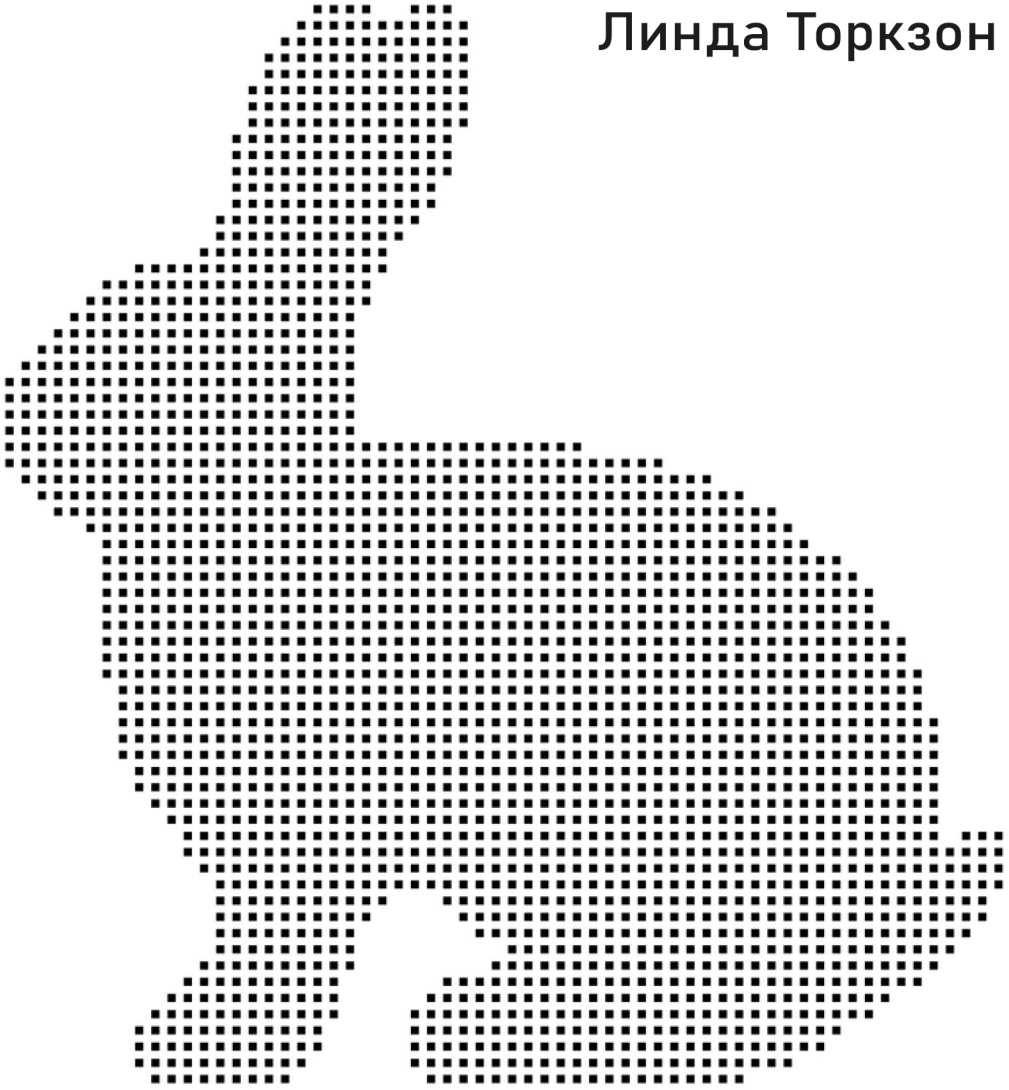
Кит Д. Купер
Линда Торкзон



Создание компиляторов: инженерный подход

ДМК ПРЕСС
ИЗДАТЕЛЬСТВО

Кит Д. Купер, Линда Торкзон

Создание компиляторов: инженерный подход

Keith D. Cooper, Linda Torczon

Engineering a Compiler

Third Edition



MORGAN KAUFMANN PUBLISHERS
AN IMPRINT OF ELSEVIER

Кит Д. Купер, Линда Торкзон

Создание компиляторов: инженерный подход

ДМК Пресс
издательство
Москва, 2026

УДК 004.4'422
ББК 32.973
К92

Купер К. Д., Торкзон Л.
К92 Создание компиляторов: инженерный подход / пер. с англ. А. А. Слинкина. – М.: ДМК Пресс, 2026. – 854 с.: ил.

ISBN 978-5-93700-428-4

Эта книга является воплощением авторской философии построения компиляторов, которая сложилась в результате сорока с лишним лет исследований, преподавания и практики. В ней исследуется все пространство проектирования компилятора и уделяется внимание как глубине проблем, так и широте охвата возможных решений. Описываются некоторые способы разрешения проблем компиляции, а также ограничения, из-за которых эти решения стали привлекательными. Важное преимущество книги заключается в оптимальном сочетании фундаментальных принципов построения современных компиляторов и практических выводов из собственного опыта авторов.

Издание адресовано широкому кругу читателей – от студентов, изучающих программирование, до опытных разработчиков, повышающих квалификацию и расширяющих свой кругозор.

УДК 004.4'422
ББК 32.973

This Russian edition of Engineering a Compiler, Third Edition (9780128154120) by Keith D. Cooper & Linda Torczon is published by arrangement with Elsevier Inc. The translation has been undertaken by DMK Press at its sole responsibility. Practitioners and researchers must always rely on their own experience and knowledge in evaluating and using any information, methods, compounds or experiments described herein. Because of rapid advances in the medical sciences, in particular, independent verification of diagnoses and drug dosages should be made. To the fullest extent of the law, no responsibility is assumed by Elsevier, authors, editors or contributors in relation to the translation or for any injury and/or damage to persons or property as a matter of products liability, negligence or otherwise, or from any use or operation of any methods, products, instructions, or ideas contained in the material herein.

Все права защищены. Любая часть этой книги не может быть воспроизведена в какой бы то ни было форме и какими бы то ни было средствами без письменного разрешения владельцев авторских прав.

ISBN 978-0-12-815412-0 (англ.)

ISBN 978-5-93700-428-4 (рус.)

Copyright © 2023 Elsevier Inc.
All rights reserved.
© Перевод, оформление, издание, ДМК Пресс, 2026

Посвящаем эту книгу:

- *нашим родителям, которые привили нам жажду знаний и были нам опорой, когда мы приобретали навыки, необходимые для ее удовлетворения;*
- *нашим детям, которые еще раз показали нам, каким прекрасным может быть процесс обучения и взросления;*
- *нашим половинкам, без которых эта книга никогда не была бы написана.*

Содержание

От издательства 15
Об авторах 16
Предисловие 17

Глава 1. Обзор процесса компиляции 22
1.1. Введение 22
1.2. Структура компилятора 28
1.3. Краткий обзор трансляции 32
1.3.1. Фаза анализа 32
1.3.2. Оптимизатор 37
1.3.3. Фаза синтеза 39
1.4. Техническая разработка 44
1.5. Итоги и перспективы 45
Замечания к этой главе 46
Упражнения 47

Глава 2. Сканеры 49
2.1. Введение 49
2.2. Распознавание слов 52
2.2.1. Формализм для описания распознавателей 54
2.2.2. Распознавание более сложных слов 56
2.3. Регулярные выражения 58
2.3.1. Формализация нотации 60
2.3.2. Примеры регулярных выражений 61
2.3.3. Свойства замкнутости РВ 64
2.4. От регулярного выражения к сканеру 67
2.4.1. Недетерминированные конечные автоматы 67
2.4.2. РВ в НКА: построение Томпсона 70
2.4.3. НКА в ДКА: построение подмножеств 71
2.4.4. ДКА в минимальный ДКА 76
2.4.5. Использование ДКА в качестве сканера 81
2.5. Реализация сканеров 84
2.5.1. Таблично управляемые сканеры 85
2.5.2. Непосредственно кодируемые сканеры 88
2.5.3. Сканеры, написанные вручную 90
2.5.4. Практические вопросы реализации 91
2.6. Дополнительные темы 95
2.6.1. Преобразование ДКА в регулярное выражение 96

2.6.2. Регулярные выражения, свободные от замыканий 97
2.6.3. Альтернатива алгоритму минимизации ДКА 98
2.7. Итоги и перспективы 101
Замечания к этой главе 101
Упражнения 102

Глава 3. Парсеры 105

3.1. Введение 105
3.2. Выражение синтаксиса 107
3.2.1. Почему бы не использовать регулярные выражения? 108
3.2.2. Контекстно-свободные грамматики 109
3.2.3. Более сложные примеры 112
3.2.4. Кодирование семантики в структуре 115
3.2.5. Нахождение вывода для заданной строки 118
3.3. Нисходящий разбор 119
3.3.1. Преобразование грамматики 121
3.3.2. Нисходящие парсеры рекурсивного спуска 132
3.3.3. Таблично управляемые LL(1)-парсеры 133
3.4. Восходящий разбор 138
3.4.1. Алгоритм LR(1)-разбора 140
3.4.2. Построение таблиц LR(1)-разбора 147
3.4.3. Ошибки при построении таблиц 157
3.5. Практические вопросы 160
3.5.1. Восстановление после ошибки 161
3.5.2. Унарные операторы 162
3.5.3. Обработка контекстно-зависимой неоднозначности 163
3.6. Дополнительные темы 165
3.6.1. Оптимизация грамматики 165
3.6.2. Уменьшение размера таблиц LR(1)-разбора 167
3.7. Итоги и перспективы 172
Замечания к этой главе 172
Упражнения 173

Глава 4. Промежуточные представления 176

4.1. Введение 176
4.2. Таксономия 179
4.3. Графовые ПП 183
4.3.1. Деревья, связанные с синтаксисом 183
4.3.2. Графы 187
4.4. Линейные ПП 192
4.4.1. Код для стековой машины 194
4.4.2. Трехадресный код 195
4.4.3. Представление линейных кодов 196
4.4.4. Построение ГПУ из линейного кода 198
4.5. Таблицы символов 201
4.5.1. Разрешение имен 202

4.5.2. Реализация таблиц 205
4.6. Пространства имен 207
4.6.1. Пространства имен в ПП 209
4.6.2. Форма с единственным статическим присваиванием 212
4.7. Размещение значений в памяти 215
4.7.1. Модели памяти 215
4.7.2. Хранение значений в регистрах 218
4.7.3. Отнесение значений к областям данных 219
4.8. Итоги и перспективы 222
Замечания к этой главе 222
Упражнения 223

Глава 5. Синтаксически управляемая трансляция 227
5.1. Введение 227
5.2. Предварительные сведения 230
5.3. Синтаксически управляемая трансляция 232
5.3.1. Первый пример 232
5.3.2. Трансляция выражений 235
5.3.3. Трансляция управляющих конструкций 242
5.4. Моделирование окружения именования 245
5.4.1. Лексические иерархии 246
5.4.2. Иерархии наследования 252
5.4.3. Видимость 256
5.4.4. Разрешение имен на этапе компиляции 257
5.5. Информация о типе 258
5.5.1. Применения типов при трансляции 259
5.5.2. Компоненты системы типов 261
5.5.3. Вывод типа выражения 266
5.6. Размещение в памяти 270
5.6.1. Классы памяти и области данных 270
5.6.2. Размещение в виртуальном адресном пространстве 273
5.6.3. Назначение памяти 275
5.6.4. Место назначения памяти в процессе трансляции 280
5.6.5. Ограничения выравнивания и дополнение 280
5.7. Дополнительные темы 282
5.7.1. Структура грамматики и ассоциативность 282
5.7.2. Более трудные проблемы вывода типов 285
5.7.3. Относительные смещения и производительность кеша 287
5.8. Итоги и перспективы 289
Замечания к этой главе 290
Упражнения 291

Глава 6. Реализация процедур 294
6.1. Введение 294
6.2. Предварительные сведения 298
6.3. Поддержка именования на этапе выполнения 301

6.3.1. Поддержка времени выполнения в Algol-подобных языках 302
6.3.2. Поддержка объектно-ориентированных языков на этапе выполнения 308
6.4. Передача значений между процедурами 313
6.4.1. Передача параметров 313
6.4.2. Возврат значений 316
6.4.3. Обеспечение адресуемости нелокальных переменных 318
6.5. Стандартные соглашения о связях 324
6.6. Дополнительные темы 329
6.6.1. Явное управление кучей 330
6.6.2. Неявное освобождение 333
6.7. Итоги и перспективы 339
Замечания к этой главе 340
Упражнения 341

Глава 7. Форма кода 346

7.1. Введение 346
7.2. Арифметические операторы 349
7.2.1. Вызовы функций в выражении 351
7.2.2. Выражения со смешанными типами 352
7.2.3. Уменьшение спроса на регистры 354
7.3. Методы доступа к значениям 356
7.3.1. Методы доступа к скалярным переменным 356
7.3.2. Методы доступа к агрегатам 359
7.3.3. Проверка диапазона 366
7.4. Булевы и реляционные операторы 368
7.4.1. Аппаратная поддержка реляционных выражений 369
7.4.2. Варианты аппаратной поддержки 371
7.5. Управляющие конструкции 374
7.5.1. Условное выполнение 375
7.5.2. Циклы и итерация 377
7.5.3. Предложения выбора 381
7.6. Операции со строками 385
7.6.1. Длина строки 385
7.6.2. Присваивание строке 385
7.6.3. Конкатенация строк 387
7.6.4. Оптимизация операций со строками 387
7.7. Вызовы процедур 388
7.7.1. Вычисление фактических параметров 389
7.7.2. Сохранение и восстановление регистров 390
7.8. Итоги и перспективы 392
Замечания к этой главе 393
Упражнения 394

Глава 8. Введение в оптимизацию 398

8.1. Введение 398

8.2. Предварительные сведения ... 400
8.2.1. Примеры ... 402
8.2.2. Соображения по поводу оптимизации ... 406
8.2.3. Возможности для оптимизации ... 409
8.3. Область оптимизации ... 410
8.4. Локальная оптимизация ... 414
8.4.1. Локальная нумерация значений ... 414
8.4.2. Балансировка дерева по высоте ... 421
8.5. Региональная оптимизация ... 430
8.5.1. Суперлокальная нумерация значений ... 431
8.5.2. Разворачивание цикла ... 434
8.6. Глобальная оптимизация ... 437
8.6.1. Нахождение неинициализированных переменных с помощью анализа живучести ... 438
8.6.2. Глобальное размещение кода ... 443
8.7. Межпроцедурная оптимизация ... 450
8.7.1. Встраивание ... 450
8.7.2. Размещение процедуры в памяти ... 454
8.7.3. Прагматические аспекты межпроцедурной оптимизации ... 458
8.8. Итоги и перспективы ... 461
Замечания к этой главе ... 462
Упражнения ... 464

Глава 9. Анализ потока данных ... 468

9.1. Введение ... 468
9.2. Итеративный анализ потока данных ... 471
9.2.1. Доминирование ... 471
9.2.2. Анализ живучести переменных ... 476
9.2.3. Ограничения на анализ потока данных ... 480
9.2.4. Другие задачи о потоке данных ... 483
9.3. Форма с единственным статическим присваиванием ... 488
9.3.1. Наивный метод построения SSA-формы ... 490
9.3.2. Границы доминирования ... 492
9.3.3. Размещение φ-функций ... 495
9.3.4. Переименование ... 499
9.3.5. Трансляция программы в форме SSA ... 505
9.3.6. Использование формы SSA ... 513
9.4. Межпроцедурный анализ ... 517
9.4.1. Построение графа вызовов ... 518
9.4.2. Межпроцедурное распространение констант ... 521
9.5. Дополнительные темы ... 525
9.5.1. Структурный анализ потока данных и приводимость ... 525
9.5.2. Ускорение итеративного каркаса доминирования ... 529
9.6. Итоги и перспективы ... 531
Замечания к этой главе ... 532
Упражнения ... 533

Глава 10. Скалярная оптимизация 537
10.1. Введение 537
10.2. Устранение мертвого кода 542
10.2.1. Устранение бесполезного кода 543
10.2.2. Устранение бесполезного потока управления 545
10.2.3. Устранение недостижимого кода 549
10.3. Перемещение кода 550
10.3.1. Отложенное перемещение кода 550
10.3.2. Поднятие кода 558
10.4. Специализация 560
10.4.1. Оптимизация хвостового вызова 560
10.4.2. Оптимизация листового вызова 562
10.4.3. Перенос параметров 563
10.5. Устранение избыточности 564
10.5.1. Идентичность значений и идентичность имен 565
10.5.2. Нумерация значений на основе доминирования 566
10.6. Обеспечение других преобразований 569
10.6.1. Клонирование суперблока 570
10.6.2. Клонирование процедуры 572
10.6.3. Расчленение цикла 572
10.6.4. Переименование 573
10.7. Дополнительные темы 575
10.7.1. Комбинирование оптимизаций 575
10.7.2. Снижение стоимости 580
10.7.3. Выбор последовательности оптимизаций 590
10.8. Итоги и перспективы 592
Замечания к этой главе 592
Упражнения 593

Глава 11. Выбор команд 596
11.1. Введение 596
11.2. Предварительные сведения 600
11.2.1. Влияние дизайна ISA на выбор команд 601
11.2.2. Поясняющий пример 605
11.2.3. Ситуативное сопоставление 607
11.3. Выбор команд с помощью оконной оптимизации 608
11.3.1. Оконная оптимизация 608
11.3.2. Упроститель 610
11.3.3. Сопоставитель 614
11.4. Выбор с помощью сопоставления с деревом-образцом 616
11.4.1. Представление деревьев 617
11.4.2. Правила переписывания 618
11.4.3. Вычисление покрытий 622
11.4.4. Инструменты 630
11.5. Дополнительные темы 631
11.5.1. Обучение оконным образцам 631

11.5.2. Генерирование последовательностей команд 633
11.6. Итоги и перспективы 634
Замечания к этой главе 635
Упражнения 636

