16)       emit("goto L" + b);
17)    }
18) }
```

Класс Do очень похож на класс While.

```
 1) package inter;                 // Файл Do.java
 2) import symbols.*;
 3) public class Do extends Stmt {
 4)    Expr expr; Stmt stmt;
 5)    public Do() { expr = null; stmt = null; }
 6)    public void init(Stmt s, Expr x) {
 7)       expr = x; stmt = s;
 8)       if( expr.type != Type.Bool )
 9)          expr.error("boolean required in do");
10)    }
11)    public void gen(int b, int a) {
12)       after = a;
13)       int label = newlabel();   // Метка для expr
14)       stmt.gen(b,label);
15)       emitlabel(label);
16)       expr.jumping(b,0);
17)    }
18) }
```

Класс Set реализует присваивание с идентификатором в левой части и выражением в правой. В основном, код класса Set предназначен для построения узла и проверки типов (строки 5–17). Функция gen генерирует трехадресную команду (строки 18–21).

```
 1) package inter;                 // Файл Set.java
 2) import lexer.*; import symbols.*;
 3) public class Set extends Stmt {
 4)    public Id id; public Expr expr;
 5)    public Set(Id i, Expr x) {
 6)       id = i; expr = x;
 7)       if ( check(id.type, expr.type) == null )
 8)           error("type error");
 9)    }
10)    public Type check(Type p1, Type p2) {
11)       if ( Type.numeric(p1) && Type.numeric(p2) )
12)           return p2;
13)       else if ( p1 == Type.Bool && p2 == Type.Bool )
14)           return p2;
15)       else
16)           return null;
17)    }
18)    public void gen(int b, int a) {
19)       emit( id.toString() + " = " +
20)            expr.gen().toString() );
21)    }
22) }
```

Класс SetElem реализует присваивание элементу массива.

```
 1) package inter;                 // Файл SetElem.java
 2) import lexer.*; import symbols.*;
 3) public class SetElem extends Stmt {
 4)    public Id array; public Expr index; public Expr expr;
 5)    public SetElem(Access x, Expr y) {
 6)       array = x.array; index = x.index; expr = y;
 7)       if ( check(x.type, expr.type) == null )
 8)           error("type error");
 9)    }
10)    public Type check(Type p1, Type p2) {
11)       if ( p1 instanceof Array || p2 instanceof Array )
12)           return null;
13)       else if ( p1 == p2 )
14)           return p2;
15)       else if ( Type.numeric(p1) && Type.numeric(p2) )
16)           return p2;
17)       else
18)           return null;
19)    }
20)    public void gen(int b, int a) {
21)       String s1 = index.reduce().toString();
22)       String s2 = expr.reduce().toString();
23)       emit(array.toString() + " [ " +
24)            s1 + " ] = " + s2);
25)    }
26) }
```

Класс `Seq` реализует последовательность инструкций. Проверки на нулевые инструкции в строках 6 и 7 выполняются, чтобы избежать лишних меток. Обратите внимание, что для нулевой инструкции `Stmt.Null` никакой код не генерируется, так как метод `gen` класса `Stmt` не выполняет никаких действий.

```
1) package inter;                    // Файл Seq.java
2) public class Seq extends Stmt {
3)    Stmt stmt1; Stmt stmt2;
4)    public Seq(Stmt s1, Stmt s2) { stmt1 = s1; stmt2 = s2; }
5)    public void gen(int b, int a) {
6)       if ( stmt1 == Stmt.Null ) stmt2.gen(b, a);
7)       else if ( stmt2 == Stmt.Null ) stmt1.gen(b, a);
8)       else {
9)          int label = newlabel();
10)         stmt1.gen(b,label);
11)         emitlabel(label);
12)         stmt2.gen(label,a);
13)      }
14)   }
15) }
```

Инструкция `break` передает управление за пределы цикла или инструкции выбора. Класс `Break` использует поле `stmt` для сохранения охватывающей конструкции (синтаксический анализатор гарантирует, что `Stmt.Enclosing` описывает узел синтаксического дерева для охватывающей конструкции). Код объекта `Break` представляет собой переход к метке `stmt.after`, которая маркирует команду, следующую непосредственно после кода `stmt`.

```
1) package inter;                    // Файл Break.java
2) public class Break extends Stmt {
3)    Stmt stmt;
4)    public Break() {
5)       if( Stmt.Enclosing == null )
6)          error("unenclosed break");
7)       stmt = Stmt.Enclosing;
8)    }
9)    public void gen(int b, int a) {
10)      emit( "goto L" + stmt.after);
11)   }
12) }
```

А.8 Синтаксический анализатор

Синтаксический анализатор считывает поток токенов и строит синтаксическое дерево путем вызовов соответствующих конструкторов из разделов А.5–А.7. Текущая таблица символов поддерживается так, как в схеме трансляции на рис. 2.38 в разделе 2.7.

Пакет parser состоит из единственного класса Parser.