Глава 12. Планирование команд 638

12.1. Введение 638
12.2. Предварительные сведения 641
12.2.1. Архитектурные средства, влияющие на производительность 642
12.2.2. Задача планирования команд 645
12.3. Локальное планирование 649
12.3.1. Алгоритм 650
12.3.2. Переименование 651
12.3.3. Построение графа зависимостей 653
12.3.4. Вычисление приоритетов 655
12.3.5. Планирование списка 656
12.3.6. Прямое и обратное планирование списка 660
12.4. Региональное планирование 663
12.4.1. Суперлокальное планирование 663
12.4.2. Планирование трасс 665
12.4.3. Клонирование с учетом контекста 668
12.5. Дополнительные темы 669
12.5.1. Стратегия организации программного конвейера 670
12.5.2. Алгоритм программного конвейера 673
12.5.3. Последний пример 677
12.6. Итоги и перспективы 679
Замечания к этой главе 679
Упражнения 681

Глава 13. Распределение регистров 684

13.1. Введение 684
13.2. Предварительные сведения 687
13.2.1. Пространство имен для распределения: интервалы жизни переменных 687
13.2.2. Интерференция 690
13.2.3. Код выгрузки 692
13.2.4. Классы регистров 693
13.3. Локальное распределение регистров 696
13.3.1. Переименование в локальном распределителе 698
13.3.2. Распределение и назначение 700
13.4. Глобальное распределение с помощью раскрашивания 704
13.4.1. Нахождение глобальных интервалов жизни 706
13.4.2. Построение графа интерференции 708
13.4.3. Консолидация операций копирования 710
13.4.4. Оценка глобальной стоимости выгрузки 712
13.4.5. Раскрашивание графа 714

13.4.6. Вставка кода выгрузки и восстановления ... 717
13.4.7. Обработка совмещенных классов регистров ... 718
13.5. Дополнительные темы ... 723
13.5.1. Вариации на тему консолидации ... 723
13.5.2. Вариации на тему выгрузки ... 725
13.5.3. Другие формы интервалов жизни ... 727
13.6. Итоги и перспективы ... 732
Замечания к этой главе ... 732
Упражнения ... 733

Глава 14. Оптимизация на этапе выполнения ... 737

14.1. Введение ... 737
14.2. Предварительные сведения ... 741
14.2.1. Модель выполнения ... 743
14.2.2. Триггеры компиляции ... 745
14.2.3. Гранулярность оптимизации ... 747
14.2.4. Источники улучшения ... 749
14.2.5. Построение оптимизатора этапа выполнения ... 752
14.3. Оптимизация горячих трасс ... 753
14.3.1. Поток выполнения ... 754
14.3.2. Связывание трасс ... 759
14.4. Оптимизация горячих методов ... 762
14.4.1. Горячие методы в смешанной среде ... 762
14.4.2. Горячие методы в платформенной среде ... 769
14.5. Дополнительные темы ... 772
14.5.1. Уровни оптимизации ... 773
14.5.2. Замещение в стеке ... 773
14.5.3. Управление кешем методов ... 775
14.5.4. Управление изменениями исходного кода ... 777
14.6. Итоги и перспективы ... 779
Замечания к этой главе ... 779
Упражнения ... 780

Приложение A. ILOC ... 783

A.1. Введение ... 783
A.2. Соглашения об именовании ... 785
A.3. Вычислительные операции ... 785
A.4. Операции перемещения данных ... 786
A.5. Операции потока управления ... 788
A.6. Таблицы кодов операций ... 791

Приложение B. Структуры данных ... 794

B.1. Введение ... 794
B.2. Представление множеств ... 795
B.2.1. Представление множеств упорядоченными списками ... 796
B.2.2. Представление множеств битовыми векторами ... 798

B.2.3. Представление разреженных множеств 799
B.2.4. Роль хеш-таблиц 800
B.3. Реализация ПП 801
B.3.1. Непосредственные графовые представления 801
B.3.2. Линейные промежуточные формы 806
B.4. Реализация хеш-таблиц 809
B.4.1. Выбор хеш-функции 809
B.4.2. Метод цепочек 810
B.4.3. Метод открытой адресации 812
B.4.4. Хранение записей символов 813
B.5. Гибкий дизайн таблицы символов 814
Замечания 816

Литература 818

Предметный указатель 845

От издательства

Отзывы и пожелания

Мы всегда рады отзывам наших читателей. Расскажите нам, что вы думаете об этой книге – что понравилось или, может быть, не понравилось. Отзывы важны для нас, чтобы выпускать книги, которые будут для вас максимально полезны.

Вы можете написать отзыв на нашем сайте *www.dmkpress.com*, зайдя на страницу книги и оставив комментарий в разделе «Отзывы и рецензии». Также можно послать письмо главному редактору по адресу *dmkpress@gmail.com*; при этом укажите название книги в теме письма.

Если вы являетесь экспертом в какой-либо области и заинтересованы в написании новой книги, заполните форму на нашем сайте по адресу *http://dmkpress.com/authors/publish_book/* или напишите в издательство по адресу *dmkpress@gmail.com*.

Список опечаток

Хотя мы приняли все возможные меры для того, чтобы обеспечить высокое качество наших текстов, ошибки все равно случаются. Если вы найдете ошибку в одной из наших книг, мы будем очень благодарны, если вы сообщите о ней главному редактору по адресу *dmkpress@gmail.com*. Сделав это, вы избавите других читателей от недопонимания и поможете нам улучшить последующие издания этой книги.

Нарушение авторских прав

Пиратство в интернете по-прежнему остается насущной проблемой. Издательство «ДМК Пресс» очень серьезно относится к вопросам защиты авторских прав и лицензирования. Если вы столкнетесь в интернете с незаконной публикацией какой-либо из наших книг, пожалуйста, пришлите нам ссылку на интернет-ресурс, чтобы мы могли применить санкции.

Ссылку на подозрительные материалы можно прислать по адресу электронной почты *dmkpress@gmail.com*.

Мы высоко ценим любую помощь по защите наших авторов, благодаря которой мы можем предоставлять вам качественные материалы.

Об авторах

Кит Д. Купер – профессор вычислительной техники в университете Райса. Занимался широким кругом проблем в области оптимизации откомпилированного кода, включая анализ межпроцедурного потока данных и его приложения, нумерацию значений, алгебраическую реассоциацию выражений, распределение регистров и планирование выполнения команд. Он читал самые разные курсы, в т. ч. введение в программирование, алгоритмическое мышление, организацию компьютеров, коллективное программирование, несколько вариантов курса по конструированию компиляторов и курс по оптимизации кода для аспирантов. Исполнял должности заведующего кафедрой информатики в университете Райса, заведующего кафедрой вычислительной и прикладной математики там же и заместителя декана по исследованиям в инженерной школе при университете Райса. Является членом Ассоциации вычислительной техники (ACM).

Линда Торкзон в течение долгого времени работала старшим научным сотрудником кафедры информатики в университете Райса. Занималась широким кругом задач в области компиляции, включая анализ межпроцедурного потока данных, подстановку встраиваемых функций, распределение регистров, адаптивную оптимизацию и проектирование сред программирования. Несколько раз читала курс по компиляторам для студентов младших курсов в университете Райса. Д-р Торкзон занимала должность исполнительного директора Центра исследований по параллельным вычислениям при Национальном центре фундаментальных наук и технологий. Она также работала исполнительным директором компании HiPerSoft, Лос-Аламосского института информатики и проекта по разработке приложений виртуальных сеток. Была главным исследователем в проекте разработки адаптируемых сред компиляции, спонсируемым агентством DARPA, целью которого являлось проектирование оптимизирующего компилятора, который мог бы автоматически подстраивать свои алгоритмы и стратегии оптимизации под новые платформы. Также занимала должность директора программы присвоения степени мастера-профессионала по информатике в университете Райса.

Предисловие

Изменения в третьем издании

В третье издание книги «Разработка компилятора» (EaC3e) изменения вносились по двум главным причинам: новые способы использования технологии трансляции языков программирования и изменившаяся техническая подготовка наших студентов. Эти две движущие силы побуждали нас постоянно пересматривать способы преподавания методов конструирования компиляторов. В третье издание включены эти изменения, протестированные на аудитории.

В издании EaC3e материал организован по-новому: некоторые темы, присутствовавшие в предыдущих изданиях, исключены, а вместо них добавлены новые – по крайней мере в книге. В издание EaC2e уже были включены важные изменения материалов по оптимизации (главы 8, 9 и 10), и эти главы перешли в третье издание почти без изменений. Для EaC3e изменения были внесены в разные места текста, но основной упор был сделан на реорганизацию и переработку средней части книги: главы с 4 по 7. Широкое использование JIT-компиляторов побудило нас добавить главу, посвященную введению в эти методы. Конкретно:

- материал о промежуточных представлениях перемещен в главу 4, перед синтаксически управляемой трансляцией. Студенты должны ознакомиться с этим материалом до изложения синтаксически управляемой трансляции;
- материал об атрибутных грамматиках, который в двух предыдущих изданиях находился в главе 4, исключен. Атрибутные грамматики – по-прежнему интересная тема, но на данный момент преобладающей парадигмой в фазе анализа компилятора является ситуативная трансляция;
- в главе 5 теперь более глубоко излагается механизм синтаксически управляемой трансляции и его применение. В эту главу перенесено несколько относящихся к трансляции тем, которые раньше находились в главах 4, 6 и 7;
- глава 7 получила новую организацию и структуру, основанную на интенсивных экспериментах в учебной аудитории;
- в главе 13 основное внимание уделяется двум распределителям: локальному, основанному на алгоритме Бэста, и глобальному, основанному на работе Хайтина. В разделе «Дополнительные темы» главы 13 изучаются модификации базовой схемы Хайтина–Бриггса, порождающие такие методы, как распределители на основе линейного сканирования, распределители на основе SSA и итеративные объединяющие распределители;

В предыдущих изданиях алгоритм Бэста назывался «локальным восходящим» алгоритмом.

- в главе 14 теперь представлен обзор оптимизации на этапе выполнения, или JIT-компиляции. JIT-компиляторы теперь можно встретить всюду. Студенты должны понимать, как работают JIT-компиляторы и как они связаны с классической компиляцией перед выполнением.

Нашей целью по-прежнему является учебник и курс, на котором студенты знакомятся с важнейшими проблемами современных компиляторов и получают базовые знания, позволяющие справляться с этими проблемами.

Организация

В третьем издании материал разбит на четыре примерно одинаковых по размеру части.

- Первая часть, главы 2 и 3, посвящена проектированию фазы анализа компилятора (front end) и алгоритмам, на которых основаны инструментальные средства, автоматизирующие конструирование фазы анализа. В процессе описания алгоритмов, которые выводят сканеры из регулярных выражений, а парсеры – из контекстно-свободных грамматик, вводятся некоторые ключевые идеи, в т. ч. понятие алгоритма с неподвижной точкой.
- Во второй части (главы 4–7) изучается преобразование исходного кода в промежуточную форму. Здесь рассматриваются виды кода, которые фаза анализа может генерировать для оптимизатора и фазы синтеза (back end).
- В третьей части, в которую входят главы 8, 9, 10 и 14, приводится обзор методов оптимизации кода. В главе 8 представлен общий взгляд на виды оптимизации, выполняемые компилятором. В главах 9 и 10 более глубоко изучаются анализ потока данных и скалярная оптимизация. Глава 14 тематически примыкает к этой части, но в ней предполагается знание материала, составляющего четвертую часть.

> Обычно из программы для студентов младших курсов мы опускаем главы 9 и 10. Наш опыт показывает, что глава 14 вызывает у студентов больший интерес.

- Четвертая часть, главы 11–13, посвящена основным алгоритмам, встречающимся в фазе синтеза: выбор команд, планирование выполнения команд и распределение регистров. В третьем издании мы переработали материал по распределению регистров, так что теперь количество идей уменьшилось, но те, что остались, рассматриваются глубже. Благодаря новой главе студенты получают основательные знания, позволяющие понять большинство современных алгоритмов распределения.

Программа для студентов младших курсов представляет собой линейную выжимку из этого материала. За недостатком времени мы часто опускаем главы 9 и 10. Материал главы 14 был подготовлен в ответ на вопросы слушателей курса.

Подход

Конструирование компилятора – упражнение из области инженерного проектирования. Автор компилятора должен выбрать путь в пространстве проектирования, изобилующем различными альтернативами, каждая из которых характеризуется стоимостью, набором преимуществ и сложностью. Любое решение оказывает влияние на результирующий компилятор. Качество конечного продукта зависит от решений, принятых на каждом шаге.

Поэтому часто не существует единственно верного ответа на вопрос о том, какое из множества решений выбрать при проектировании компилятора. Даже для «хорошо проработанных» и «решенных» проблем есть нюансы дизайна и реализации, которые влияют как на поведение компилятора, так и на качество порождаемого им кода. При принятии решения учитываются различные соображения. Например, выбор промежуточного представления оказывает значительное влияние на остальные части компилятора – от требований к памяти и времени работы до легкости применения различных алгоритмов. Однако на принятие решения часто не хватает времени. В четвертой части рассматривается вопрос о промежуточных представлениях и некоторые проблемы, которые следует учитывать при выборе представления. Этот вопрос поднимается еще в нескольких местах книги – как прямо в тексте, так и опосредованно в упражнениях.

В третьем издании исследуется все пространство проектирования компилятора и уделяется внимание как глубине проблем, так и широте охвата возможных решений. Описываются некоторые способы разрешения проблем компиляции, а также ограничения, из-за которых эти решения стали привлекательными. Авторы компиляторов должны понимать параметры проблем и их решения. Они должны сознавать, что решение одной проблемы может повлиять на спектр возможностей и ограничений в других частях компилятора. И только в этом случае они смогут сделать обоснованный и разумный выбор.

Философия

Этот учебник является воплощением нашей философии построения компиляторов, которая сложилась в результате сорока с лишним лет исследований, преподавания и практики. Например, промежуточные представления должны раскрывать те детали, которые имеют значение в конечном коде; это убеждение побуждает нас отдавать предпочтение низкоуровневым представлениям. Значения должны храниться в регистрах до тех пор, пока распределитель не обнаружит, что регистров не хватает; такая практика приводит к компиляторам, оперирующим в терминах виртуальных регистров, которые хранят значения в памяти, только если этого невозможно избежать. Данный подход также повышает важность эффективных алгоритмов на фазе синтеза. Каждый компилятор должен включать этап оптимизации, это упро-

щает остальные части компилятора. При отборе материала и способа его изложения мы руководствовались своим многолетним опытом.

Несколько слов об упражнениях

Курс по конструированию компиляторов дает возможность исследовать сложные проблемы в контексте конкретного приложения, основные функции которого понятны любому студенту, обладающему достаточной подготовкой для посещения курса по компиляторам. В большинстве вариантов такого курса упражнения на программирование играют важную роль.

Мы читали этот курс в вариантах, когда студенты создают простой компилятор от начала до конца – начиная со сгенерированных сканера и парсера и заканчивая кодогенератором для упрощенного набора RISC-команд. Мы также читали этот курс в вариантах, когда студенты пишут программы для решения четко поставленных узких задач, таких как распределение регистров или планирование команд. Выбор упражнений сильно зависит от роли, которую данный курс играет в общем учебном плане.

В некоторых учебных заведениях курс по компиляторам является кульминацией для студентов старших курсов, в нем идеи, с которыми слушатели познакомились на других курсах, соединяются в крупном практическом проекте, подразумевающем как проектирование, так и реализацию. Слушатели такого курса могут написать полный компилятор какого-нибудь простого языка или модифицировать компилятор с открытым исходным кодом, добавив в него какое-то языковое средство или новую архитектурную возможность. В таком варианте курса материал может излагаться линейно, близко к организации текста в учебнике.

В других вузах кульминация может достигаться в иных курсах или иными способами. В таком случае преподаватель может уделить больше внимания упражнениям, относящимся к конкретным алгоритмам или их реализациям; например, на семинарах можно разработать распределитель регистров или проход перебалансировки дерева с выравниванием высоты. В этом варианте курса можно опускать некоторые представленные в учебнике темы и изменять порядок изложения, подстраивая его под нужды семинаров. Мы обнаружили, что студенты, посещающие этот курс, умеют программировать на языке ассемблера, поэтому не затрудняются с пониманием того, как должен работать планировщик команд или распределитель регистров.

При любом сценарии курс должен иметь связи с другими курсами в учебном плане для младших курсов. Очевидно, он связан с курсами по организации компьютера, программированию на языке ассемблера, операционным системам, компьютерной архитектуре, алгоритмам и формальным языкам. Но немало и менее очевидных связей. Материал главы 7 о копировании литер поднимает вопросы производительности, которые критически важны для реализации стека протоколов. Материал главы 2 имеет разнообразные применения: от фильтрации URL-адресов до задания правил для брандмау-

эров и маршрутизаторов. И конечно, алгоритм Бэста из главы 13 – это более ранняя версия алгоритма Белади офлайнового замещения страниц кеша, MIN.

Дополнительные материалы

Для адаптации представленного в книге материала к вашему курсу имеются дополнительные ресурсы: полный комплект слайдов для авторского варианта курса, читаемого в университете Райса, и решения всех упражнений. Для получения дополнительной информации заходите на сайт *https://educate.elsevier.com/book/details/9780128154120*.

Благодарности

В подготовке третьего издания книги «Разработка компилятора» участвовало много народу. Члены сообщества вежливо указали на несогласованности, типографические недочеты и ошибки. Всем им мы выражаем благодарность.

Преподавание часто само по себе является наградой. Двое наших коллег, Зоран Будимилич и Майкл Бёрк, заслуживают отдельной благодарности. Зоран – гений в искусстве абстрагирования проблем. Майкл глубоко понимает теорию, стоящую за частями книги, посвященными фазам анализа и оптимизации. Каждый из них повлиял на наше осмысление части материала.

Коллектив издательства Elsevier, в особенности Бэт Логиудиче, Стив Меркен и Манчу Мохан, сыграл важную роль в преобразовании рукописи в окончательную форму. Все они значительно улучшили книги своими идеями и помощью. Аарон Кин, Майкл Лэм и другие рецензенты поделились с нами ценными и своевременными замечаниями о главе 14.