```
 1) package parser;                    // Файл Parser.java
 2) import java.io.*; import lexer.*;
 3) import symbols.*; import inter.*;
 4) public class Parser {
 5)    private Lexer lex;    // Лексический анализатор
 6)    private Token look;   // Предпросмотр
 7)    Env top = null;       // Текущая или верхняя
 8)                          // таблица символов
 9)    int used = 0;         // Память для объявлений
10)    public Parser(Lexer l) throws IOException {
11)       lex = l; move(); }
12)    void move() throws IOException {
13)       look = lex.scan(); }
14)    void error(String s) {
15)       throw new Error("near line "+
16)                       lex.line+": "+s); }
17)    void match(int t) throws IOException {
18)       if( look.tag == t ) move();
19)       else error("syntax error");
20)    }
```

Подобно простому транслятору выражений из раздела 2.5 класс Parser содержит по одной процедуре для каждого нетерминала. Процедуры основаны на грамматике, образованной путем устранения левой рекурсии из грамматики исходного языка, приведенной в разделе А.1.

Синтаксический анализ начинается с вызова процедуры program, которая вызывает block() (строка 23) для разбора входного потока и построения синтаксического дерева. Строки 24–27 генерируют промежуточный код.

```
21)    public void program() throws IOException {
22)       // program -> block
23)       Stmt s = block();
24)       int begin = s.newlabel();
25)       int after = s.newlabel();
26)       s.emitlabel(begin);  s.gen(begin, after);
27)       s.emitlabel(after);
28)    }
```

Обработка таблицы символов показана в процедуре block явно.[3] Переменная top (объявленная в строке 7) хранит верхнюю таблицу символов; переменная savedEnv (строка 31) представляет собой связь с предыдущей таблицей символов.

[3]Привлекательной альтернативой является добавление методов push и pop в класс Env с обращением к текущей таблице посредством статической переменной Env.top.

```
29)    Stmt block() throws IOException {
30)       // block -> { decls stmts }
31)       match('{');  Env savedEnv = top;
32)       top = new Env(top);
33)       decls(); Stmt s = stmts();
34)       match('}');  top = savedEnv;
35)       return s;
36)    }
```

Объявления дают нам записи для идентификаторов в таблице символов (строка 43). Хотя здесь это и не показано, объявление также приводит к командам, резервирующим память для идентификаторов во время выполнения.

```
37)    void decls() throws IOException {
38)       while( look.tag == Tag.BASIC ) {
39)          // D -> type ID ;
40)          Type p = type(); Token tok = look;
41)          match(Tag.ID); match(';');
42)          Id id = new Id((Word)tok, p, used);
43)          top.put( tok, id );
44)          used = used + p.width;
45)       }
46)    }
47)    Type type() throws IOException {
48)       Type p = (Type)look;
49)       // Ожидается look.tag == Tag.BASIC
50)       match(Tag.BASIC);
51)       if( look.tag != '[' ) return p; // T -> basic
52)       else return dims(p);            // Тип массива
53)    }
54)    Type dims(Type p) throws IOException {
55)       match('[');  Token tok = look;
56)       match(Tag.NUM);  match(']');
57)       if( look.tag == '[' )
58)       p = dims(p);
59)       return new Array(((Num)tok).value, p);
60)    }
```

Процедура `stmt` содержит инструкцию выбора с вариантами, соответствующими продукциям нетерминала *Stmt*. Каждый вариант строит узел для конструкции с использованием конструкторов, рассмотренных в разделе А.7. Узлы для конструкций while и do строятся, когда синтаксический анализатор встречает соответствующее открывающее конструкцию ключевое слово. Эти узлы строятся до анализа инструкции, чтобы позволить любому оператору break в пределах цикла указывать на охватывающий цикл. Вложенные циклы обрабатываются при помощи переменной `Stmt.Enclosing` в классе `Stmt` и `savedStmt` (объявленной в строке 67) для работы с текущим охватывающим циклом.

```
 61)    Stmt stmts() throws IOException {
 62)       if ( look.tag == '}' ) return Stmt.Null;
 63)       else return new Seq(stmt(), stmts());
 64)    }
 65)    Stmt stmt() throws IOException {
 66)       Expr x;  Stmt s, s1, s2;
 67)       Stmt savedStmt;         // Сохранение охватывающей
 68)                               // конструкции для break
 69)       switch( look.tag ) {
 70)       case ';':
 71)          move();
 72)          return Stmt.Null;
 73)       case Tag.IF:
 74)          match(Tag.IF); match('(');
 75)          x = bool(); match(')');
 76)          s1 = stmt();
 77)          if( look.tag != Tag.ELSE )
 78)             return new If(x, s1);
 79)          match(Tag.ELSE);
 80)          s2 = stmt();
 81)          return new Else(x, s1, s2);
 82)       case Tag.WHILE:
 83)          While whilenode = new While();
 84)          savedStmt = Stmt.Enclosing;
 85)          Stmt.Enclosing = whilenode;
 86)          match(Tag.WHILE); match('(');
 87)          x = bool(); match(')');
 88)          s1 = stmt();
 89)          whilenode.init(x, s1);
 90)          Stmt.Enclosing = savedStmt;
 91)          // Сброс Stmt.Enclosing
 92)          return whilenode;
 93)       case Tag.DO:
 94)          Do donode = new Do();
 95)          savedStmt = Stmt.Enclosing;
 96)          Stmt.Enclosing = donode;
 97)          match(Tag.DO);
 98)          s1 = stmt();
 99)          match(Tag.WHILE); match('(');
100)          x = bool(); match(')');
101)          match(';');
102)          donode.init(s1, x);
103)          Stmt.Enclosing = savedStmt;
104)          // Сброс Stmt.Enclosing
105)          return donode;
106)       case Tag.BREAK:
107)          match(Tag.BREAK); match(';');
108)          return new Break();
109)       case '{':
110)          return block();
```

```
111)         default:
112)            return assign();
113)         }
114)      }
```

Для удобства код для присваивания выделен во вспомогательную процедуру `assign`.