Наконец, многие люди интеллектуально и эмоционально поддерживали нас на протяжении пяти последних лет. Прежде всего отдаем должное нашим семьям и коллегам по университету Райса, которые воодушевляли нас на каждом шаге пути. Кристина и Каролина в особенности вытерпели бесчисленное множество нескончаемых дискуссий по отбору тем для курса по конструированию компиляторов. Стив Меркен руководил работой по подготовке этого издания от идеи для публикации, демонстрируя энтузиазм, бесконечное терпение и отличное чувство юмора. Всем этим людям мы выражаем сердечную благодарность.

Глава 1

Обзор процесса компиляции

АННОТАЦИЯ

Компилятор – это компьютерная программа, которая транслирует программу, написанную на одном языке, в программу, написанную на другом языке. В то же время компилятор – это крупная программная система с большим числом внутренних компонентов и алгоритмов. Эти компоненты сложными способами взаимодействуют между собой. Изучение конструирования компилятора является одновременно введением в методы трансляции и улучшения программ и практическим упражнением по применению программной инженерии. В этой главе приводится концептуальный обзор всех основных компонентов современного компилятора.

КЛЮЧЕВЫЕ СЛОВА

Компилятор, интерпретатор, автоматическая трансляция.

1.1. Введение

Роль компьютеров в повседневной жизни растет с каждым годом. Компьютеры критически важны в индустрии развлечений, связи, транспорте, в медицинских приборах, в проектировании, производстве и даже в таких бытовых устройствах, как домашние терморегуляторы. Компьютеры создали целые новые виды деятельности, от видеоигр до социальных сетей. Они изменили методы прогнозирования погоды и способы получения новостей. Встраиваемые компьютеры синхронизируют работу светофоров и доставляют электронную почту в ваше мобильное устройство.

Компилятор
компьютерная программа, которая транслирует другие компьютерные программы.

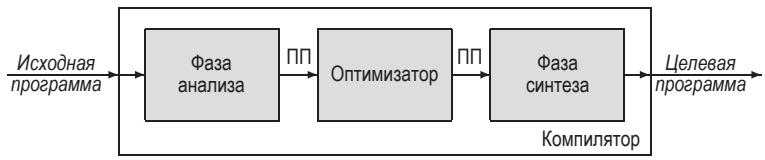
Употребляемые в тексте термины «код» и «программа» означают либо некоторую часть программы, либо всю программу. Что имеется в виду, должно быть понятно из контекста.

Почти все компьютерные приложения зависят от программного обеспечения – программ, которые создают виртуальные инструменты поверх низкоуровневых абстракций, предоставляемых оборудованием.

Почти все эти программы транслируются *компилятором*. Компилятор – это просто компьютерная программа, которая транслирует другие программы, подготавливая их к выполнению. В этой книге представлены основные методы автоматической трансляции, которые применяются при конструировании компиляторов. Описаны многие проблемы, с которыми сталкиваются авторы компиляторов, и алгоритмы, которыми они пользуются, чтобы их преодолеть.

Концептуальная дорожная карта

Компилятор транслирует программу, написанную на одном языке, на другой язык. Чтобы перевести текст с одного языка на другой, инструмент должен понимать как форму, или синтаксис, так и содержание, или семантику, входного языка. Он должен понимать правила, описывающие синтаксис и семантику выходного языка. Наконец, ему необходима схема отображения содержимого с входного языка на выходной.



Структура типичного компилятора вытекает из этих простых наблюдений. У компилятора имеется фаза анализа, предназначенная для обработки входного языка. Имеется также фаза синтеза, предназначенная для генерирования кода на выходном языке. Имеется структура, соединяющая фазы анализа и синтеза и служащая для хранения промежуточного представления (ПП) программы; его семантика мало зависит от обоих языков. Большинство компиляторов включают оптимизатор, расположенный между фазами анализа и синтеза; он анализирует ПП и переписывает его в форме, которая по одному или нескольким параметрам лучше.

Краткий обзор

Компьютерные программы – это последовательности абстрактных операций, записанные на *языке программирования* – формальном языке, предназначенном для выражения вычислений. У языков программирования имеются строгие свойства и семантика – в отличие от естественных языков, таких как китайский, португальский или английский. Языки программирования проектируются с желанием добиться выразительности, краткости и ясности. Естественные языки допускают неоднозначность. Языки программирования никакой неоднозначности не допускают, поскольку неоднозначная программа могла бы иметь несколько интерпретаций. Язык программирования должен точно описывать вычисления; неоднозначность вступала бы в противоречие с точностью.

Вообще говоря, языки программирования проектируются так, чтобы человек мог выразить вычисления в виде последовательности операций. Компьютерные процессоры, которые мы далее будем называть процессорами, микропроцессорами или машинами, предназначены для выполнения последовательностей операций. Операции, выполняемые процессорами, обычно находятся на гораздо более низком уровне абстракции, чем описываемые на языке программирования. Например, язык программирования обычно включает краткий способ записать число в файл. Это единственное предложение языка программирования транслируется в сотни или тысячи машинных операций, исполняемых процессором.

Инструмент, выполняющий такую трансляцию, называется компилятором. Компилятор принимает входную программу на некотором языке и порождает на выходе эквивалентную программу. Компилятор в классическом понимании порождает выходную программу, выраженную в терминах операций некоторого конкретного процессора, который часто называют целевой машиной. Если рассматривать компилятор как черный ящик, то картина может выглядеть так:



Система команд
Множество операций, поддерживаемых процессором, называется его *системой команд*.

Общий дизайн системы команд часто называют *архитектурой системы команд* (*англ.* instruction set architecture, ISA).

К числу типичных «исходных» языков относятся C, C++, JAVA, ML и RUBY. «Целевым» языком обычно является система команд некоторого процессора.

Некоторые компиляторы порождают целевую программу на языке программирования, ориентированном на человека, а не на языке ассемблера какого-то процессора. Порождаемые такими компиляторами программы должны быть подвергнуты дополнительной трансляции, прежде чем смогут быть выполнены компьютером. Многие исследовательские компиляторы порождают на выходе программы на языке C. Поскольку компиляторы с C имеются для большинства компьютеров, целевая программа на всех таких компьютерах оказывается исполняемой, правда, ценой дополнительной компиляции в окончательную форму. Компиляторы, для которых целевым является другой язык программирования, а не система команд компьютера, называются *транспиляторами*.

Многие другие системы можно рассматривать как компиляторы. Например, наборную программу, генерирующую код на языке PostScript, можно считать компилятором. Она принимает на входе спецификацию внешнего вида документа на печатной странице, а на выходе порождает PostScript-файл. PostScript – это не что иное, как язык описания изображений. Посколь-

ку наборная программа принимает исполняемую спецификацию и порождает другую исполняемую спецификацию, это компилятор.

Код, преобразующий PostScript в пиксели, обычно является *интерпретатором*, а не компилятором. Интерпретатор принимает на входе исполняемую спецификацию, а его выходом является результат исполнения этой спецификации.



Некоторые языки, например PERL, PYTHON, RUST и SCHEME, чаще реализуются в виде интерпретаторов, а не компиляторов.

Исторически большинство компиляторов проектировались как отдельный шаг, предшествующий выполнению. Исключения были, но *упреждающая* (ahead-of-time – AOT) компиляция была правилом. Однако в последние годы появились языки программирования и системы, в которых имело смысл откладывать компиляцию. Эти системы, называемые оптимизаторами времени выполнения, или просто *своевременными* (just-in-time – JIT) *компиляторами*, транслируют код в исполняемую форму на этапе выполнения. Такая стратегия добавляет стоимость JIT-компиляции ко времени выполнения программы, поэтому время, потраченное на работу JIT-компилятора, должно быть компенсировано повышением производительности.

Упреждающий (AOT) компилятор
традиционный компилятор, в котором трансляция предшествует выполнению.

Своевременный (JIT) компилятор
компилятор, в котором трансляция производится на этапе выполнения.

JIT-компиляция может усложнить схему трансляции. Например, модель выполнения JAVA включает как компиляцию, так и интерпретацию. Результатом компиляции исходного JAVA-кода является более компактное представление, называемое *байт-кодом*; его порождает AOT-компилятор. Исполнение приложения сводится к выполнению байт-кода на виртуальной машине JAVA (JVM), которая является эмулятором, или интерпретатором, байт-кода. Для повышения производительности многие реализации JVM включают JIT-компилятор, который компилирует в машинную форму и оптимизирует часто используемые последовательности байт-кода. В главе 14 изучаются некоторые проблемы, возникающие в процессе построения JIT-компилятора.

Виртуальная машина
Виртуальная машина – это эмулятор ISA идеализированного процессора.

У интерпретаторов и компиляторов много общего. Их задачи во многом одинаковы. Те и другие анализируют входную программу и определяют, корректна она или нет. Те и другие строят внутреннюю модель структуры и семантики программы. Те и другие решают, где хранить значения во время

выполнения. Однако результат, порождаемый в процессе интерпретации кода, принципиально отличен от вывода оттранслированной программы, которую можно выполнить, чтобы получить результат. В этой книге обсуждаются проблемы, возникающие при построении компиляторов. Но и авторы интерпретаторов смогут найти в ней много полезного.

Зачем изучать конструирование компиляторов?

Конечно, авторы считают, что материал, излагаемый в курсе по конструированию компиляторов, сам по себе завораживает. Но студентам часто нужны более конкретные причины, оправдывающие потраченное время. Мы видим два разных ответа: педагогический и практический.

Для студента конструирование компилятора – кульминация, которая сводит воедино элементы из разных разделов информатики с целью применить их к большому проекту, включающему проектирование и реализацию. Хороший компилятор на практике использует жадные алгоритмы (распределение регистров), методы эвристического поиска (планирование списка), алгоритмы на графах (исключение недостижимого кода), динамическое программирование (выбор команд), теорию автоматов (лексический и грамматический разборы) и алгоритмы с неподвижной точкой (анализ потока данных). Он решает такие задачи, как динамическое распределение, синхронизация, именование, локальность, управление иерархией памяти и планирование конвейера. Мало найдется программных систем, содержащих одновременно так много сложных и разнородных компонентов.

Вам предстоит изучать компиляторы в интересное время

В 1951 году Грейс Мюррей Хоппер написала программу A0, которую некоторые считают первым компилятором. К началу 1970-х годов компиляторы умели генерировать высококачественный код для тогдашних машин. Почему же тогда конструирование компиляторов до сих пор является предметом изучения и исследований? Изменения языков и вычислительных сред ставят ряд проблем, которые заставляют авторов компиляторов адаптировать сложившиеся практики для достижения наилучших результатов.

Прекращение роста производительности в однопоточном режиме, провозглашаемое концом закона масштабирования Деннарда и закона Мура, побудило конструкторов систем сосредоточиться на улучшениях архитектуры и распараллеливании. Эти изменения, в свою очередь, ставят новые вызовы перед оптимизацией и генерированием кода.

Включение в языки таких средств, как объектная ориентированность, поздняя привязка имен к типам и классам, динамическая загрузка и полиморфизм, не обошлось без новых накладных расходов. Эти средства открывают новые возможности для анализа и оптимизации программ с целью уменьшить данные расходы.

Наконец, откомпилированный код находит применение в новых ситуациях – от смартфонов до слуховых аппаратов и дверных звонков – и в новых режимах использования – от JIT-компиляции до разделяемых кешей кода. По мере того как вычисления распространяются в новых направлениях, цели и ограничения компиляции изменяются, что несет с собой новые проблемы и новые решения.

Компиляторы демонстрируют успешное применение теории к практике. Инструменты, разработанные для генерирования сканеров и парсеров, являют пример применения результатов, полученных в рамках формальной теории языков. Те же самые инструменты используются при поиске текста, в фильтрах сайтов и в текстовых процессорах; они находят применение при написании языков разметки, скриптовых языков и спецификаций для маршрутизаторов. В проверке типов и статическом анализе результаты из теории решеток, теории чисел и других разделов математики применяются с целью понять и улучшить программы. В кодогенераторах алгоритмы сопоставления с древовидным образцом, синтаксического разбора, динамического программирования и поиска в тексте применяются для автоматизации выбора команд.

Для построения успешного компилятора необходимы опыт, инженерное мастерство и планирование. В хороших компиляторах для решения трудных проблем применяются приближенные решения. На первое место выдвигается эффективность реализации самого компилятора и генерируемого им кода. Их внутренние структуры данных и представления знаний имеют необходимый уровень детализации – не больше, не меньше, а достаточно для строгой оптимизации, но не слишком много, чтобы компилятор не погряз в деталях. Конструирование компиляторов сводит воедино идеи и методы, заимствованные из самых разных разделов информатики, и применяет их для решения ряда очень трудных задач.

Для программиста конструирование компилятора раскрывает истинную стоимость абстракций, используемых в программе, от предложений выбора до вызова процедур, от библиотек регулярных выражений до хеш-таблиц. Программист не может эффективно настроить производительность приложения, не зная сравнительной стоимости частей, из которых оно состоит. Вообще говоря, программисты должны использовать при проектировании подходящие абстракции. Затем, если производительность оказывается камнем преткновения, следует произвести измерения, установить, где код тратит время, и переделать эту конкретную часть, снизив затраты.

Однако при таком подходе программист должен понимать, как абстракции исходного кода транслируются в команды целевой машины. Например:

- сравнение строк на равенство обычно требует времени, пропорционального длине строк, тогда как сравнение целых на равенство имеет сложность $O(1)$ и выполняется быстро. Поэтому всюду, где возможно, имеет смысл хешировать строки в небольшие целые числа;
- рассмотрим реализацию преобразования $a = f(x)$ с помощью хеш-таблицы и простого вектора. Сложность обоих вариантов, вероятно, равна $O(1)$. Но разница в стоимости может быть десятикратной и даже больше. Имеет это значение или нет, зависит от разреженности области значений f и от того, как часто f вызывается.

Изучение компиляции поможет программисту понять многие фундаментальные компромиссы, возникающие в процессе практической реализации.

А эти знания, в свою очередь, позволят принимать более обоснованные проектные решения.

Фундаментальные принципы компиляции

Компиляторы – большие, сложные и тщательно спроектированные программы. Хотя у многих проблем, возникающих при проектировании компиляторов, существует несколько решений и интерпретаций, есть два фундаментальных принципа, о которых любой автор компилятора должен помнить. Первый принцип непреложен:

Компилятор обязан сохранять семантику входной программы.

Корректность – фундаментальный вопрос программирования. Компилятор должен в точности реализовать «смысл» поданной ему на вход программы. Этот принцип лежит в основе общественного договора между автором и пользователем компилятора. Если компилятор вообще может допускать вольности в интерпретации, то почему бы ему просто не сгенерировать команду `nop` или `return`? Если некорректная трансляция в принципе допустима, то зачем тратить силы на достижение корректности?

Второй принцип, который должен соблюдать компилятор, чисто практический:

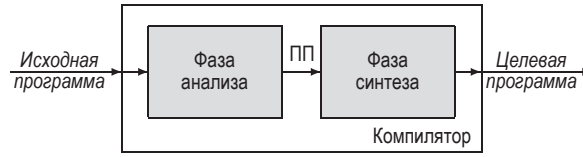
Компилятор должен ощутимо улучшать входную программу.

Традиционный компилятор улучшает входную программу, делая ее непосредственно исполнимой на некоторой целевой машине. Другие «компиляторы» улучшают свой вход иными способами. Например, `tpic` – программа, которая принимает спецификацию чертежа на графическом языке `pic` и преобразует ее в LaTeX; «улучшением» является большая доступность и общность LaTeX-представления. Транспилятор, транслирующий PYTHON на C, должен генерировать выходную программу, которая порождает тот же выход, что исходная; выходная программа должна выполняться гораздо быстрее исходного кода на PYTHON.

1.2. Структура компилятора

Компилятор – большая и сложная программная система. Сообщество создает компиляторы начиная с 1951 года и выучило много уроков о том, как их структурировать. В реальности компилятор сложнее простого черного ящика, показанного выше.

По сути дела, компилятор должен понимать исходную программу, которая подана ему на вход, и отображать ее функциональность на целевую машину. Поскольку природа двух этих задач различна, необходимо разделение труда, что ведет к дизайну, при котором компиляция разбивается на две большие части: *фазу анализа* и *фазу синтеза*.



Назначение фазы анализа – понять программу на исходном языке, а фазы синтеза – отобразить программу на целевую машину. Это разделение обязанностей имеет несколько важных следствий для проектирования и реализации компиляторов.

Фаза анализа должна закодировать знания об исходной программе в некоторой структуре, *промежуточном представлении* (ПП), для дальнейшего использования. ПП становится определяющим представлением компилируемого кода. В каждый момент процесса у компилятора будет определяющее представление. На самом деле он может использовать несколько разных ПП для разных целей, но в каждой точке одно представление будет определяющим. Мы рассматриваем определяющее ПП как версию программы, передаваемую между независимыми этапами компилятора, как, например, ПП, передаваемое от фазы анализа фазе синтеза на рисунке выше.

Промежуточное представление
Компилятор использует набор структур данных для представления обрабатываемого кода. Эта форма называется *промежуточным представлением*, или ПП.

Несколько слов о времени

Одна из самых трудных проблем, возникающих при изучении конструирования компиляторов, – *время*, точнее отслеживание моментов, когда происходят те или иные вещи. Некоторые решения принимаются на *этапе проектирования* компилятора. Некоторые алгоритмы работают, когда компилятор собирается, т. е. на *этапе сборки*. Многие операции имеют место, когда работает сам компилятор, т. е. на *этапе компиляции*. И наконец, откомпилированный код может многократно выполняться на *этапе выполнения*.

При изучении конструирования компилятора нужно думать обо всех этих этапах: какие алгоритмы и структуры данных существуют в каждый момент и как поведение, имеющее место в один момент, планируется в другой момент.

В начале каждой главы, кроме этого введения, имеется краткое обсуждение вопроса о времени. Эти обсуждения в разделах «Несколько слов о времени» показывают, в какие промежутки времени попадают описываемые в этой главе методы.

В двухфазном компиляторе фаза анализа должна удостовериться, что исходная программа корректна, а затем преобразовать ее код в ПП. Фаза синтеза должна преобразовать ПП-программы в машинные команды и конечные ресурсы целевой машины. Поскольку фаза синтеза обрабатывает только ПП, созданное фазой анализа, она может предполагать, что ПП не содержит ни синтаксических, ни семантических ошибок.