```
115)      Stmt assign() throws IOException {
116)         Stmt stmt;  Token t = look;
117)         match(Tag.ID);
118)         Id id = top.get(t);
119)         if( id == null )
120)            error(t.toString() + " undeclared");
121)         if( look.tag == '=' ) {       // S -> id = E ;
122)            move();  stmt = new Set(id, bool());
123)         }
124)         else {                        // S -> L = E ;
125)            Access x = offset(id);
126)            match('=');  stmt = new SetElem(x, bool());
127)         }
128)         match(';');
129)         return stmt;
130)      }
```

Синтаксический анализ арифметических и булевых выражений аналогичен. В каждом случае создается соответствующий узел синтаксического дерева. Генерация кода для этих выражений различна и рассмотрена в разделах А.5 и А.6.

```
131)      Expr bool() throws IOException {
132)         Expr x = join();
133)         while( look.tag == Tag.OR ) {
134)            Token tok = look;  move();
135)            x = new Or(tok, x, join());
136)         }
137)         return x;
138)      }
139)      Expr join() throws IOException {
140)         Expr x = equality();
141)         while( look.tag == Tag.AND ) {
142)            Token tok = look;  move();
143)            x = new And(tok, x, equality());
144)         }
145)         return x;
146)      }
147)      Expr equality() throws IOException {
148)         Expr x = rel();
149)         while( look.tag == Tag.EQ ||
150)            look.tag == Tag.NE ) {
151)            Token tok = look;  move();
```

```
152)          x = new Rel(tok, x, rel());
153)       }
154)       return x;
155)    }
156)    Expr rel() throws IOException {
157)       Expr x = expr();
158)       switch( look.tag ) {
159)          case '<':
160)          case Tag.LE:
161)          case Tag.GE:
162)          case '>':
163)             Token tok = look;  move();
164)             return new Rel(tok, x, expr());
165)          default:
166)             return x;
167)       }
168)    }
169)    Expr expr() throws IOException {
170)       Expr x = term();
171)       while( look.tag == '+' ||
172)              look.tag == '-' ) {
173)          Token tok = look;  move();
174)          x = new Arith(tok, x, term());
175)       }
176)       return x;
177)    }
178)    Expr term() throws IOException {
179)       Expr x = unary();
180)       while(look.tag == '*' ||
181)             look.tag == '/' ) {
182)          Token tok = look;  move();
183)          x = new Arith(tok, x, unary());
184)       }
185)       return x;
186)    }
187)    Expr unary() throws IOException {
188)       if( look.tag == '-' ) {
189)          move();
190)          return new Unary(Word.minus, unary());
191)       }
192)       else if( look.tag == '!' ) {
193)          Token tok = look;  move();
194)   return new Not(tok, unary());
195)       }
196)       else return factor();
197)    }
```

Остальной код синтаксического анализатора работает с "множителями" в выражениях. Вспомогательная процедура `offset` генерирует код для вычисления адресов массивов (см. раздел 6.4.3).

```
198)    Expr factor() throws IOException {
199)       Expr x = null;
200)       switch( look.tag ) {
201)       case '(':
202)          move(); x = bool(); match(')');
203)          return x;
204)       case Tag.NUM:
205)          x = new Constant(look, Type.Int);
206)          move(); return x;
207)       case Tag.REAL:
208)          x = new Constant(look, Type.Float);
209)          move(); return x;
210)       case Tag.TRUE:
211)          x = Constant.True; move(); return x;
212)       case Tag.FALSE:
213)          x = Constant.False; move(); return x;
214)       default:
215)          error("syntax error");
216)          return x;
217)       case Tag.ID:
218)          String s = look.toString();
219)          Id id = top.get(look);
220)          if( id == null )
221)             error(look.toString() +
222)                   " undeclared");
223)          move();
224)          if( look.tag != '[' ) return id;
225)          else return offset(id);
226)       }
227)    }
228)    Access offset(Id a) throws IOException {
229)       // I -> [E] | [E] I
230)       Expr i; Expr w; Expr t1, t2; Expr loc;
231)       Type type = a.type;
232)       // Первый индекс, I -> [ E ]
233)       match('['); i = bool(); match(']');
234)       type = ((Array)type).of;
235)       w = new Constant(type.width);
236)       t1 = new Arith(new Token('*'), i, w);
237)       loc = t1;
238)       // Многомерный I -> [ E ] I
239)       while( look.tag == '[' ) {
240)          match('['); i = bool(); match(']');
241)          type = ((Array)type).of;
242)          w = new Constant(type.width);
243)          t1 = new Arith(new Token('*'), i, w);
244)          t2 = new Arith(new Token('+'), loc, t1);
245)          loc = t2;
246)       }
247)       return new Access(a, loc, type);
```

```
248)     }
249) }
```

А.9 Построение начальной стадии

Код пакетов располагается в пяти каталогах: `main`, `lexer`, `symbols`, `parser` и `inter`. Команды для построения компилятора варьируются от одной операционной системы к другой. Вот команды для UNIX:

```
javac lexer/*.java
javac symbols/*.java
javac inter/*.java
javac parser/*.java
javac main/*.java
```

Команда `javac` создает `.class`-файлы для каждого класса. После этого запуск транслятора может быть осуществлен при помощи команды `java main.Main`, за которой следует исходная транслируемая программа. Например, если взять файл `test`

```
1) {                          // Файл test
2)    int i; int j; float v; float x; float[100] a;
3)    while( true ) {
4)       do i = i+1; while( a[i] < v);
5)       do j = j-1; while( a[j] > v);
6)       if( i >= j ) break;
7)       x = a[i]; a[i] = a[j]; a[j] = x;
8)    }
9) }
```

и транслировать его, на выходе мы получим

```
 1) L1:L3:  i = i + 1
 2) L5:     t1 = i * 8
 3)         t2 = a [ t1 ]
 4)         if t2 < v goto L3
 5) L4:     j = j - 1
 6) L7:     t3 = j * 8
 7)         t4 = a [ t3 ]
 8)         if t4 > v goto L4
 9) L6:     iffalse i >= j goto L8
10) L9:    goto L2
11) L8:    t5 = i * 8
12)        x = a [ t5 ]
13) L10:   t6 = i * 8
14)        t7 = j * 8
15)        t8 = a [ t7 ]
16)        a [ t6 ] = t8
17) L11:   t9 = j * 8
```

```
18)         a [ t9 ] = x
19)         goto L1
20) L2:
```

Проверьте сами.

ПРИЛОЖЕНИЕ Б

Поиск линейно независимых решений

Алгоритм Б.1. Поиск максимального множества линейно независимых решений $A\vec{x} \geqslant \vec{0}$ и выражение их как строк матрицы B

ВХОД: матрица A размером $m \times n$.

ВЫХОД: матрица B линейно независимых решений $A\vec{x} \geqslant \vec{0}$.

МЕТОД: псевдокод алгоритма приведен ниже. Заметим, что $X[y]$ означает y-ю строку матрицы X, $X[y:z]$ означает строки y–z матрицы X, а $X[y:z][u:v]$ — прямоугольник в матрице X со строками y–z и столбцами u–v. □

$M = A^T$;

$r_0 = 1$;

$c_0 = 1$;

$B = I_{n \times n}$; /* Тождественная матрица размером $n \times n$ */

while (**true**) {

/* 1. Преобразуем $M[r_0 : r' - 1][c_0 : c' - 1]$ в диагональную матрицу
с положительными диагональными элементами, и
$M[r' : n][c_0 : m] = 0$. $M[r' : n]$ является решением. */

$r' = r_0$;

$c' = c'_0$;

while (существует $M[r][c] \neq 0$, такое, что и $r - r'$, и $c - c'$ не меньше 0) {

Переместить опорный элемент $M[r][c]$ в $M[r'][c']$ путем
перестановки соответствующих строк и столбцов;

Поменять местами строки r и r' в B;

if ($M[r'][c'] < 0$) {

$M[r'] = -1 * M[r']$;

$B[r'] = -1 * B[r']$;

```
        }
        for ( row = r' до n ) {
            if ( row ≠ r' и M[row][c'] ≠ 0 ) {
                u = −(M[row][c']/M[r'][c']);
                M[row] = M[row] + u * M[r'];
                B[row] = B[row] + u * B[r'];
            }
        }
        r' = r' + 1;
        c' = c' + 1;
    }

    /* 2. Поиск решения, кроме M[r' : n]. Оно должно быть
            неотрицательной комбинацией M[r_0 : r' − 1][c_0 : m] */
    Ищем k_{r_0}, ..., k_{r'−1} ⩾ 0, такие, что k_{r_0} M[r_0][c' : m] + ···
            + k_{r'−1} M[r' − 1][c' : m] ⩾ 0;
    if ( существует нетривиальное решение, скажем, k_r > 0 ) {
        M[r] = k_{r_0} M[r_0] + ··· + k_{r'−1} M[r' − 1];
        NoMoreSoln = false;
    }
    else /* M[r' : n] является единственным решением */
        NoMoreSoln = true;

    /* 3. Делаем M[r_0 : r_n − 1][c_0 : m] ⩾ 0 */
    if ( NoMoreSoln ) { /* Перемещаем решения M[r' : n] в M[r_0 : r_n − 1] */
        for ( r = r' до n )
            Обмениваем строки r и r_0 + r − r' в M и B;
        r_n = r_0 + n − r' + 1;
    }
    else { /* Используем добавление строки для поиска решений */
        r_n = n + 1;
        for ( col = c' до m )
            if ( существует M[row][col] < 0, такое, что row ⩾ r_0 )
```

```
                if ( существует M[r][col] > 0, такое, что r ≥ r0 ) {
                        for ( row = r0 до rn − 1 )
                                if ( M[row][col] < 0 ) {
                                        u = ⌈(M[row][col]/M[r][col])⌉;
                                        M[row] = M[row] + u * M[r];
                                        B[row] = B[row] + u * B[r];
                }
                else
                        for ( row = rn − 1 до r0 с шагом −1 )
                                if ( M[row][col] < 0 ) {
                                        rn = rn − 1;
                                        Перестановка M[row] и M[rn];
                                        Перестановка B[row] и B[rn];
                                }
}

/* 4. Делаем M[r0 : rn − 1][1 : c0 − 1] ≥ 0 */
for ( row = r0 до rn − 1 )
        for ( col = 1 до c0 − 1 )
                if ( M[row][col] < 0 ) {
                        Выбираем r, такое, что M[r][col] > 0 и r < r0;
                        u = ⌈(M[row][col]/M[r][col])⌉;
                        M[row] = M[row] + u * M[r];
                        B[row] = B[row] + u * B[r];
                }

/* 5. При необходимости повторяем со строками M[rn : n] */
if ( NoMoreSoln или rn > n или rn == r0 ) {
        Удаляем строки rn–n из B;
        return B;
}
else {
        cn = m + 1;
```