Компилятор может произвести несколько проходов по ПП кода, прежде чем выведет целевую программу. Поскольку компилятор может анализиро-

вать код в одной фазе и использовать полученные знания в другой, такой многопроходный подход дает более качественный код, чем было бы возможно получить за один проход. Эта стратегия требует, чтобы знания, приобретенные на первом проходе, были записаны в ПП, где последующие проходы смогут их найти и использовать.

Переориентация
Задачу изменения компилятора с целью генерирования кода для нового процессора часто называют *переориентацией* компилятора.

Наконец, двухфазная структура может упростить процесс *переориентации* компилятора. Легко можно вообразить создание нескольких фаз синтеза для одной фазы анализа и таким образом породить компиляторы, принимающие один и тот же входной язык, но генерирующие программы для разных ISA. Аналогично можно вообразить фазы анализа для разных языков, порождающие одно и то же ПП и пользующиеся общей фазой синтеза. В обоих сценариях предполагается, что одно ПП может обслуживать несколько комбинаций исходного и целевого языка; на практике в ПП обычно присутствуют как языково-зависимые, так и машинно-зависимые детали.

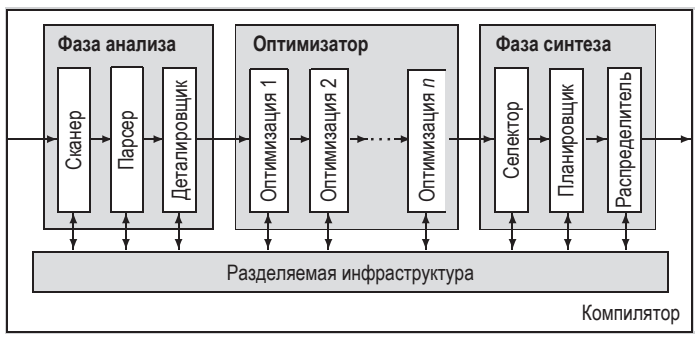
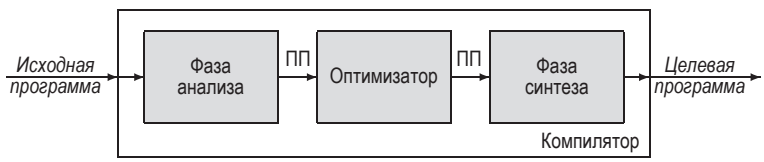


Рис. 1.1 Внутренняя структура типичного компилятора

Оптимизатор
Средняя часть компилятора, называемая *оптимизатором*, анализирует и преобразует ПП, стремясь улучшить его.

Введение ПП позволяет автору компилятора добавить фазу между анализом и синтезом. Эта средняя фаза, *оптимизатор*, пытается улучшить программу в форме ПП. Используя ПП как интерфейс, автор компилятора может вставлять оптимизации с минимальными изменениями фаз анализа и синтеза. Результатом является трехфазный компилятор.



Оптимизатор преобразует ПП в ПП или в последовательность ПП, пытаясь на каждом шаге каким-то образом улучшить ПП. Оптимизатор может со-

вершать один или несколько проходов по ПП, анализировать ПП и перезаписывать его. Оптимизатор вправе перезаписывать ПП любым способом, который позволит фазе синтеза породить более быструю или меньшую по размеру программу. У него могут быть и другие цели, например породить программу, генерирующую меньше страничных отказов или потребляющую меньше энергии.

Отдельные проходы, или преобразования оптимизатора, сами по себе являются компиляторами, согласно данному ранее определению.

Концептуально трехфазная структура соответствует классическому оптимизирующему компилятору. На практике каждая фаза подразделяется на ряд проходов. В фазе анализа можно выделить два или три прохода, которые в совокупности распознают корректные программы на исходном языке и порождают начальное ПП программы. Оптимизатор включает проходы, выполняющие различные виды анализа и преобразования с целью улучшения кода. Количество и назначение проходов зависят от компилятора. Фаза синтеза включает ряд проходов, каждый из которых приближает программу в форме ПП к системе команд целевой машины. Все три фазы и их проходы разделяют общую инфраструктуру. Эта структура показана на рис. 1.1.

Раздельная компиляция

Большинство компиляторов поддерживают *раздельную компиляцию*. Программист может компилировать файлы с кодом по отдельности, часто в разное время. Каждая такая компиляция порождает объектный файл. Несколько объектных файлов можно скомпоновать в один исполняемый файл.

Раздельная компиляция позволяет программисту ограничить объем кода, который нужно откомпилировать после изменения программы, и тем самым значительно уменьшить время компиляции. (Только представьте, что после изменения всего одной строчки в программе, содержащей миллион строк, нужно было бы перекомпилировать всю программу.) Это открывает возможность для построения и использования библиотек. Кроме того, несколько программистов могут независимо работать над одним большим приложением. Раздельная компиляция жизненно важна для построения крупномасштабных программных систем.

На практике концептуальное расчленение компилятора на три фазы – анализа, оптимизации и синтеза – создает полезное разделение обязанностей. Каждая фаза решает свои задачи. Перед фазой анализа стоит задача понять исходную программу и представить результаты анализа в форме ПП. Оптимизатор стремится улучшить ПП, так чтобы оно порождало более эффективную программу. Фаза синтеза отображает оптимизированную программу в форме ПП на ограниченные ресурсы ISA целевой машины, так чтобы использовать их наиболее эффективно.

Построение компилятора в виде последовательности независимых проходов, сообщающихся между собой посредством одного или нескольких общих ПП, дает значительное преимущество. Эта структура проще для отладки, чем монолитный компилятор. В процессе разработки автор компилятора

может вставить между проходами инструментальное средство, проверяющее корректность ПП. Автор может также удалять проходы и тестировать их независимо или тестировать подмножество проходов с целью обнаружить незапланированные и неожиданные взаимодействия.

Из этих трех фаз оптимизатор имеет самое мутное описание. Термин *оптимизация* означает, что компилятор находит оптимальное решение какой-то задачи. Но проблемы, возникающие при оптимизации кода, настолько сложны и переплетены, что на практике решить их оптимально невозможно. Ко всему прочему фактическое поведение откомпилированного кода зависит от взаимодействия всех примененных в компиляторе методов. Таким образом, даже если для какого-то одного прохода найдено оптимальное решение, из-за его взаимодействий с другими проходами конечный результат может оказаться неоптимальным. В итоге хороший оптимизирующий компилятор может улучшить качество кода по сравнению с неоптимизированной версией, но никаких гарантий оптимальности он не дает.

Оптимизатор может быть реализован как один монолитный проход, применяющий для улучшения кода одну или несколько оптимизаций. А может быть организован в виде последовательности меньших проходов, каждый из которых читает и записывает ПП. Монолитная структура может оказаться более эффективной. А многопроходная выливается в менее сложную реализацию и более простую отладку компилятора. Кроме того, она обладает достаточной гибкостью, чтобы задействовать разные подмножества оптимизаций в зависимости от ситуации. Какой подход выбрать, зависит от ограничений на построение и эксплуатацию компилятора.

1.3. Краткий обзор трансляции

Для трансляции кода, написанного на некотором языке программирования, в код, исполняемый целевой машиной, компилятор выполняет много шагов. Чтобы сделать этот абстрактный процесс более конкретным, рассмотрим, какие шаги нужны для генерирования исполняемого кода следующего выражения:

```
a ← a × 2 × b × c × d
```

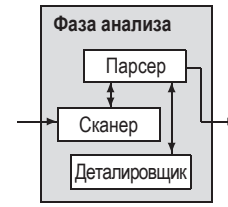
где `a`, `b`, `c` и `d` – переменные, ← означает присваивание, а `×` – оператор умножения. В следующих подразделах мы проследим путь, которым следует компилятор, чтобы преобразовать это простое присваивание в исполняемый код.

1.3.1. Фаза анализа

Прежде чем компилятор сможет транслировать выражение в исполняемый код целевой машины, он должен понять его форму, или *синтаксис*, и смысл, или *семантику*. Фаза анализа определяет, является ли выходной код синтаксически и семантически корректным. Если да, то создается промежуточное

представление кода, иначе пользователю выдаются диагностические сообщения, помогающие найти ошибки в коде.

На рис. 1.1 показан классический компилятор в виде последовательности проходов. Оптимизатор и фаза синтеза действительно работают таким образом, но у фазы анализа несколько иная структура управления, она показана во врезке. Сканер преобразует поток литер исходного кода в поток слов. Он распознает правильно написанные слова; например, в большинстве языков программирования `1g2h3i` не является ни допустимым идентификатором, ни допустимым числом. Парсер сопоставляет полученные от сканера слова с основанной на правилах моделью синтаксиса входного языка, называемой *грамматикой*. Он вызывает сканер всякий раз, как ему понадобится очередное слово. В процессе сопоставления слов с правилами грамматики парсер может вызывать деталировщик (elaborator), который производит дополнительные вычисления. Примерами таких вычислений могут служить построение ПП, проверка согласованности типов или выделение и форматирование памяти.



Грамматика
Модель синтаксиса языка программирования

Нотация

Книги по компиляторам – это, по существу, книги о нотации. Ведь компилятор преобразует программу, написанную в одной нотации, в эквивалентную программу, написанную в другой нотации. При чтении этой книги у вас не раз будут возникать вопросы, относящиеся к нотации. В некоторых случаях от них будет напрямую зависеть понимание материала.

Выражение алгоритмов. Мы хотели, чтобы алгоритмы были краткими. Они написаны на сравнительно высоком уровне, т. е. предполагается, что читатель способен восполнить детали реализации. Алгоритмы набраны *наклонным беззасечным шрифтом*. Отступы значимы, как в языке PYTHON. Код с одинаковыми отступами образует блок. В следующем фрагменте кода

```
if Action[s,word] = "shift s_i" then
   push ⟨word, s_i⟩
   word ← NextWord( )
else if ...
```

все предложения между `then` и `else` являются частью ветви `then` конструкции `if-then-else`. В случае когда блок состоит из одного предложения, мы иногда записываем в одной строке с `then` или `else`, если смысл не вызывает сомнений.

Запись кода. В некоторых примерах мы приводим фактический текст программы, написанный на некотором языке, выбранном для демонстрации конкретного положения. Реальный текст программы набран моноширинным шрифтом.

Арифметические операторы. Наконец, мы отказались от традиционного использования знаков * вместо × и / вместо ÷ всюду, кроме реального текста программ. Надеемся, что смысл будет понятен читателю.

Сканеры и парсеры находят широкое применение и в других системах. Инструментальные средства генерируют онлайновый контент на языках разметки, например HTML; браузеры сканируют и разбирают этот контент для отображения. Для передачи данные записываются в каком-то внешнем формате, например YAML или CSV; программы, читающие эти данные, должны просканировать и разобрать их. Электронная таблица знает, что функция, например `average`, принимает список аргументов; она должна сканировать и разбирать формулы, чтобы найти имена функций и убедиться в правильности списков аргументов.

Сканеры и парсеры встроены в мириады других инструментов. Некоторые из них реализованы с помощью генераторов, как описано в главах 2 и 3. Другие написаны вручную. Но так или иначе, все эти сканеры и парсеры не смутят автора компилятора.

Проверка синтаксиса

Чтобы проверить синтаксис программы, компилятор должен сопоставить структуру программы с грамматикой языка. Грамматика – это формальное определение синтаксиса языка. Из нее также вытекает эффективный механизм проверки того, что вход действительно является предложением языка.

Грамматика определяет множество, обычно бесконечное, строк, состоящих из слов. В ней имеется множество правил, называемых *продукциями*, определенных на множестве слов языка, и множество синтаксических переменных, необходимых для привнесения структуры.

В грамматических правилах используются части речи, или синтаксические категории, а не отдельные слова. Таким образом, одно правило определяет много предложений. Например, в английском языке многие предложения имеют форму:

Предложение → *Субъект* `глагол` *Объект* `знак конца`

где `глагол` и `знак конца` – части речи, а *Предложение*, *Субъект* и *Объект* – синтаксические переменные. *Предложение* представляет любую строку вида, описываемого данным правилом. Символ «→» читается как «это» и означает, что экземпляр правой части можно свести к синтаксической переменной в левой части.

Сканер
проход компилятора, который преобразует строку литер в поток классифицированных слов.

Рассмотрим предложение «Compilers are engineered systems». Первый шаг к пониманию синтаксиса этого предложения – выделить отдельные слова и классифицировать их как части речи. В компиляторе этим занимается сканер, который преобразует вход, т. е. поток литер, в поток классифицированных слов, т. е. пар вида (*p*, *s*), где *p* – *часть речи*, приписанная слову, а *s* – написание самого слова. Сканер мог бы преобразовать предложение из нашего примера в следующий поток классифицированных слов:

(noun, “Compilers”), (verb, “are”), (adjective, “engineered”),
(noun, “systems”), (endmark, “.”)

На практике строку, содержащую написание слова, можно было бы хранить в хеш-таблице и представлять парой с целочисленным индексом, чтобы упростить проверки на равенство. В главе 2 обсуждается теория и практика построения сканеров.

Проверка строк на равенство занимает время, пропорциональное длине строки. Проверка небольших целых чисел на равенство обычно имеет сложность $O(1)$.

На следующем шаге компилятор пытается сопоставить поток слов с правилами грамматики. Например, базовое владение английским предполагает знание грамматических правил, перечисленных на рис. 1.2. Легко увидеть следующий *вывод* предложения «Compilers are engineered systems.».

1	*Предложение*	→	*Субъект* `глагол` *Объект* `знак конца`
2	*Субъект*	→	`существительное`
3	*Субъект*	→	*Модификатор* `существительное`
4	*Объект*	→	`существительное`
5	*Объект*	→	*Модификатор* `существительное`
6	*Модификатор*	→	`прилагательное`

Рис. 1.2 Простые правила английского языка

Правило	Прототип предложения
–	*Предложение*
1	*Субъект* `глагол` *Объект* `знак конца`
2	`существительное глагол` *Модификатор* `существительное знак конца`
5	`существительное глагол` *Объект* `знак конца`
6	`существительное глагол прилагательное знак конца`

Вывод начинается с синтаксической переменной *Предложение*. На каждом шаге перезаписывается одна синтаксическая переменная в прототипе предложения – вместо нее подставляется правая часть некоторого правила. На первом шаге используется правило 1 для замены *Предложение*. На втором шаге используется правило 1 для замены *Субъект*. На третьем шаге используется правило 5 для замены *Объект*, а на последнем шаге – правило 6 для замены *Модификатор* на `прилагательное`. В этот момент выведенное предложение совпадает с потоком классифицированных слов, порожденным сканером.

Вывод доказывает, что предложение «Compilers are engineered objects.» принадлежит языку, который описывается простой грамматикой, образованной правилами 1–6. Это предложение грамматически правильно. Процесс автоматического нахождения вывода называется *синтаксическим разбором*. В главе 3 описаны методы, которыми компиляторы пользуются для разбора входной программы.

Парсер
проход компилятора, который определяет, является ли входной поток предложением исходного языка.

Грамматически правильное предложение может быть бессмысленным. Например, предложение «Rocks are green vegetables.» (Камни являются зелеными овощами) состоит из тех же частей речи и в том же порядке, что предложение «Compilers are engineered objects.», но никакого разумного смысла у него нет. Чтобы понять, в чем разница между этими двумя предложениями, необходимы контекстуальные знания о программных системах, камнях и овощах.

Когда парсер нашел вывод, доказав тем самым, что программа синтаксически корректна, он должен создать различные структуры данных, необходимые другим частям компилятора. Этот процесс включает: построение ПП входной программы, дополненного таблицами, которые описывают все встречающиеся в программе имена; использование ПП кода для проверки согласованности типов; назначение памяти каждому значению, которое будет вычислять программа. Некоторые из этих вычислений, например построение ПП входной программы, выполняются постепенно, по мере работы парсера. Другие, например назначение памяти, производятся путем обхода ПП по завершении работы парсера.

Эти дополнительные вычисления мы называем семантической детализацией. Они расширяют «смысл» программы, переходя от простого синтаксиса к операционному определению. В процессе вычислений компилятор может обнаружить ошибки на более глубоких уровнях, чем может описать грамматика, например: несоответствие типов и численные ошибки, такие как ссылка на элемент массива, содержащая неправильное число измерений, или вызов процедуры со слишком большим числом параметров. В главе 5 обсуждаются некоторые проблемы, возникающие в процессе семантической детализации, и механизмы, которые авторы компиляторов используют для их разрешения.

Промежуточные представления

Последняя проблема, которую решает фаза анализа, – генерирование ПП кода. Типичное ПП состоит из представления кода плюс структуры данных, содержащие дополнительную информацию.

$t_0 \leftarrow a \times 2$
$t_1 \leftarrow t_0 \times b$
$t_2 \leftarrow t_1 \times c$
$t_3 \leftarrow t_2 \times d$
$a \leftarrow t_3$

Низкоуровневое последовательное ПП для $a \leftarrow a \times 2 \times b \times c \times d$

Компиляторы пользуются различными видами ПП в зависимости от исходного языка, целевого языка и конкретных целей компилятора. Некоторые ПП представляют программу в виде графа. Другие напоминают последовательный ассемблерный код программы. Во врезке показано, как могло бы выглядеть последовательное низкоуровневое ПП для выражения $a \leftarrow a \times 2 \times b \times c \times d$. В главе 4 приведен обзор различных видов ПП.

Для каждой конструкции исходного языка у компилятора должна быть стратегия ее реализации в виде ПП. От ее выбора зависит способность компилятора преобразовывать и улучшать код. Главы 4 и 5 посвящены пробле-

мам, которые возникают при генерировании ПП для конструкций исходного кода. Вызовы процедур являются одновременно источником неэффективности конечного кода и клеем, скрепляющим разные исходные файлы в единую программу. Предмет главы 6 – реализация вызовов процедур. Затем в главе 7 представлены стратегии реализации для большинства других языковых конструкций.

1.3.2. Оптимизатор

При выводе ПП входной программы фаза анализа обрабатывает предложения по одному в том порядке, в каком они встречаются. Таким образом, начальное ПП содержит общие стратегии реализации, которые будут работать в любом объемлющем контексте, который мог бы сгенерировать компилятор. Но на этапе выполнения код будет исполняться в более ограниченном и предсказуемом контексте. Оптимизатор может проанализировать ПП кода, установить какие-то факты, относящиеся к контексту, и, используя полученные знания, переписать код, так чтобы он вычислял тот же самый ответ более эффективно.