```
        for ( col = m до 1 с шагом −1 )
            if ( не существует M[r][col] > 0, такого, что r < r_n ) {
                c_n = c_n − 1;
                Переставить строки col и c_n в M;
            }
        r_0 = r_n;
        c_0 = c_n;
    }
}
```

Предметный указатель

C

CISC, 53, 620

D

Datalog, 1082

J

JVM, 620

L

l-значение, 143, 453
Lex, 191, 371
    Прогностический оператор, 197, 226
    Программа, 192
    Разрешение конфликтов, 196
LL(k), 283
LR(0)-автомат, 312, 316
LR-анализатор, 319
LR-грамматика, 301

N

NUMA, 916

O

O-обозначения, 213

P

Purify, 567

R

r-значение, 143, 453
RISC, 52, 620

S

SIMD, 53
SLR(1)-анализатор, 324
SLR(1)-таблица, 324
SLR-анализ, 322
SPMD, 919

V

VLIW, 51, 53, 844

Y

Yacc, 311, 363
    Восстановление после ошибки, 372
    Неоднозначная грамматика, 368
    Семантические действия, 367

А

Абстрактное синтаксическое дерево, 110
Адрес возврата, 533
Активация, 529
    Запись, 532
Активный префикс, 327
Алгоритм
    LR-анализа, 320
    Анализ активных переменных, 734
    Анализ на основе областей, 812
    Ахо–Корасик, 187
    Выведение типа для полиморфных функций, 485
    Вычисление границ для заданного порядка переменных, 943
    Генерации кода с использованием динамического программирования, 697
    Генерация кода для помеченного дерева выражения, 690, 692
    Генерация кода, последовательно выполняющего части разбиения программы, 992
    Глубинное остовное дерево и упорядочение графа в глубину, 792
    Достигающие определения, 730

Доступные выражения, 739
Евклида, 968
Инкрементное вычисление программы Datalog, 1091
Исключение Фурье–Моцкина, 942
Итеративное решение обобщенной задачи потока данных, 753
Кнута–Морриса–Пратта, 187
Кока–Янгера–Касами, 252, 300
Копирующий сборщик Чени, 587
Левая факторизация грамматики, 278
Мак-Нотона–Ямады–Томпсона, 213
Максимизация степени параллельности с использованием O(1) синхронизаций, 1012
Метод номера значения построения узла ориентированного ациклического графа, 448
Минимизация количества состояний ДКА, 238
Моделирование ДКА, 203
Моделирование НКА, 210
Объединение двух BDD, 1122
Определение информации о живучести и последующем использовании для каждой инструкции базового блока, 643
Определение предпросмотров, 346
Оптимизация локальности данных в однопроцессорной системе, 1047
Оптимизация параллелизма и локальности данных в многопроцессорной системе, 1049
Отложенное перемещение кода, 775, 784
Планирование на основе областей, 871
Планирование списка базового блока, 861
Поезда, 599
Поиск всех степеней максимально крупномодульного параллелизма в программе, 1035
Поиск доминаторов, 789
Поиск множества допустимых максимально независимых отображений временных разбиений для внешнего последовательного цикла, 1027
Поиск не требующего синхронизации аффинного разбиения программы с наивысшим рангом, 988
Построение восходящего порядка областей приводимого графа потока, 808
Построение ДКА из регулярного выражения, 235
Построение естественного цикла обратной дуги, 797
Построение множеств LR(1)-пунктов, 333
Построение подмножества ДКА из НКА, 206
Построение таблиц канонического LR-анализа, 336
Построение таблицы SLR-анализа, 323
Построение таблицы предиктивного синтаксического анализа, 290
Предиктивный синтаксический анализ, управляемый таблицей, 293
Преобразования регулярного выражения в НКА, 213
Программная конвейеризация, 895
Программная конвейеризация ациклического графа зависимости данных, 889
Программная конвейеризация с модульным расширением переменной, 901
Простое вычисление программ Datalog, 1089
Простое построение LALR-таблицы, 341
Разбиение трехадресных команд на базовые блоки, 641
Решение задачи целочисленного линейного программирования методом ветвей и границ, 974

Сборка мусора "пометить и подмести", 578
Сборщик мусора "пометить и подмести" Бейкера, 582
Сборщик мусора "пометить и сжать", 585
Сжатие массива, 1043
Символический анализ на основе областей, 829
Унификация пар узлов в графе типа, 488
Устранение левой рекурсии, 277
Эрли, 252
Эффективное вычисление ядер наборов множеств LALR(1)-пунктов, 347
Алфавит, 165
Входной, 200
Альтернатива, 260
Анализ, 32
Активных переменных, 733
Лексический, 33
Предиктивный, 278
Семантический, 37
Символический, 819
Синтаксический, 35, 252
Указателей, 1095
Анализ потоков данных, 719, 744
Анализатор
Предиктивный, 106
Архитектура процессора, 842
Ассемблер, 32
Ассоциативность, 85, 171, 353
Атрибут, 91, 159, 383
Наследуемый, 93, 385
Синтезируемый, 93, 384
Аффинное выражение, 820

Б

Базовый блок, 640, 641
Лидер, 641
Барьерная синхронизация, 919, 1006
Блочная структура, 62
Буферизация ввода, 162
Бэкуса–Наура форма, 76, 251

В

Векторная машина, 921
Виртуальный метод, 1062, 1076
Висящий указатель, 566
Вложение циклов, 924
Волновое распараллеливание, 1031
Восстановление после ошибки, 255, 358, 372
Глобальная коррекция, 257
На уровне фразы, 297, 359
Продукции ошибок, 257
Режим паники, 256, 295, 359
Уровень фразы, 257
Встраивание, 50, 1061
Выведение типа, 478
Вывод, 261
Выполнение с опережением, 852
Выравнивание адресов, 463
Выражение типа, 459
Вычет цикла, 972
Вычисление, инвариантное относительно цикла, 714

Г

Генерация
Кода, 39, 617, 660
Выбор порядка вычислений, 625
Промежуточного кода, 38, 136
Глобальное планирование, 864
Глубина вложенности, 545
Грамматика, 251
LALR(1), 342
LL(1), 288
LL(k), 283
LR, 301
LR(1), 337
SLR(1), 324
Атрибутная, 386
Контекстно-свободная, 79, 258, 310
Леворекурсивная, 275
Неоднозначная, 84, 265, 353, 368
Устранение неоднозначности, 273
Грамматический символ, 260
Граф
Взаимодействия регистров, 676
Вызовов, 1062
Зависимостей, 391

Зависимостей программы, 1007
Зависимости данных, 858
Критический путь, 861
Ориентированный ациклический, 445, 648
Переходов, 200
Потока, 640, 644, 720
Глубина, 795
Полный, 802
Представление, 646
Цикл, 646
Раскраска, 676
Сильно связанный компонент, 891
Топологическая сортировка, 393

Д

Дерево
Активаций, 531
Глубинное остовное, 790
Доминаторов, 787
Крона, 263
Разбора, 82, 100, 263
Аннотированное, 387
Синтаксическое, 35, 400
Дескриптор
Адреса, 661
Регистра, 661
Диаграмма бинарного выбора, 1110, 1115
Упорядоченная, 1118
Диаграмма переходов, 179, 289
Динамическое программирование, 695
Диофантово уравнение, 967
Диспетчер памяти, 555
Дисплей, 552
Доминатор, 787
Достигающее определение, 721, 725
Алгоритм, 730
Доступное выражение, 735
Дублирование констант, 722, 760

З

Зависимость через данные, 846
Загрузчик, 32
Задача потока данных, 722
Закон Амдаля, 917
Замыкание
Клини, 167, 168
Позитивное, 168
Запись активации, 532
Заполнение, 526
Зарезервированное слово, 122

И

Идемпотентность, 171
Идентификатор, 34, 61
Распознавание, 181
Иерархическое время, 1012
Иерархическое приведение, 903
Иерархия памяти, 51
Имитационное моделирование, 55
Инкапсуляция, 65
Инкрементная трансляция, 470
Интерпретатор, 30
Исключение Фурье–Моцкина, 942
Использование, 643
Итерация, 168

К

Кадр, 532
Канонический набор, 312
Ключевое слово, 121
Распознавание, 181
Код
Мертвый, 649, 651
Трехадресный, 38
Коммутативность, 171
Компилятор, 29
Заключительная стадия, 33
Начальная стадия, 33
Структура, 32
Компоновщик, 32
Конвейер команд, 842
Конвейеризация, 1014
Конвейеризация программная, 876
Конечный автомат, 199
Важные состояния НКА, 229
Детерминированный, 200
ДКА с минимальным количеством состояний, 237
Недетерминированный, 199, 200, 268
Преобразование НКА в ДКА, 206
Тупиковое состояние, 240
Тупиковые состояния ДКА, 227

Конкатенация, 166, 168
Конкретное синтаксическое дерево, 111
Конструктор типа, 459
Контекстно-свободная грамматика, 258, 310
Кусочно-аффинное разбиение, 982
Куча, 528, 555
Кэш, 557, 560
    Интерференция, 933
    Строка, 559

Л

Левая рекурсия, 107
    Устранение, 276
Левая факторизация, 278
Левоассоциативность, 85
Лексема, 33, 118, 158
Лексическая ошибка, 255
Лексический анализ, 33, 118
Лексический анализатор, 123, 155
Лемма Фаркаша, 1026, 1027
Линейно независимое множество, 954
Линкер, 32
Логическая ошибка, 255
Локальная оптимизация, 668
Локальность, 559, 920, 1039
    Временная, 559, 1040
    Данных, 911
    И параллельность, 1040
    Пространственная, 559, 920
Локальные общие подвыражения, 649

М

Макрос, 32
Матрица
    Нуль-пространство, 955
    Ранг, 954
    Унимодулярная, 1003
Мертвый код, 649, 651
Метод, 50
Модульное расширение переменной, 901
Мутатор, 569

Н

Наименьшая фиксированная точка, 730
Недетерминированный конечный автомат, 200, 268
Неоднозначная грамматика, 265
Неоднозначность, 353, 379
    Грамматики, 84
    Устранение, 273
Нетерминал, 79, 258, 260
Номер значения, 447
Нормальная форма
    Хомски, 299
Нуль-пространство матрицы, 955

О

Область, 804
    Видимости, 58, 129
    Выходной блок, 808
Обратные поправки, 504
Обход дерева, 95
    В глубину, 95
    В обратном порядке, 97
    В прямом порядке, 97
Общие подвыражения, 445, 710
    Глобальные, 769
Объединение, 166, 168
Объектный модуль, 620
Объявление, 462
Ожидаемость выражений, 774
Оконечная рекурсия, 114
Оператор сбора, 729
Опережающее чтение, 120
Определение