Об ILOC

В этой книге низкоуровневые примеры написаны на промежуточном представлении, которое называется ILOC – акроним «intermediate language for an optimizing compiler» (промежуточный язык для оптимизирующего компилятора). С годами ILOC претерпел много изменений. В приложении А он описывается во всех подробностях.

Вы можете рассматривать ILOC как язык ассемблера для простой RISC-машины. Он располагает стандартным набором операций. Большинство операций читают операнды из регистров и записывают результаты в регистры. Операции с памятью, `load` и `store`, перемещают значения между памятью и регистрами. Чтобы упростить изложение, мы в большинстве примеров предполагаем, что все данные целые.

У любой операции есть множество операндов и множество результатов. Операция состоит из пяти частей: код операции, список операндов, разделитель, список результатов и факультативный комментарий. Чтобы сохранить сумму регистров 1 и 2 в регистре 3, программист мог бы написать

```
add r1,r2 ⇒ r3      // пример команды
```

Разделитель ⇒ предшествует списку результатов. Он визуально напоминает, что информация течет слева направо, и устраняет неоднозначность, которая могла бы возникнуть, если бы читатель ассемблерного кода перепутал операнды и результаты (см., например, операции `loadAI` и `storeAI`).

Чтобы упростить обсуждение в главе 7, ILOC предлагает несколько способов записи условного ветвления. Он поддерживает булево сравнение и условный переход, а также более традиционный переход по коду условия. В обоих случаях ветви для выбранного и невыбранного путей снабжаются метками, чтобы все межблочные переходы были явными. ILOC не поддерживает «проваливание» в ветвлении.

Полное время работы
Под полным временем выполнения понимается время, необходимое для выполнения программы в системе, где больше никакие программы не работают.

Под эффективностью можно понимать разные вещи. Цель классической оптимизации кода – уменьшить полное время выполнения приложения. При других обстоятельствах оптимизатор может попытаться уменьшить размер откомпилированного кода или потребление энергии процессором. Компиляторы настраиваются в соответствии с различными понятиями эффективности. В главе 8 приведено введение в оптимизацию кода и рассмотрены примеры для нескольких участков кода разного размера.

```
b ← …
c ← …
a ← 1
for i = 1 to n
    read d
    a ← a × 2 × b × c × d
    end
```

(a) Оригинальный код в контексте

```
b ← …
c ← …
a ← 1
t ← 2 × b × c
for i = 1 to n
    read d
    a ← a × d × t
    end
```

(b) Улучшенный код

Рис. 1.3 Контекст имеет значение

Инвариант цикла
Значение, которое не изменяется при выполнении итераций цикла, называется *инвариантом цикла*.

Чтобы конкретизировать эти идеи, на рис. 1.3 показан наш пример присваивания, `a ← a × 2 × b × c × d`, погруженный в простой цикл. В операции присваивания участвует пять значений: `a`, `b`, `c`, `d` и `2`. На каждой итерации цикла `a` и `d` изменяются, тогда как `b`, `c` и `2` в цикле не изменяются, они являются *инвариантами цикла*. Если оптимизатор в состоянии установить их инвариантность, то он может переписать код, как показано на рис. 1.3(b). В этой версии кода количество умножений уменьшено с 4n до 2n + 2. Для n > 1 переписанный цикл должен работать быстрее.

Анализ

Анализ потока данных
форма рассуждений на этапе компиляции о потоке значений на этапе выполнения.

Большинство оптимизаций состоят из анализа и преобразования. В процессе анализа определяется, где компилятор может безопасно и с выгодой применить преобразование. Для поддержки преобразований компиляторы применяют несколько видов анализа. Идея *анализа потока данных* – проследить поток значений во время выполнения. Такие анализаторы обычно решают систему уравнений, выведенную из артефактов ПП кода. В процессе *анализа зависимостей* используются теоретико-числовые критерии для рассуждений об относительной независимости ссылок на память. Компиляторы применяют этот вид анализа для устранения неодно-

значности ссылок на элементы массива. В главе 9 мы будем изучать анализ потока данных, а также трансляцию в форму с единственным статическим присваиванием и обратно; это широко используемое ПП, в котором закодирована информация о потоках значений и управления.

Преобразование

Для улучшения кода компилятор должен не ограничиваться его анализом, а использовать результаты анализа, чтобы переписать код в более эффективной форме. Придумано бессчетное количество преобразований, позволяющих улучшить временные или пространственные характеристики исполняемого кода. Одни, например перемещение инвариантного кода за пределы цикла, показанное на рис. 1.3(b), улучшают время работы программы. Другие делают код более компактным. Преобразования разнятся по эффекту, протяженности участка кода, на котором действуют, и анализу, необходимому для их поддержки. Литература по преобразованиям обширна; тема настолько широка и глубока, что заслуживает отдельной книги или даже нескольких. В главе 10 обсуждаются скалярные преобразования, призванные повысить производительность кода, работающего на одном процессоре. Там представлена таксономия для систематизации предмета и приведены соответствующие примеры.

1.3.3. Фаза синтеза

В фазе синтеза компилятор обходит ПП и генерирует код для целевой машины. При этом решаются по меньшей мере три важные задачи.

- Преобразовать операции ПП в эквивалентные операции ISA целевого процессора; этот процесс, называемый *выбором* (селекцией) *команд*, является предметом главы 11.
- Выбрать порядок выполнения операций; этот процесс, называемый *планированием (выполнения) команд*, рассматривается в главе 12.
- Решить, какие значения должны находиться в регистрах в каждой точке кода; этот процесс, называемый *распределением регистров*, – предмет главы 13.

Мы тщательно проводим различие между *операцией*, одной атомарной инструкцией, исполняемой процессором или функциональным блоком, и *командой*, набором, состоящим из одной или нескольких операций, которые начинают выполняться в одном и том же такте процессора.

В большинстве компиляторов эти три процесса обрабатываются по отдельности. Эти раздельные, но тесно связанные между собой процессы часто объединяют термином «генерирование кода», хотя только у селектора команд основной обязанностью является генерирование команд целевой машины.

Каждая из этих трех задач сама по себе является вычислительно трудной проблемой. Не очень понятно, как определить оптимальный выбор команд,

но задача генерирования самой быстрой последовательности команд для процедуры в данной ISA упирается в перебор огромного количества альтернатив. Планирование команд – NP-полная задача даже для простого блока в большинстве реалистичных моделей выполнения, а переход к более длинным участкам кода не делает ее проще. Задача распределения регистров в общем виде также является NP-полной в процедурах с нетривиальным потоком управления.

Выбор команд

$t_0 \leftarrow a \times 2$
$t_1 \leftarrow t_0 \times b$
$t_2 \leftarrow t_1 \times c$
$t_3 \leftarrow t_2 \times d$
$a \leftarrow t_3$

Низкоуровневое ПП для присваивания $a \leftarrow a \times 2 \times b \times c \times d$

На первом этапе кодогенератор преобразует операции ПП в операции целевой машины, этот процесс называется *выбором команд*. При этом каждая операция ПП, с учетом своего контекста, преобразуется в одну или несколько операций целевой машины. Для иллюстрации рассмотрим преобразование нашего примера выражения a ← a × 2 × b × c × d в ILOC-код. Низкоуровневое ПП этого присваивания показано во врезке.

```
loadAI   r_arp, @a ⇒ r_a      // загрузить 'a'
loadI    2         ⇒ r_2      // записать константу 2 в r_2
loadAI   r_arp,@b  ⇒ r_b      // загрузить 'b'
loadAI   r_arp,@c  ⇒ r_c      // загрузить 'c'
loadAI   r_arp,@d  ⇒ r_d      // загрузить 'd'
mult     r_a, r_2  ⇒ r_a      // r_a ← a × 2
mult     r_a, r_b  ⇒ r_a      // r_a ← (a × 2) × b
mult     r_a, r_c  ⇒ r_a      // r_a ← (a × 2 × b) × c
mult     r_a, r_d  ⇒ r_a      // r_a ← (a × 2 × b × c) × d
storeAI  r_a       ⇒ r_arp, @a // записать r_a назад в 'a'
```

Рис. 1.4 Пример на языке ILOC

r_{arp}
регистр, предназначенный для хранения указателя на текущую запись активации.

Область локальных данных начинается с фиксированного смещения от адреса в r_{arp}.

На рис. 1.4 показана последовательность команд ILOC, которую мог бы сгенерировать компилятор. В ней используется всего четыре операции ILOC:

Операция ILOC	Семантика
loadAI r_1,c_2 ⇒ r_3	Память (r_1 + c_2) → r_3
loadI c_1 ⇒ r_2	c_1 → r_2
mult r_1,r_2 ⇒ r_3	r_1 × r_2 → r_3
storeAI r_1 ⇒ r_2,c_3	r_1 → Память (r_2 + c_3)

Здесь предполагается, что a, b, c и d размещены со смещениями @a, @b, @c и @d от адреса, хранящегося в регистре r_{arp}. Код загружает все необходимые значения в регистры, выполняет операции умножения по порядку и сохраняет результат в ячейке памяти, отведенной для a.

Виртуальный регистр
символическое имя регистра, которое компилятор использует, чтобы показать, что значение может быть сохранено в регистре.

Предполагается, что имеется неограниченный запас *виртуальных регистров*. Им присваиваются символические имена, например в r_a хранится a, а в r_{arp} хранится адрес, с которого начинается область памяти, отведенная для наших именованных значений. Этот селектор команд полагается на то, что распределитель регистров отобразит виртуальные регистры в физические регистры целевой машины.

Физический регистр
имя, представляющее реальный регистр ISA целевой машины.

Этот селектор команд может воспользоваться специальными операциями целевой машины. Например, если доступна операция непосредственного умножения, `multI`, то можно заменить `mult` r_a, $r_2 \Rightarrow r_a$ на `multI` r_a, `2` $\Rightarrow r_a$, устранив необходимость в операции `loadI 2` $\Rightarrow r_2$ и уменьшив спрос на регистры на 1. С другой стороны, если сложение быстрее умножения, то можно было бы просто заменить последовательность `loadI` – `mult` операцией `add` r_a, $r_a \Rightarrow r_a$, которая должна быть быстрее. В главе 11 описано два метода выбора команд, в которых сопоставление с образцом используется для выбора эффективных реализаций операций ПП.

Распределение регистров

В процессе выбора команд компилятор сознательно игнорирует тот факт, что множество физических регистров целевой машины ограничено. Вместо физических он использует виртуальные регистры и предполагает, что существует «достаточное» число регистров. На практике на ранних стадиях компиляции спрос на физические регистры может превышать их предложение оборудованием. Распределитель регистров должен отобразить виртуальные регистры в физические. То есть для каждой точки кода распределитель должен решить, какие значения будут находиться в физических регистрах. Затем он переписывает код, чтобы отразить свои решения. Например, распределитель регистров мог бы переписать код нашего примера следующим образом:

```
loadAI  r_arp, @a ⇒ r1       // загрузить 'a'
add     r1, r1    ⇒ r1       // r1 ← a × 2
loadAI  r_arp, @b ⇒ r2       // загрузить 'b'
mult    r1, r2    ⇒ r1       // r1 ← (a × 2) × b
loadAI  r_arp, @c ⇒ r2       // загрузить 'c'
mult    r1, r2    ⇒ r1       // r1 ← (a × 2 × b) × c
loadAI  r_arp, @d ⇒ r2       // загрузить 'd'
mult    r1, r2    ⇒ r1       // r1 ← (a × 2 × b × c) × d
storeAI r1        ⇒ r_arp, @a // записать r1 назад в 'a'
```

В этой последовательности используется три регистра вместо шести.

Минимизация использования регистров может оказаться контрпродуктивной. Если, например, какие-либо из именованных величин `a`, `b`, `c`, `d` уже находятся в регистрах, то код мог бы сослаться на эти регистры непосредственно. Если все они находятся в регистрах, то последовательность можно было бы реализовать так, что никаких дополнительных регистров не потребуется. С другой стороны, если в каком-то выражении поблизости также вычисляется `a × 2`, то, возможно, лучше было бы сохранить это значение в регистре и не вычислять его заново. При такой оптимизации повторное вычисление `a × 2` исключается, но ценой увеличения спроса на регистры.

Живучесть
Говорят, что значение *v* живо в некоторой точке *p*, если существует путь из *p* к использованию *v*, вдоль которого *v* не переопределяется.

Для реализации такого компромисса распределитель регистров анализирует код с целью определить, где *живо* каждое значение, обычно представленное виртуальным регистром. Цель распределителя – удерживать каждое значение в регистре на участке кода, где это значение живо. Если это невозможно, то он должен выбрать множество хранящихся в регистрах значений так, чтобы минимизировать затраты на этапе выполнения, вызванные переносом некоторых из них в память. В главе 13 обсуждаются проблемы, возникающие при распределении регистров, и методы, применяемые авторами компиляторов для их разрешения.

Планирование выполнения команд

Чтобы порожденный код выполнялся быстро, кодогенератору, возможно, придется переупорядочить операции, отразив конкретные ограничения производительности целевой машины. Время выполнения различных операций может сильно отличаться. Операции доступа к памяти могут занимать десятки или сотни тактов, а некоторые арифметические операции, особенно умножение и деление, – несколько тактов. Влияние таких длительных задержек на производительность откомпилированного кода может оказаться весьма значительным.

Полное время выполнения (еще раз)
Для линейного блока кода (без переходов) полное время выполнения равно просто общему количеству тактов, необходимых для завершения блока, помноженному на продолжительность одного такта.

Предположим, к примеру, что операция `loadAI` или `storeAI` занимает три такта, `mult` – два такта, а все остальные операции – один такт. В таблице ниже показано, как показанный ранее фрагмент кода выполняется при таких предположениях. В столбце «Начало» показан такт, в котором начинается выполнение каждой операции, а в столбце «Конец» – такт, в котором она завершается.

Начало	Конец	Код				
1	3	loadAI	r_{arp}, @a	⇒ r_1	// загрузить 'a'	
4	4	add	r_1, r_1	⇒ r_1	// $r_1 \leftarrow a \times 2$	
5	7	loadAI	r_{arp}, @b	⇒ r_2	// загрузить 'b'	
8	9	mult	r_1, r_2	⇒ r_1	// $r_1 \leftarrow (a \times 2) \times b$	
10	12	loadAI	r_{arp}, @c	⇒ r_2	// загрузить 'c'	
13	14	mult	r_1, r_2	⇒ r_1	// $r_1 \leftarrow (a \times 2 \times b) \times c$	
15	17	loadAI	r_{arp}, @d	⇒ r_2	// загрузить 'd'	
18	19	mult	r_1, r_2	⇒ r_1	// $r_1 \leftarrow (a \times 2 \times b \times c) \times d$	
20	22	storeAI	r_1	⇒ r_{arp}, @a	// записать r_1 назад в 'a'	

Эта последовательность девяти операций занимает 22 такта. Минимизация использования регистров не приведет к более быстрому выполнению.

Многие процессоры умеют инициировать новые операции, пока длительная операция выполняется. При условии что на результат длительной операции никто не ссылается, пока она не завершится, при таком выполнении не возникнет никаких проблем. Но если какая-то из параллельно выполняемых операций пытается раньше времени прочитать результат длительной операции, то процессор задерживает такую операцию до момента окончания длительной операции. Операция не может начаться, пока все ее операнды не будут готовы, а ее результаты не готовы, пока операция не завершится.

Планировщик команд переупорядочивает операции в коде. Он пытается минимизировать число тактов, расходуемых на ожидание операндов. Разумеется, планировщик должен гарантировать, что новая последовательность порождает тот же результат, что исходная. Во многих случаях планировщик может значительно повысить производительность «наивного» кода. Для нашего примера планировщик мог бы предложить такую последовательность:

Начало	Конец	Код				
1	3	loadAI	r_{arp}, @a	⇒ r_1	// загрузить 'a'	
4	4	loadAI	r_{arp}, @b	⇒ r_2	// загрузить 'b'	
5	7	loadAI	r_{arp}, @c	⇒ r_3	// загрузить 'c'	
8	9	add	r_1, r_1	⇒ r_1	// $r_1 \leftarrow a \times 2$	
10	12	mult	r_1, r_2	⇒ r_1	// $r_1 \leftarrow (a \times 2) \times b$	
13	14	loadAI	r_{arp}, @d	⇒ r_2	// загрузить 'd'	
15	17	mult	r_1, r_3	⇒ r_1	// $r_1 \leftarrow (a \times 2 \times b) \times c$	
18	19	mult	r_1, r_2	⇒ r_1	// $r_1 \leftarrow (a \times 2 \times b \times c) \times d$	
20	22	storeAI	r_1	⇒ r_{arp}, @a	// записать r_1 назад в 'a'	

Эта версия кода занимает всего 13 тактов. В коде используется на один регистр больше, чем раньше. Операции начинаются в каждом такте, кроме 8, 10 и 12. Возможны и другие эквивалентные планы, как и планы равной длины, в которых используется больше регистров. В главе 12 описывается несколько широко применяемых методов планирования.

Взаимодействия между компонентами генерирования кода

Многие по-настоящему трудные проблемы компиляции возникают именно на этапе генерирования кода. Мало того, эти проблемы еще и взаимосвязаны. Например, планировщик команд отодвигает операции загрузки `load` от операций, в которых загруженные значения используются. Это может увеличить период, в течение которого значения нужны, а значит, и растет спрос на регистры на этом участке кода. Аналогично запись значений в конкретные регистры может наложить ограничение на планирование команд, создав «ложную» зависимость между двумя операциями. (Вторую операцию нельзя запланировать, пока первая не завершится, пусть даже значения в общем регистре независимы.) Переименование значений зачастую может устранить такую ложную зависимость, но, возможно, ценой использования большего числа регистров.

1.4. Техническая разработка

Конструирование компилятора – это пример технической разработки, т. е. проектирования и реализации при наличии ограничений. В типичном компиляторе имеется несколько проходов, которые совместно транслируют код с исходного языка на целевой. Такая структура создает четкое разделение обязанностей. Парсер не обращает внимания на то, как используются регистры, а распределитель регистров не задумывается о том, предшествует ли константе «13» начальный нуль. Многопроходная структура современных компиляторов упрощает реализацию, отладку и сопровождение.

В компиляторах применяются десятки алгоритмов и структур данных. Автор компилятора должен на каждом шаге трансляции выбирать один или несколько эффективных алгоритмов. Этот процесс подразумевает поиск компромиссов между временем выполнения, сложностью реализации и эффективностью. Для любой конкретной задачи автор компилятора может найти несколько разных решений; у каждого из них будут свои сильные и слабые стороны, затраты и выгоды. Автор компилятора комбинирует набор решений, добиваясь желаемого результата.

Успешный компилятор выполняется бессчетное число раз. Только представьте себе, сколько раз запускается компилятор GCC. За время существования GCC даже мелкие неэффективности накапливаются и занимают ощутимое время. Экономия, достигнутая за счет хорошего дизайна и реализации, со временем дает о себе знать. Поэтому авторы компиляторов должны обращать внимание на временные затраты, в т. ч. на асимптотическую сложность алгоритмов, фактическое время работы реализации и память, занятую структурами данных. Автор компилятора должен думать о том, сколько времени компилятор будет тратить на различные задачи.

Фаза анализа компилятора должна быть быстрой. Основанные на теории методы, представленные в главах 2 и 3, порождают сканеры и парсеры, работающие за время $O(n)$, где n – мера размера и структуры входной программы.

С другой стороны, на этапах оптимизации и генерирования кода решаются задачи, требующие больше времени, и сложность применяемых алгоритмов может быть больше $O(n)$. Таким образом, выбор алгоритма для оптимизатора и фазы синтеза может оказать на время компиляции влияние, большее, чем фаза анализа. Автор компилятора должен соизмерять точность анализа и эффективность оптимизации с ростом времени компиляции. Решения, принимаемые на стадии проектирования и реализации, должны быть хорошо продуманными и взвешенными.

Конструирование JIT-компилятора – вершина технической разработки компиляторов. Поскольку система должна компенсировать время, затраченное на компиляцию, улучшенным временем выполнения, автор JIT-компилятора сталкивается с более строгими ограничениями, чем автор традиционного компилятора. В главе 14 обсуждается, какие компромиссы возникают при проектировании и реализации оптимизаторов, работающих на этапе выполнения.

1.5. Итоги и перспективы

Конструирование компилятора – сложная задача. В хорошем компиляторе соединяются идеи из таких областей, как теория формальных языков, изучение алгоритмов, искусственный интеллект, проектирование систем, компьютерная архитектура и теория языков программирования. Эти идеи применяются к проблеме трансляции программ. Компилятор вбирает в себя жадные алгоритмы, эвристические методы, алгоритмы на графах, динамическое программирование, автоматы, алгоритмы с неподвижной точкой, синхронизацию и локальность, распределение и именование. Многие проблемы, с которыми сталкивается компилятор, слишком трудны или требуют непомерных вычислительных ресурсов для нахождения оптимального решения; поэтому применяются различные аппроксимации и эвристики. В итоге компиляторы содержат сложные взаимодействия, которые могут приводить к непредвиденным результатам – не только хорошим, но и плохим.

Чтобы ввести всю эту деятельность в упорядоченные рамки и обеспечить четкое разделение обязанностей, большинство компиляторов организованы в виде трех больших фаз: анализа, оптимизации и синтеза. В каждой фазе решается свой набор задач, а применяемые для решения подходы сильно различаются. Цель фазы анализа – трансляция исходного кода в некоторое промежуточное представление (ПП). Эта фаза опирается на результаты из теории формальных языков и теории типов, с изрядной дозой алгоритмов и структур данных. Средняя фаза, или оптимизатор, транслирует одно ПП в другое с целью создать ПП-программу, которая будет исполняться эффективно. Оптимизаторы анализируют программу, чтобы узнать о ее поведении на этапе выполнения, а затем используют эти знания для преобразования кода и улучшения его поведения. Фаза синтеза отображает ПП программы на систему команд конкретного процессора. В этой фазе ищутся приближен-

ные решения трудных задач распределения регистров и планирования команд, и качество аппроксимации напрямую влияет на скорость и размер откомпилированного кода.

> Материал глав 5–7 переплетен. Мы рекомендуем вам прочитать все три главы, а затем перечитать их еще раз.

В этой книге рассматриваются все фазы. В главах 2 и 3 описываются алгоритмы, применяемые в фазе анализа. Глава 4 содержит введение в ПП. В главах 5–7 обсуждается реализация различных абстракций языка программирования, в т. ч. процедур и управляющих конструкций, а также механизмы, применяемые для создания их образов в ПП. Глава 8 – введение в оптимизацию кода. В главах 9 и 10 для интересующихся читателей приведено более подробное рассмотрение анализа и оптимизации. Главы 11–13 посвящены методам выбора команд, планирования выполнения команд и распределения регистров. Наконец, в главе 14 описано, как все эти методы сочетаются в современных оптимизаторах этапа выполнения и в JIT-компиляторах.

Замечания к этой главе

Первые компиляторы появились в 1950-х годах. Эти ранние системы поражали своей технической изощренностью. Оригинальный компилятор языка FORTRAN представлял собой многопроходную систему, включавшую сканер, парсер и распределитель регистров, а также некоторые оптимизации [27, 28]. Система Alpha, созданная Ершовым и его сотрудниками, выполняла локальную оптимизацию [150] и применяла алгоритм раскраски графов, чтобы уменьшить объем памяти, необходимой для элементов данных [151, 152]. В ранних трудах по ALGOL-60 было разработано много принципиальных идей, которые до сих пор играют важную роль в реализации компиляторов; примеры см. в выпуске журнала «Communications of the ACM» (CACM) за январь 1961 года [160, 213, 214, 316].

Кнут опубликовал интересные воспоминания о конструировании компиляторов в начале 1960-х годов [238]. Ранделл и Рассел описывают, как создавалась первая реализация ALGOL-60 [303]. Аллен живописует историю разработки компилятора в компании IBM с упором на переплетение теории и практики [15].

В 1960-е и 1970-е годы было построено много влиятельных компиляторов. К ним относятся классический оптимизирующий компилятор FORTRAN H [260, 317], компиляторы Bliss-11 и Bliss-32 [79, 368] и переносимый компилятор BCPL [308]. Они порождали высококачественный код для различных машин с архитектурой CISC. С другой стороны, компиляторы для студентов ставили во главу угла быстроту компиляции, хорошие диагностические сообщения и механизмы исправления ошибок [104, 156].

Появление архитектуры RISC в 1980-х годах привело к созданию нового поколения компиляторов, которые были сфокусированы на качестве опти-

мизации и генерирования кода [25, 88, 96, 215]. В этих компиляторах присутствовали полноценные многопроходные оптимизаторы. Современные компиляторы для RISC-машин по-прежнему следуют этой модели.

На протяжении 1990-х годов основным содержанием исследований по конструированию компиляторов было реагирование на быстрые изменения, происходившие в архитектуре микропроцессоров и систем. Это десятилетие началось появлением процессора Intel i860, который поставил перед авторами компиляторов задачу управления конвейером и задержками памяти. А закончилось оно такими разнородными проблемами, как наличие нескольких функциональных блоков, долгие задержки памяти и генерирование параллельного кода. Структура и организация RISC-компиляторов 1980-х годов оказались достаточно гибкими, чтобы справиться с этими новыми вызовами. Исследователи просто добавили новые проходы в свои оптимизаторы и кодогенераторы.

Хотя в системах JAVA используется комбинация компиляции и интерпретации [69, 288], JAVA – не первый язык, в котором встречается такая смесь. Системы LISP уже давно включали как компиляторы в платформенный код, так и схемы реализации на основе виртуальной машины [275, 336]. В системе SMALLTALK-80 применялись распространение байт-кода и виртуальная машина [242]; в несколько реализаций были добавлены также JIT-компиляторы [137].

После 2000 года снижение стоимости компиляции позволило авторам компиляторов отвести больше ресурсов для анализа и преобразований. Это привело к внедрению более мощных методов анализа проблем, включая анализ целых программ и библиотек [372], а также к широкому распространению анализа указателей с целью исключить неоднозначность ссылок на память [203]. Рост задержек памяти придал большую важность оптимизациям, направленным на улучшение локальности и маскировку задержек [2, 52, 350].

Упражнения

1. Автор компилятора не раз сталкивается с необходимостью идти на компромиссы. Назовите пять качеств компилятора, которые вы как пользователь считаете самыми важными при покупке. Изменится ли этот список, если вы поставите себя на место автора компилятора? Что этот список говорит вам о компиляторе, который вы стали бы реализовывать?
2. Рассмотрим простой веб-браузер, который принимает текстовую строку в формате HTML и отображает описанную в ней графику на экране. Чем является процесс отображения – компиляцией или интерпретацией?
3. Компиляторы используются в разных ситуациях. Какие различия вы ожидали бы увидеть между компиляторами, предназначенными для следующих приложений?
   a. JIT-компилятор для трансляции кода пользовательского интерфейса, загруженного по сети.

b. Компилятор, ориентированный на встраиваемый процессор в мобильном устройстве.
c. Компилятор, используемый на уроках по введению в программирование в школе.
d. Компилятор для программы аэродинамического моделирования, которая работает в большом кластере одинаковых процессоров.
e. Компилятор, ориентированный на создание расчетных программ, работающих на большом числе разнородных машин.

Глава 2

Сканеры

АННОТАЦИЯ

Задача сканера – преобразовать поток литер в поток слов входного языка. Каждому слову должна быть сопоставлена его синтаксическая категория, «часть речи». Сканер – единственный проход компилятора, на котором читаются все символы входной программы. Авторы компиляторов ставят во главу угла скорость сканирования, отчасти потому что вход сканера в некотором смысле больше входа любого другого прохода, а отчасти потому что весьма эффективные методы сканирования легко понять и реализовать.

Эта глава содержит введение в регулярные выражения – специальную нотацию, используемую для описания множества допустимых слов языка программирования. Здесь же разрабатываются формальные механизмы генерирования сканеров по регулярным выражениям, вручную или автоматически.

КЛЮЧЕВЫЕ СЛОВА

Сканер, конечный автомат, регулярное выражение, алгоритм вычисления неподвижной точки

2.1. Введение

Сканирование – первая из трех частей, используемых в фазе анализа, чтобы понять входную программу. Сканер, или лексический анализатор, читает поток литер и порождает поток слов. Парсер, или синтаксический анализатор, сопоставляет этот поток слов с грамматической моделью входного языка. Он вызывает функции семантической детализации для выполнения углубленного анализа и построения структур, моделирующих истинный смысл входной программы.

Сканер составляет из литер слова. Для каждого слова он определяет, является ли оно допустимым словом входного языка. Каждому допустимому слову сопоставляется *синтаксическая категория*, или часть речи.

Синтаксическая категория классификация слов в соответствии с их грамматическим употреблением. Практически эти категории соответствуют терминальным символам грамматики языка (см. раздел 3.2.2).

Сканер – единственный проход компилятора, который видит каждую литеру входной программы. Поскольку сканеры решают сравнительно простую задачу – сгруппировать литеры в слова и знаки препинания входного языка, – их реализации работают быстро. Широко употребляются автоматические инструменты генерирования сканеров, которые обрабатывают математическое описание лексической структуры языка и порождают быстрый распознаватель. Тем не менее во многих компиляторах используются написанные вручную сканеры, потому что писать их легко, а результирующая программа получается быстрой и надежной.

Концептуальная дорожная карта

Чтобы оттранслировать программу, компилятор должен понимать как ее лексическую структуру – как правильно писать слова, так и синтаксическую структуру – как грамматически правильно соединять слова в предложения и программы. В этой главе описываются математические инструменты и методы программирования, которые обыкновенно применяются для конструирования сканеров – программ, анализирующих лексическую структуру кода.

Распознаватель
программа, идентифицирующая определенные слова в потоке литер.

Методология конструирования сканеров уходит корнями в теорию формальных языков. Сканеры основаны на *распознавателях*, моделирующих детерминированные конечные автоматы. Автор компилятора описывает лексическую структуру входного языка с помощью множества регулярных выражений (РВ). Посредством серии построений эта спецификация преобразуется в сканер, который читает поток литер и порождает поток слов, каждое из которых снабжено синтаксической категорией, взятой из грамматики языка программирования.

Технология, автоматизирующая конструирование сканера, распространена очень широко – от компиляторов до редакторов фильтров URL-адресов в браузере. Легкодоступные инструменты пользуются хорошо разработанной теорией конечных автоматов для построения эффективных распознавателей по простым спецификациям.

Краткий обзор

Лексема
текст слова, распознанного сканером.

Сканер, входящий в состав компилятора, читает входной поток и порождает на выходе поток слов, аннотированных синтаксическими категориями, – эквивалент *части речи* в естественном языке. При каждом вызове сканер возвращает пару ⟨*лексема*, *категория*⟩, где *лексема* – текст слова, а *категория* – его синтаксическая категория. Иногда эту пару называют *токеном*.

Для поиска и классификации слов сканер применяет набор правил, описывающий лексическую структуру входного языка программирования, который

иногда называют *микросинтаксисом*. Микросинтаксис определяет, как литеры группируются в слова и, наоборот, как разделить подряд идущие слова. (В контексте сканирования знаки препинания и другие символы рассматриваются как отдельные слова.)

Микросинтаксис
лексическая структура языка.

У западных языков, например английского, микросинтаксис простой. Соседние буквы, читаемые слева направо, образуют слово. Слово завершается пробелом, а также любой литерой, отличной от буквы. (Алгоритм построения слов может рассматривать дефис в середине слова как букву.) После того как выделена группа литер, потенциально образующих слово, алгоритм построения слов может определить, допустимо ли оно, и если да, то какой частью речи является, произведя поиск в словаре.

У большинства языков программирования столь же простой микросинтаксис. Литеры объединяются в слова. В большинстве языков пробелы и знаки препинания завершают слово. Например, в языке ALGOL-60 и производных от него *идентификатор* определяется как последовательность букв и цифр, начинающаяся с буквы. Идентификатор заканчивается первой же литерой, отличной от букв и цифр. Таким образом, `fee` и `f1e` – допустимые идентификаторы, а `12fum` – нет. Заметим, что множество допустимых слов описывается правилами, а не словарем.

Ключевое слово
слово, которое зарезервировано для определенной синтаксической цели и потому не может быть использовано в качестве идентификатора.

В типичном языке программирования некоторые слова, называемые *ключевыми* или *зарезервированными*, по правилу являются идентификаторами, но имеют специальный смысл. Так, `while` и `static` – ключевые слова в языках C и JAVA. Ключевые слова (и знаки препинания) образуют отдельные синтаксические категории. И хотя слово `static`, согласно правилу, является идентификатором, сканер компилятора C или JAVA, без сомнения, отнесет его к категории, содержащей только один элемент – ключевое слово `static`. Методы, применяемые для реализации сканеров, должны уметь различать отдельные слова, например ключевые, не жертвуя эффективностью распознавания объемлющих их более крупных классов.

В силу простоты лексической структуры языков программирования сканеры получаются эффективными. Автор компилятора начинает со спецификации микросинтаксиса языка. Он либо кодирует микросинтаксис в нотации, понятной генератору сканеров, который затем строит исполняемый файл, либо использует эту спецификацию для построения сканера вручную. Сложность как сгенерированного, так и написанного вручную сканера составляет **O**(1) на литеру, т. е. их время работы пропорционально количеству литер во входном потоке.

В разделе 2.2 мы познакомимся с моделью *распознавателей*, т. е. программ, которые выделяют слова из потока литер. В разделе 2.3 описана формальная нотация, *регулярные выражения*, для описания синтаксиса и распознавателей. В разделе 2.4 показано, как автоматизировать конструирование исполняемо-

го распознавателя по РВ. И наконец, в разделе 2.5 обсуждаются практические вопросы реализации сканеров – как автоматически сгенерированных, так и написанных вручную.

Несколько слов о времени

Предметом главы 2 является проектирование и реализация сканера компилятора. Следовательно, в ней рассматриваются вопросы, возникающие в три разных промежутка времени: на этапе проектирования, на этапе сборки и на этапе компиляции.

На *этапе проектирования* автор компилятора создает спецификации микросинтаксиса для языка программирования (написание слов). Он выбирает метод реализации и пишет необходимый код.

На *этапе сборки* автор компилятора запускает инструменты, собирающие исполняемый сканер. Для сканера, описываемого спецификацией, этот процесс включает использование инструментов, которые преобразуют спецификацию в код, а также компилятора, который преобразует этот код в исполняемый образ.

На *этапе компиляции* конечный пользователь вызывает компилятор, чтобы оттранслировать код приложения в исполняемый код. В состав компилятора входит сканер, и компилятор использует его, чтобы преобразовать код приложения из потока литер в поток слов и отнести каждое слово к синтаксической категории (части речи).

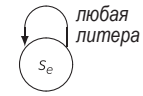
2.2. Распознавание слов

Мы предполагаем, что, достигнув конца файла, *NextChar* возвращает `eof`.

Алгоритм распознавания слов из прочитанных символов прост и понятен. Структура кода может пролить свет на решаемую задачу. Рассмотрим задачу распознавания ключевого слова `new` (см. рис. 2.1(a)). Предполагается, что в коде имеется функция *NextChar*, возвращающая литеры одну за другой. Код сравнивает прочитанную литеру с `n`, затем с `e`, потом с `w`. На каждом шаге несовпадение заставляет код отвергнуть строки и *попробовать что-то другое*. Если единственной целью программы было распознавание «`new`», то она должна сообщить об ошибке. Но поскольку сканеры редко распознают одно-единственное слово, мы сознательно оставили обработку ошибок неопределенной.

В этом фрагменте кода литеры проверяются по одной. Мы можем представить его в виде простой диаграммы переходов, показанной на рис. 2.1(b). Эта диаграмма представляет распознаватель. Каждый помеченный кружок представляет абстрактное состояние вычисления. Начальным состоянием является s_0. Состояние s_3 допускающее; распознаватель достигает s_3, только когда на вход была подана строка «`new`». Двойной кружок обозначает, что s_3 – допускающее состояние.