    L-атрибутное, 394, 415, 422
    S-атрибутное, 386, 394
    переменной достигающее, 725
    Синтаксически управляемое, 384
Оптимизация, 46
    Выражения, инвариантные относительно циклов, 770
    Глобальные общие подвыражения, 769
    Дублирование констант, 714
    Использование машинных идиом, 671
    Кода, 39
    Локальная, 668
    Отложенное перемещение кода, 774
    Перемещение кода, 714
    Потока управления, 670
    Потоков данных, 49
    Распространечие копирований, 712

Сжатие массива, 1041
Удаление бесполезного кода, 713
Удаление общих подвыражений, 710
Устранение излишних загрузок и сохранений, 668
Устранение недостижимого кода, 669
Устранение частичной избыточности, 768
Частично избыточные выражения, 771
Отложенное перемещение кода, 774
Относительный адрес, 459
Ошибка
Лексическая, 255
Логическая, 255
Семантическая, 255
Синтаксическая, 255

П

Память, 51, 525
Выделение, 555
Выделение в стеке, 528, 635
Динамическая, 558
Диспетчер, 555
Иерархия, 557
Освобождение, 555
Статическая, 558
Статическое выделение, 632
Статическое и динамическое распределение, 527
Страница, 559
Утечка, 565
Фрагментация, 561
Параллелизм, 51
На уровне задач, 919
Параметрический полиморфизм, 482
Перегрузка, 144
Передаточная функция, 723, 750
Передача параметров
По значению, 68
По имени, 69
По ссылке, 69
Переменная, 61
Глобальная, 543
Индукции, 820
Ссылочная, 819
Перенос, 305
Планирование
Глобальное, 864
Перемещение кода, 874
Списков базовых блоков, 859
Планирование кода, 845
Повторное использование данных, 951
Подпоследовательность, 167
Подстрока, 167
Подсчет ссылок, 567
Полностью переставляемые циклы, 1017, 1030
Полурешетка, 744
Высота, 750
Порождение, 81, 261
Каноническое, 263
Последовательная сверхрелаксация, 1016
Последовательное вычисление, 696
Последовательность
Возврата, 535
Вызовов, 535
Постфиксная запись, 91
Поток
Данных, 722, 744
Анализ, 719
Управления, 491, 729
Правило последнего вложения, 130
Правоассоциативность, 86
Предвыборка, 1053
Предикатная команда, 854
Предиктивный анализ, 104
Предиктивный анализатор, 106
Предложение, 166
Предпросмотр, 332
Препроцессор, 31
Префикс, 167
Приведение типа, 37, 144
Примитивное аффинное преобразование, 1000
Приоритет, 353
Приоритет операторов, 86
Проверка
Статическая, 443
Типов, 37, 56, 143, 459, 477, 519
Программная конвейеризация, 876, 895
Продукция, 79, 259
Единичная, 299

Одинарная, 111
Ошибки, 257
Промежуточное представление, 136
Протокол когерентной кэш-памяти, 915
Проход, 41
Процедура
Вложенная, 543
Псевдоним, 70, 848
Пункт
LR(0), 311
LR(1), 331
Путь выполнения, 720

Р

Разбиение аффинного пространства, 981
Разбор, 35
Разыменование висящего указателя, 565
Ранг матрицы, 954
Распределение регистров, 40
Распространение констант, 760
Расширение переменной, 900
Модульное, 901
Регистр
Глобальное распределение, 672
Граф взаимодействия, 676
Дескриптор, 661
Назначение, 623
Распределение, 623
Регулярное выражение, 168, 268
Эквивалентность, 170
Регулярное множество, 170
Регулярное определение, 171
Рекурсия
Левая, 107
Оконечная, 114
Решетка, 748
Диаграмма, 747
Произведения, 749

С

Сборка мусора, 50, 528, 569
Алгоритм поезда, 599
Инкрементная, 591
Конкурентная, 605
Копирующая, 587
На основе отслеживания, 574
Отложенный подсчет ссылок, 576
Параллельная, 605
Перемещающая, 583
По поколениям, 597
С подсчетом ссылок, 574
Свертка, 302, 305
Свертывание констант, 652
Связь
Доступа, 534, 547, 548
Управления, 534
Семантическая ошибка, 255
Семантический анализ, 37
Семантическое действие, 97, 406
Сентенциальная форма, 262
Сжатие массива, 1041
Сигнатура, 66, 448
Символический анализ, 819
Симметричная мультипроцессорность, 914
Синтаксическая ошибка, 255
Синтаксически управляемая трансляция, 76
Синтаксически управляемое определение, 92, 96, 384
Синтаксический анализ, 35, 82, 84, 252
Восходящий, 100, 253, 301
Методом рекурсивного спуска, 104, 283
Нисходящий, 100, 101, 253, 281
Перенос/свертка, 301, 304
Предиктивный, 104
Синтаксический анализатор
Нерекурсивный предиктивный, 292
Предиктивный, 288
Синтаксическое дерево, 35, 77, 110, 137
Абстрактное, 110
Аннотированное, 93
Конкретное, 111
Синтез, 33
Синтез типа, 477
Синхронизирующий барьер, 1006
Система типов
Надежная, 477
Сканирование, 33
Сканируемый символ, 102
Слабые ссылки, 610
Словарь, 448
Слово, 166

Сокращенные вычисления, 492
Ссылочная переменная, 819
Стартовый символ, 79, 259
Статическая проверка программы, 142
Статическое единственное присваивание, 457
Стек, 528
Строка, 166
  Вызовов, 1067
  Конкатенация, 166
  Основа, 302
  Подпоследовательность, 167