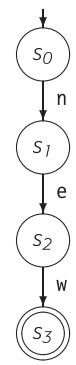
Состояние ошибки
s_e – недопускающее состояние, при получении любой литеры производится переход в него же.



Стрелки представляют переходы из одного состояния в другое при чтении входных литер. Если распознаватель начинает работу в состоянии s_0 и читает литеры `n`, `e` и `w`, то по переходам мы попадаем в состояние s_3. Что случится, если будет подан любой другой вход, например `n`, `o`, `t`? Буква `n` переводит распознаватель в состояние s_1. Буква `o` не помечает никакое ребро, исходящее из s_1. В коде ветви, не совпадающие с `new`, исполняют *попробовать что-то другое*. Распознаватель переходит в состояние ошибки. Рисуя диаграмму переходов распознавателя, мы обычно опускаем переходы в состояние ошибки. Но из каждого состояния ведет неявный переход в состояние ошибки, когда прочитан неспецифицированный вход.

```
c ← NextChar();
if (c = 'n') then
    c ← NextChar();
    if (c = 'e') then
        c ← NextChar();
        if (c = 'w') then
            сообщить об успехе;
        else
            попробовать что-то другое;
    else
        попробовать что-то другое;
else
    попробовать что-то другое;
```

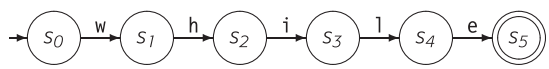
(a) Код



(b) Распознаватель

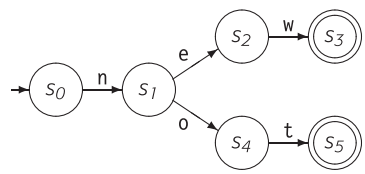
Рис. 2.1 Фрагмент кода для распознавания слова «new»

Применение того же подхода для построения распознавателя `while` дало бы такую диаграмму переходов:



Если распознаватель прошел от s_0 до s_5, то он нашел слово `while`. В соответствующем коде было бы пять вложенных конструкций `if-then-else`.

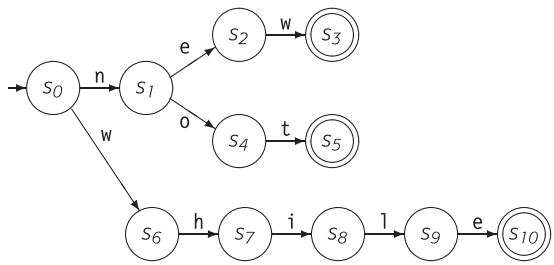
Для распознавания нескольких слов распознаватель может иметь несколько ребер, исходящих из заданного состояния. (В коде им соответствовали бы ветви `else if`.) Прямолинейный распознаватель слов `new` и `not` имеет вид:



Разумеется, читатель мог бы перенумеровать состояния распознавателя, не меняя их «смысла». Перенумерация состояний порождает эквивалентный распознаватель.

В этом распознавателе имеется общая проверка на `n`, которая переводит его из s_0 в s_1 и обозначается $s_0 \xrightarrow{n} s_1$. Если следующей будет прочитана литера `e`, то будет выполнен переход $s_1 \xrightarrow{e} s_2$. Если же прочитана литера `o`, то будет выполнен переход $s_1 \xrightarrow{o} s_4$. Наконец, чтение `w` в состоянии s_2 приводит к переходу $s_2 \xrightarrow{w} s_3$, а чтение `t` в состоянии s_4 – к переходу $s_4 \xrightarrow{t} s_5$. Состояние s_3 означает, что на входе была строка `new`, а состояние s_5 – что строка `not`. Распознаватель совершает один переход при чтении одной входной литеры.

Мы можем объединить распознаватель `new` и `not` с распознавателем `while`, отождествив их начальные состояния и изменив метки всех состояний.



Из состояния s_0 исходят переходы для `n` и `w`. Распознаватель имеет три допускающих состояния, s_3, s_5 и s_{10}. Если в каком-то состоянии встречается входная литера, не совпадающая ни с одним переходом, то распознаватель переходит в неявное состояние ошибки s_e.

Распознаватель выполняет один переход для каждой входной литеры. В предположении, что распознаватель реализован эффективно, следует ожидать, что время его работы будет пропорционально длине входной строки.

2.2.1. Формализм для описания распознавателей

Конечный автомат формализм для описания распознавателей, включающий конечное множество состояний, алфавит, функцию переходов, начальное состояние и одно или несколько допускающих состояний.

Диаграммы переходов играют роль абстракций кода, который понадобился бы для их реализации. Их можно также рассматривать как формальные математические объекты, называемые *конечными автоматами*, которые описывают распознаватели. Формально конечный автомат (КА) определяется как пятерка $(S, \Sigma, \delta, s_0, S_A)$, где

- S – конечное множество состояний распознавателя, включая s_e;
- Σ – конечный алфавит, используемый распознавателем. Как правило, Σ является объединением меток ребер на диаграмме состояний;
- $\delta(s, c)$ – функция переходов распознавателя. Она отображает каждую пару, состоящую из состояния $s \in S$ и литеры $c \in \Sigma$, в какое-то следую-

щее состояние. КА, находящийся в состоянии s_i, после чтения литеры c производит переход $s_i \xrightarrow{c} \delta(s_i, c)$;

- $s_0 \in S$ – выделенное начальное состояние;
- S_A – множество допускающих состояний, $S_A \subseteq S$. Каждое состояние, принадлежащее S_A, на диаграмме переходов обозначено двойным кружком.

В рамках этого формализма КА, распознающий `new`, `not` и `while`, записывается в виде:

$$S = \{s_0, s_1, s_2, s_3, s_4, s_5, s_6, s_7, s_8, s_9, s_{10}, s_e\}$$

$$\Sigma = \{\texttt{e}, \texttt{h}, \texttt{i}, \texttt{l}, \texttt{n}, \texttt{o}, \texttt{t}, \texttt{w}\}$$

$$\delta = \left\{ \begin{matrix} s_0 \xrightarrow{\texttt{n}} s_1, & s_0 \xrightarrow{\texttt{w}} s_6, & s_1 \xrightarrow{\texttt{e}} s_2, & s_1 \xrightarrow{\texttt{o}} s_4, & s_2 \xrightarrow{\texttt{w}} s_3, \\ s_4 \xrightarrow{\texttt{t}} s_5, & s_6 \xrightarrow{\texttt{h}} s_7, & s_7 \xrightarrow{\texttt{i}} s_8, & s_8 \xrightarrow{\texttt{l}} s_9, & s_9 \xrightarrow{\texttt{e}} s_{10} \end{matrix} \right\}$$

$$s_0 = s_0$$

$$S_A = \{s_3, s_5, s_{10}\}$$

Функция δ определена лишь частично. Для всех прочих комбинаций состояния s_i и входной литеры c мы полагаем $\delta(s_i, c) = s_e$, где s_e – состояние ошибки. Эта пятерка эквивалентна диаграмме переходов; зная одно, мы легко можем восстановить другое.

КА, начинающий работу в состоянии s_0, допускает строку x тогда и только тогда, когда последовательность переходов в результате чтения всех литер, составляющих x, оставляет КА в допускающем состоянии. Это соответствует нашему интуитивному пониманию диаграммы переходов. Для строки `new` наш распознаватель совершает переходы $s_0 \xrightarrow{\texttt{n}} s_1$, $s_1 \xrightarrow{\texttt{e}} s_2$ и $s_2 \xrightarrow{\texttt{w}} s_3$. Так как $s_3 \in S_A$ и больше никаких входных литер не осталось, то КА допускает `new`. Для входной строки `nut` поведение будет иным. Прочитав букву `n`, КА совершает переход $s_0 \xrightarrow{\texttt{n}} s_1$. Прочитав `u`, он совершает переход $s_1 \xrightarrow{\texttt{u}} s_e$. Попав в состояние s_e, КА остается в нем, пока не исчерпает входной поток.

Более формально: если строка x состоит из литер $x_1\, x_2\, x_3 \ldots x_n$, то КА $(S, \Sigma, \delta, s_0, S_A)$ допускает x тогда и только тогда, когда

$$\delta(\delta(\ldots\delta(\delta(\delta(s_0, x_1), x_2), x_3) \ldots, x_{n-1}), x_n) \in S_A.$$

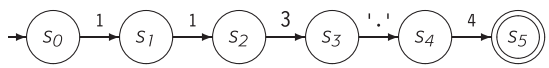
Интуитивно понятно, что это определение соответствует повторному применению δ к паре, состоящей из некоторого состояния s и входной литеры x_i. Базовый случай, $\delta(s_0, x_1)$, представляет переход КА из начального состояния s_0 по литере x_1. Затем состояние $\delta(s_0, x_1)$ подается на вход δ вместе с литерой x_2, что дает новое состояние, и так далее, пока не будет потреблена вся входная строка. Результатом последнего применения δ снова является состояние. Если это состояние допускающее, то КА допускает строку $x_1\, x_2\, x_3 \ldots x_n$.

КА может встретить во входной строке лексическую ошибку. Какая-то литера x_j могла бы перевести его в состояние ошибки s_e. Попадание в s_e имеет место, потому что $x_1\, x_2\, x_3 \ldots x_j$ не является префиксом никакого слова языка, допускаемого КА. Или же КА может достичь конца ввода, находясь в недопускающем состоянии. В любом случае это означает, что входная строка не является словом языка.

Рассмотрим случай, при котором КА останавливается в недопускающем состоянии. Если КА по пути проходил через допускающее состояние, значит, строка содержит префикс, являющийся допустимым словом. Как мы увидим в разделе 2.4.5, сканеры используют это наблюдение для нахождения границ слов.

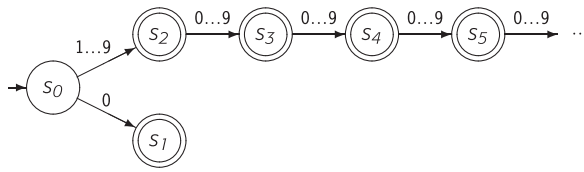
2.2.2. Распознавание более сложных слов

Модель политерного распознавания, продемонстрированная в оригинальном распознавателе `not`, легко обобщается на обработку произвольных наборов полностью специфицированных слов. Но как можно было бы распознать число с помощью такого распознавателя? Конкретное число, например 113.4, не вызывает трудностей.

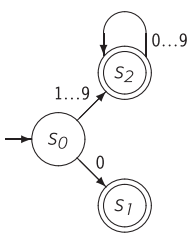


> Мы будем обозначать диапазон литер первым и последним элементами, разделенными многоточием, например `0...9`.
>
> Некоторые авторы используют дефис, но мы предпочитаем многоточие, чтобы не путать со знаком минус.

Но нам-то нужна диаграмма переходов (и соответствующий код), способная распознать любое число. Для простоты ограничимся целыми числами без знака. В общем случае целое либо равно нулю, либо состоит из одной или нескольких цифр, причем первая цифра принимает значения от 1 до 9, а все остальные – от 0 до 9 (это определение исключает начальные нули). Как нарисовать диаграмму переходов при таком определении?



Переход $s_0 \xrightarrow{0} s_1$ обрабатывает случай цифры 0. Другой путь, $\langle s_0, s_2, s_3, \ldots \rangle$, обрабатывает случай цифры, большей нуля. Однако этот путь нарушает условие конечности S.



Заметим, что все состояния на пути, начинающемся в s_2, эквивалентны, т. е. метки на выходных переходах одинаковы и все состояния допускающие. Если разрешить в диаграмме переходов циклы, то мы сможем заменить путь, начинающийся в s_2, одним переходом из s_2 в себя же, как показано во врезке.

Такая циклическая диаграмма переходов не нарушает определение КА. Но с точки зрения реализации, она сложнее,

чем ациклические диаграммы переходов, показанные выше. Мы не можем транслировать ее непосредственно в множество вложенных конструкций `if-then-else`. Появление цикла в графе переходов влечет за собой необходимость циклического потока управления. Это можно реализовать с помощью цикла *while*, как показано в коде на рис. 2.2(a). Эффективно определить δ можно с помощью таблицы:

δ	0	1	2	3	4	5	6	7	8	9	Прочие
s_0	s_1	s_2	s_2	s_2	s_2	s_2	s_2	s_2	s_2	s_2	s_e
s_1	s_e	s_e	s_e	s_e	s_e	s_e	s_e	s_e	s_e	s_e	s_e
s_2	s_2	s_2	s_2	s_2	s_2	s_2	s_2	s_2	s_2	s_2	s_e
s_e	s_e	s_e	s_e	s_e	s_e	s_e	s_e	s_e	s_e	s_e	s_e

```
state ← s0;
char ← NextChar( );
while (state ≠ se and char ≠ eof) do
   state ← δ(state,char);
   char ← NextChar( );
end;
if (state ∈ SA) then
  сообщить о допуске;
else сообщить об ошибке;
```

(a) Код для интерпретации таблицы состояний

$$S = \{s_0, s_1, s_2, s_e\}$$

$$\Sigma = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$$

$$\delta = \left\{ s_0 \xrightarrow{0} s_1,\ s_0 \xrightarrow{1\ldots9} s_2,\ s_2 \xrightarrow{0\ldots9} s_2 \right\}$$

$$S_A = \{s_1, s_2\}$$

(b) Формальное определение КА

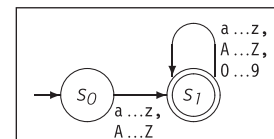
Рис. 2.2 Распознаватель целых без знака

Аналогичный код позволяет реализовать и другие распознаватели – нужно только изменить таблицу. Заметим, что эту таблицу можно сильно сжать. Столбцы для цифр от *1* до *9* одинаковы, поэтому их можно было бы включить только один раз, что сократило бы таблицу до трех столбцов: *0*, [*1*...*9*] и *прочие*. В показанном фрагменте кода сообщение об ошибке выдается сразу при входе в s_e, поэтому строка таблицы для s_e никогда не используется. Если мы исключим строку для s_e, то в таблице останется три строки и три столбца.

Аналогичные КА можно разработать для целых чисел со знаком, вещественных чисел и комплексных чисел. Упрощенная версия правила формирования имен идентификаторов в Algol-подобных языках, в частности C и JAVA, могла бы звучать так:

идентификатор состоит из буквы,
за которой следует 0 или более букв или цифр.

Это определение допускает бесконечное множество идентификаторов. Его можно специфицировать КА с двумя состояниями, показанным во врезке. Многие языки включают специальные литеры, например _ и &, в состав букв.



КА для идентификаторов

Конечный автомат можно рассматривать как спецификацию распознавателя. Однако такие специфика-

ции слишком пространны. Чтобы упростить построение сканера, нам нужна краткая нотация для спецификации лексической структуры слов и способ преобразовать такую спецификацию в КА и в реализующий его код. Последующие разделы данной главы как раз и посвящены развитию этих идей.

КА для целых чисел без знака отчетливо проявляет различие между синтаксической категорией, например «целые без знака», и конкретным словом, например 113. Категория – это множество, состоящее из одного или нескольких слов, например 12 и 113 могли бы быть членами категории «целые без знака».

ИТОГИ РАЗДЕЛА

Распознавание и классификация слов – важнейшая часть понимания синтаксиса программы. Для заданного слова или набора слов автор компилятора может создать политерный распознаватель. Такие распознаватели соответствуют диаграммам переходов, которые, в свою очередь, соответствуют конечным автоматам.

Любое конечное множество слов можно закодировать ациклической диаграммой переходов. Можно также закодировать некоторые бесконечные множества слов, такие как множество целых чисел или множество идентификаторов языка JAVA; при этом получаются циклические диаграммы переходов.

ВОПРОСЫ

1. Постройте КА для идентификаторов, состоящих из буквы, за которой следует до пяти букв или цифр.
2. Постройте КА для комментариев языка PASCAL, состоящих из открывающей фигурной скобки {, за которой следует нуль или более литер из множества Σ – } и в конце закрывающая фигурная скобка }.

2.3. Регулярные выражения

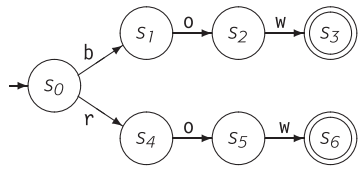
Множество слов, допускаемых конечным автоматом *F*, образует язык, обозначаемый *L*(*F*). Диаграмма переходов *F* точно специфицирует, как записывается каждое слово этого языка. Но диаграммы переходов могут быть сложными и интуитивно непонятными. Поэтому в большинстве систем для этой цели используется нотация *регулярных выражений* (РВ). Язык, описываемый РВ, называется *регулярным языком*.

РВ эквивалентны КА, описанным в предыдущем разделе. (Мы докажем это путем построения в разделе 2.4.) У простых распознавателей простые РВ-спецификации.

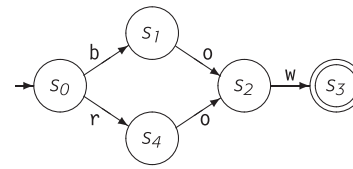
- Язык, состоящий из единственного слова `bow`, можно описать РВ, записываемым как *bow*. Запись двух литер одна за другой означает, что они должны встречаться именно в таком порядке.
- Язык, состоящий из двух слов, `bow` и `row`, можно описать РВ *bow* или *row*. Чтобы избежать потенциально неправильной интерпретации слова *или*,

мы будем вместо него использовать символ |. Таким образом, РВ записывается в виде *bow*|*row*. Символ | называется *альтернацией*.

Для любого заданного языка может существовать несколько РВ, описывающих этот язык. Например, *bow*|*row* и (*b*|*r*)*ow* описывают один и тот же язык. Но разным РВ соответствуют разные КА, как показано ниже. И значит, КА на рисунках (a) и (b) допускают один и тот же язык.



(a) КА, соответствующий РВ *bow*|*row*



(b) КА, соответствующий РВ (*b*|*r*)*ow*

Альтернация коммутативна, поэтому *bow*|*row* описывает тот же самый язык, что *row*|*bow* или (*r*|*b*)*ow*.