  Подстрока, 167
  Префикс, 167
  Пустая, 81, 166
  Суффикс, 167
  Фибоначчи, 189
Структура
  Дистрибутивная, 752
  Монотонная, 751
Суффикс, 167
Схема трансляции, 91
  Постфиксная, 407
  Синтаксически управляемая, 406

Т

Таблица
  LALR-анализа, 342
  Каноническая LR(1)-анализа, 337
  Модульного резервирования ресурсов, 886
  Переходов, 201
  Символов, 33, 40, 128
Терминал, 79, 118, 258, 260
Тип, 404
Токен, 33, 118, 157
  Атрибут, 159
  Синхронизирующий, 257
Точка вызова, 1062
Трансляция
  Бинарная, 54
Трансформации, сохраняющие семантику, 710
Трехадресный код, 38, 144, 444, 450
  Косвенная тройка, 456
  Тройка, 455
  Четверка, 454

У

Унимодулярное преобразование, 1003
Упаковка данных, 526
Упорядочение в глубину, 790
Упреждающая выборка, 853
Утечка памяти, 565

Ф

Фаза компиляции, 33
Фактические параметры, 68
Фибоначчи строка, 189
Форма Бэкуса–Наура, 76, 251
Формальные параметры, 68
Функция
  Аффинная, 912
  Отказа, 187
  Передаточная, 723, 750
  Переходов, 200

Х

Хеширование, 448

Ц

Цикл
  Естественный, 796
  Перекрестные циклы, 883

Ч

Частичный порядок, 745
Числа Ершова, 689

Ш

Шаблон, 157

Э

Эквивалентность типов, 461

Я

Язык, 166, 262
  Algol, 62
  Algol 60, 69, 521, 544
  Awk, 44
  C, 43, 49, 58, 62, 68, 162, 275, 279, 542, 958
  C++, 43, 49, 62, 65, 67, 68, 461
  Cobol, 43, 49

C#, 43, 49, 62
Fortran, 42, 43, 49, 923, 958
Haskell, 43
Java, 43, 49, 50, 58, 62, 65, 67, 68, 117,
164, 279, 461, 568
JavaScript, 44
Lisp, 43, 568
ML, 43, 67, 545, 568
Modula-3, 568
NOMAD, 43
OPS5, 43
Pascal, 544
Perl, 44, 568
PHP, 44
PL/I, 164
Postscript, 43
Prolog, 43, 568
Python, 44
REXX, 44
Ruby, 43, 44
Simula, 49
Simula 67, 43
Smalltalk, 43, 49, 568
SQL, 43, 55, 174, 199
Tcl, 44
UNCOL, 522
Verilog, 54
VHDL, 54
Ассемблер, 42
Декларативный, 43
Динамически типизированный, 569
Императивный, 43
Исходный, 29
Контекстно-свободный, 262
Объектно-ориентированный, 43
Определяемый НКА, 202
Поколения, 43
Строго типизированный, 477
Сценария, 43
фон Неймана, 43
Целевой, 29

Научно-популярное издание

Альфред В. Ахо, Моника С. Лам, Рави Сети, Джеффри Д. Ульман

Компиляторы: принципы, технологии и инструментарий

2-е издание

Литературный редактор *Л.Н. Красножон*
Верстка *А.Н. Полинчик*
Художественный редактор *С.А. Чернокозинский*
Корректоры *Л.А. Гордиенко*, *Л.В. Чернокозинская*

Издательский дом "Вильямс"
127055, г. Москва, ул. Лесная, д. 43, стр. 1

Подписано в печать 08.02.2008. Формат 70×100/16.
Гарнитура Times. Печать офсетная.
Усл. печ. л. 56,4. Уч.-изд. л. 51,1.
Тираж 1500 экз. Заказ № 7937.

Отпечатано по технологии CtP
в ОАО "Печатный двор" им. А. М. Горького
197110, Санкт-Петербург, Чкаловский пр., 15.

Это новое издание классической "книги Дракона" полностью переработано и включает последние разработки в области компиляции. Книга предлагает читателю подробное введение в разработку компиляторов, после чего сосредоточивает внимание на применении методов компиляции для решения широкого круга задач проектирования и разработки программного обеспечения. Первая половина книги написана таким образом, чтобы ее можно было использовать в качестве учебника для начинающих, посвященного компиляторам, а вторая может использоваться как пособие по оптимизации кода для студентов старших курсов.

Новый материал представлен в следующих главах:

Глава 7. Среды времени выполнения

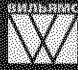
Глава 10. Параллелизм на уровне команд

Глава 11. Оптимизация параллелизма и локальности

Глава 12. Межпроцедурный анализ

gradiance

Gradiance — это специализированный Web-ресурс для студентов и преподавателей. По теме "Компиляторы" он предлагает большой набор заданий с использованием специальной методики, называющейся "корневыми вопросами". Студенты работают над задачей так, как будто это обычное домашнее задание. Затем их знания проверяются при помощи случайно выбранных вопросов с несколькими ответами на каждый из них. При неверном ответе студент получает подсказку, что именно и почему неверно в ответе, или описание более общего подхода к решению задачи. После этого студент может еще раз попытаться ответить на заданный вопрос. Такая методика позволяет надежно закрепить в памяти изученный материал. Gradiance позволяет преподавателям давать студентам домашние задания и проверять их выполнение в автоматическом режиме. Дополнительную информацию о Gradiance вы найдете на сайте **aw.com/gradiance.**



www.williamspublishing.com

Addison-Wesley Computing
Leading Authors • Quality Products
aw.com/computing