Чтобы добавить обсуждению конкретики, рассмотрим несколько примеров, встречающихся в большинстве языков программирования. Знаки препинания, например двоеточие, точка с запятой и различные скобки, представляются своими литерами. Их РВ имеют такое же «написание», как сами знаки препинания. Таким образом, в лексической спецификации языка программирования могут встречаться следующие РВ:

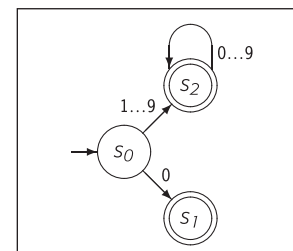
: ; ? => ( ) [ ] + //

Аналогично у ключевых слов простые РВ:

if while this integer instanceof

Для моделирования синтаксических категорий с большим числом слов, например целых чисел или идентификаторов, нам необходим способ обозначения циклического ребра КА.

КА для целого числа без знака, показанный на полях, имеет три состояния: начальное состояние s_0, допускающее состояние s_1 для числа 0 и еще одно допускающее состояние s_2 для всех остальных целых чисел. Ключ к мощности этого КА – переход $s_2 \rightarrow s_2$, имеющий место при чтении каждой дополнительной цифры. Состояние s_2 создает правило для вывода нового целого без знака из существующего: добавить еще одну цифру в конец существующего числа. По-другому это правило можно сформулировать так:



КА для целых чисел без знака

целое число без знака – это либо нуль, либо отличная от нуля цифра, за которой следует нуль или более цифр.

Замыкание Клини
РВ $x*$ означает нуль или более вхождений x.

Чтобы понять суть этого КА, нам понадобится нотация для понятия «нуль или более вхождений» некоторого РВ. Для РВ x запись x^* означает «нуль или более вхождений x». Мы называем оператор * *замыканием Клини*, или просто *замыканием*. С помощью оператора замыкания мы можем написать РВ для этого КА:

0 | (*1*|*2*|*3*|*4*|*5*|*6*|*7*|*8*|*9*) (*0*|*1*|*2*|*3*|*4*|*5*|*6*|*7*|*8*|*9*)*.

Конечно, это РВ можно записать короче: *0*|([*1...9*])([*0...9*])*.

2.3.1. Формализация нотации

Для строгой работы с РВ нам необходимо формальное определение. Пусть имеется алфавит Σ. РВ описывает множество строк над литерами Σ плюс дополнительная литера ϵ, представляющая пустую строку. Множество строк, определяемое РВ, называется *языком*. Язык, определяемый РВ r, обозначается $L(r)$.

РВ строится из трех базовых операций:

1) *альтернация*. Альтернацией, или объединением двух множеств РВ, r и s, обозначаемым $r \mid s$, называется множество $\{x \mid x \in L(r)$ или $x \in L(s)\}$;
2) *конкатенация*. *Конкатенация* двух РВ r и s, обозначаемая rs, содержит все строки, образованные дописыванием строки из $L(r)$ в начало строки из $L(s)$, т. е. $\{xy \mid x \in L(r)$ и $y \in L(s)\}$;
3) *замыкание*. Замыканием Клини r, обозначаемым r^*, называется множество $\bigcup_{i=0}^{\infty} r^i$. $L(r^*)$ содержит все строки, состоящие из нуля или более слов из $L(r)$.

Конечное замыкание
Для любого целого $i \geqslant 0$ РВ r^i обозначает от одного до i вхождений r.

Положительное замыкание
РВ r^+ обозначает одно или более вхождений r.

Для удобства мы иногда используем *конечное замыкание*. Нотация r^i, $i \geqslant 0$, обозначает от одного до i вхождений r. Конечное замыкание всегда можно заменить перечислением возможностей; например, r^3 – это просто $(\epsilon|r|rr|rrr)$. *Положительное замыкание*, обозначаемое r^+, – это просто rr^*, оно состоит из одного или более вхождений r. Поскольку конечное и положительное замыкания можно переписать в терминах альтернации, конкатенации и замыкания Клини, мы в дальнейшем не будем к ним возвращаться.

Используя альтернацию, конкатенацию и замыкание Клини, мы можем определить множество РВ над алфавитом Σ следующим образом:

1. Если a ∈ Σ, то a – это РВ, обозначающее множество {a}, и $L(a)$ совпадает с a.
2. Если r и s – РВ, обозначающие множества $L(r)$ и $L(s)$ соответственно, то
   $r|s$ – РВ, обозначающее альтернацию $L(r)$ и $L(s)$;
   rs – РВ, обозначающее конкатенацию $L(r)$ и $L(s)$;

r^* – РВ, обозначающее замыкание Клини $L(r)$.

3. ϵ – РВ, обозначающее множество, содержащее только пустую строку.

Для исключения неоднозначностей считается, что скобки имеют наивысший приоритет, а за ними следуют замыкание, конкатенация и альтернация в указанном порядке.

2.3.2. Примеры регулярных выражений

Цель этой главы – показать, как можно использовать формальные методы для автоматизации построения высококачественных сканеров и как можно включить в этот формализм микросинтаксис языков программирования. Но прежде чем двигаться дальше, приведем несколько примеров из реальных языков программирования.

1. Приведенное выше правило формирования идентификаторов в Algol-подобных языках – буква, за которой следуют нуль или более букв или цифр, – это просто ([*A*...*Z*]|[*a*...*z*])([*A*...*Z*]|[*a*...*z*]|[*0*...*9*])*.

   Большинство языков допускают в идентификаторах также некоторые специальные символы, например _, %, $, или &.

   Если язык ограничивает длину идентификатора, то можно использовать конечное замыкание, например ([*A*...*Z*]|[*a*...*z*])([*A*...*Z*]|[*a*...*z*]|[*0*...*9*])^5 для шестизначного идентификатора.
2. Целое число без знака можно описать как 0 или отличную от нуля цифру, за которой следует нуль или более цифр. РВ *0*|[*1*...*9*][*0*...*9*]* короче. Более простая спецификация [*0*...*9*]$^+$ допускает целые числа с начальными нулями, например 001.
3. Вещественные числа без знака сложнее целых. Одно из возможных РВ имеет вид (*0*|[*1*...*9*][*0*...*9*]*) (ϵ|.[*0*...*9*]*). Первая часть – это просто РВ для целого. Остаток порождает либо пустую строку, либо десятичную точку, за которой следует одна или более цифр.

   Языки программирования часто допускают научную нотацию, например

   (*0*|[*1*...*9*][*0*...*9*]*)(ϵ|.[*0*...*9*]*)*E*(ϵ|+|–)(*0*|[*1*...*9*][*0*...*9*]*).

   Это РВ описывает вещественное число, за которым следует *E* и целое число, определяющее показатель степени.

   Дополнение
   РВ ^*c* определяет множество {Σ – *c*}, т. е. дополнение *c* относительно Σ. Приоритет дополнения выше, чем у *, | и +.
4. Со строками литер в кавычках связана отдельная сложность. В большинстве языков строка может содержать любые литеры. И хотя мы можем написать РВ для строк, пользуясь только базовыми операторами, следующий пример будет первым, в котором оператор дополнения упрощает РВ. Пользуясь дополнением, строку литер в C или JAVA можно описать как «(^»)*».

Управляющая последовательность
Две или более литер, которые сканер транслирует в какую-то другую литеру. Управляющие последовательности используются для литер, не имеющих графического представления (глифа), например для знаков новой строки или табуляции, а также для синтаксически значимых литер, таких как открывающая или закрывающая кавычка.

С и C++ не разрешают строке занимать несколько строчек текста[1], т. е. если сканер достигает конца строчки внутри строки, то он завершает строку и выдает сообщение об ошибке. Если представить знак конца строки управляющей последовательностью \n, в стиле C, то РВ «(^(«|\n))*» распознает правильно сформированную строку C или C++ и произведет переход в состояние ошибки для строки, содержащей знак новой строки.

Регулярные выражения в виртуальной жизни

Регулярные выражения используются во многих приложениях для описания образцов в строках литер. Некоторые ранние работы по трансляции РВ в код преследовали цель предоставить гибкий способ задания строк в команде «find» текстового редактора QED. Оттуда нотация проникла во многие приложения.

В Unix и других операционных системах звездочка используется в качестве метасимвола для сопоставления с подстроками в именах файлов. При таком применении * – сокращенная запись РВ Σ*, которое описывает одну или более литер, взятых из всего алфавита допустимых литер. (Поскольку лишь немногие клавиатуры имеют клавишу Σ, сокращенная нотация прижилась.) Во многих системах литера ? используется в качестве метасимвола, сопоставляемого с одной произвольной литерой.

В семействе утилит `grep` и ее аналогах в системах, отличных от Unix, реализовано сопоставление с образцами, заданными регулярным выражением (grep – акроним «global regular expression pattern»).

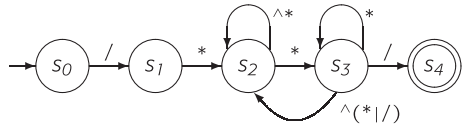
Регулярные выражения нашли широкое применение, потому что их легко написать и понять. Это один из предпочтительных методов распознавания фиксированного словаря в программе. Они хорошо работают для языков, подчиняющихся их ограниченным правилам. Они легко транслируются в исполняемую форму, получающийся при этом распознаватель работает быстро.

5. Комментарии могут принимать разные формы. C++ и JAVA предлагают два способа записи комментария. Комбинация // означает комментарий, продолжающийся до конца текущей входной строки. РВ для такого комментария тривиально: // (^\n)* \n.

   Многострочные комментарии в C, C++ и JAVA начинаются ограничителем /* и заканчиваются ограничителем */. Если бы мы могли запретить * в комментарии, то РВ было бы простым: /*(^*)**/. Если же * разрешена, то РВ сложнее: /*(^*|*+^/)**/.

   КА, реализующий это РВ, имеет вид:

[1] Печально, но английские слова *line* и *string* переводятся на русский одним словом *строка*. Поэтому в переводе возникают такие уродцы. Надеемся, что смысл понятен из контекста. – *Прим. перев.*

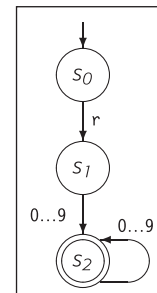


Связь между РВ для многострочных комментариев и его КА не столь очевидна, как в предшествующих примерах.

Сложность РВ и КА для многострочных комментариев проистекает из использования многолитерных ограничителей. Переход из s_2 в s_3 кодирует тот факт, что распознаватель видел * и потому может правильно обработать как наличие литеры /, так и ее отсутствие.

С другой стороны, в языке PASCAL используются однолитерные ограничители комментариев: { и }, поэтому РВ для комментариев совсем простое: {^}*}.

Во многих случаях более жесткие спецификации приводят к более сложным РВ. Рассмотрим, к примеру, спецификатор регистра в типичном языке ассемблера. Он состоит из буквы `r`, за которой следует небольшое целое число. В языке ILOC, допускающем неограниченное множество имен регистров, РВ можно было бы записать в виде $r[0...9]^+$, ему соответствует КА, показанный во врезке. Этот КА допускает `r29` и отвергает `s29`. Он также допускает `r99999`, хотя ни у какого современного процессора нет 100 000 регистров.

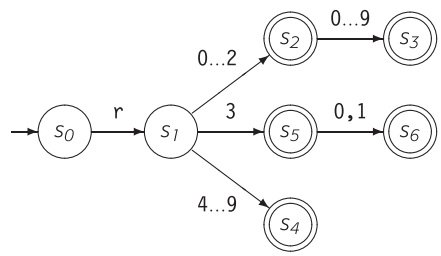


КА для имен регистров

В типичном процессоре множество имен регистров строго ограничено – скажем, 32, 64, 128 или 256 регистрами. С помощью более сложного РВ сканер может проверить допустимость имени регистра. Например, РВ

$$r([0...2]([0...9] \mid \epsilon) \mid [4...9] \mid (3(0 \mid 1 \mid \epsilon)))$$

описывает гораздо меньший язык. Номера регистров ограничены диапазоном [0, 31], и для однозначных имен регистров допускается необязательный начальный 0. Таким образом, оно допускает имена `r0`, `r00`, `r01` и `r31`, но отвергает `r001`, `r32` и `r99999`. Соответствующий КА выглядит так:



Какой КА лучше? Оба выполняют один переход при чтении каждой литеры. Следовательно, их стоимость одинакова, хотя второй КА проверяет более

Альтернативой, конечно, является использование РВ *r*[*0*...*9*][*0*...*9*] с последующей проверкой номера регистра как целого числа.

сложную спецификацию. Более сложный КА имеет больше состояний и переходов, поэтому его представление занимает больше места. Однако операционные затраты одинаковы.

Это критический момент: стоимость работы КА пропорциональна длине входа, а не сложности РВ и не числу состояний КА. Больше состояний требуют больше памяти, но не больше времени. Стоимость *генерирования* КА для более сложного РВ может быть выше, но операционные затраты – все равно один переход на каждую входную литеру. Сложность обработки одного перехода при хорошей реализации равна O(1).

Можно ли улучшить РВ для имени регистра? Предыдущее РВ было сложным и интуитивно неочевидным. Вот более простая альтернатива:

r0 | *r00* | *r1* | *r01* | *r2* | *r02* | *r3* | *r03* | *r4* | *r04* | *r5* | *r05* | *r6* | *r06* | *r7* | *r07* | *r8* | *r08* | *r9* | *r09* | *r10* | *r11* | *r12* | *r13* | *r14* | *r15* | *r16* | *r17* | *r18* | *r19* | *r20* | *r21* | *r22* | *r23* | *r24* | *r25* | *r26* | *r27* | *r28* | *r29* | *r30* | *r31*

Это РВ концептуально проще, но гораздо длиннее. Впрочем, результирующий КА все равно выполняет один переход на каждую входную литеру. Таким образом, если мы можем контролировать рост числа состояний, то предпочли бы эту версию РВ за ее чистоту и очевидность. Однако когда процессоры обзаведутся 256 или 384 регистрами, перечислить их будет утомительно.

2.3.3. Свойства замкнутости РВ

Регулярные языки
Любой язык, который можно описать регулярным выражением, называется *регулярным*.

РВ и порождаемые ими языки подверглись тщательному изучению. Они обладают многими интересными и полезными свойствами. Некоторые из этих свойств играют важную роль в построении распознавателей по РВ.

Множество РВ замкнуто относительно многих операций, т. е. результатом применения операции к РВ или набору РВ снова является РВ. Очевидные примеры – конкатенация, объединение и замыкание. Конкатенацией двух РВ *x* и *y* является просто *xy*. Их объединением является *x* | *y*. Замыканием Клини РВ *x* выступает *x**. По определению РВ, все эти выражения также являются РВ.

Эти свойства замкнутости критически важны при использовании РВ для построения сканеров. Пусть имеются РВ для каждой синтаксической категории исходного языка, a_0, a_1, a_2, ..., a_n. Тогда, чтобы построить РВ для всех допустимых слов языка, мы можем соединить их с помощью альтернации $a_0 \mid a_1 \mid a_2 \mid \ldots \mid a_n$. Поскольку множество РВ замкнуто относительно объединения, результатом будет РВ. Все, что мы можем делать с РВ для одной синтаксической категории, можно проделать и для РВ для всех допустимых слов языка.

Из замкнутости относительно объединения следует, что любой конечный язык регулярен. Мы можем построить РВ для любого множества слов, просто перечислив их в большой альтернации. Замкнутость гарантирует, что альтернация является РВ и что соответствующий язык регулярен.

Программы фильтрации URL-адресов опираются на это свойство для построения быстрых распознавателей, основанных на КА.

Замкнутость относительно конкатенации позволяет строить сложные РВ из более простых путем конкатенации. Это свойство кажется очевидным. Оно дает нам возможность систематически строить РВ из кусочков. Замкнутость гарантирует, что *ab* является РВ, коль скоро таковыми являются *a* и *b*. Таким образом, любой метод, применимый к *a* или *b*, применим и к *ab*.

Языки программирования и естественные языки

Лексический анализ выявляет одно из тонких различий между языками программирования и такими естественными языками, как английский или португальский. В естественных языках связь между представлением слова – его написанием или пиктограммой – и его смыслом не очевидна. В английском *are* – глагол, тогда как *art* – существительное, хотя эти слова различаются только в последней букве. Кроме того, не все комбинации букв являются осмысленными словами. Например, *arz* совсем немного отличается от *are* и *art*, но такого слова в нормативном английском языке нет.

Сканер для английского языка мог бы использовать основанную на КА методику для распознавания потенциальных слов, поскольку все английские слова составлены из букв ограниченного алфавита. Но затем ему пришлось бы искать потенциальное слово в словаре, чтобы установить, является ли оно действительно словом. Если слово можно отнести только к одной части речи, то поиск в словаре определит и его категорию. Однако многие английские слова можно классифицировать как разные части речи. Например, *buoy* и *stress* могут быть как глаголом, так и существительным. Для таких слов определить, какой частью речи они являются, можно только с учетом контекста. В некоторых случаях понимания грамматического контекста достаточно для классификации слова. Но в других требуется понимать еще и смысл как самого слова, так и его контекста.

С другой стороны, в языке программирования слова почти всегда определены лексически. Так, любая строка вида [*1…9*][*0…9*]* является положительным целым числом. РВ [*a…z*]([*a…z*] | [*0…9*])* определяет подмножество идентификаторов языка Algol; *arz*, *are* и *art* – это всё идентификаторы, и чтобы установить этот факт, никакого поиска в словаре не нужно. Правда, некоторые идентификаторы могут быть зарезервированными ключевыми словами. Но и эти исключения можно определить лексически. Никакой контекст не требуется.

Это свойство – следствие осознанного решения при проектировании языка программирования. Тот факт, что текст слова однозначно определяет часть речи, упрощает сканирование и разбор и, очевидно, мало сказывается на выразительности языка. Некоторые языки разрешали частеречный дуализм слов – например, в PL/I нет зарезервированных ключевых слов. Но более поздние языки отказались от этой идеи, потому что сложность ее реализации перевешивала преимущества, которые несла дополнительная гибкость.

Множество РВ также замкнуто относительно замыкания Клини и конечного замыкания. Это свойство позволяет описывать специальные виды больших и даже бесконечных множеств с помощью конечных образцов. Замыкание Клини дает возможность описать бесконечное множество коротким конечным образцом; примерами могут служить целые числа и идентификаторы неограниченной длины.

Конечные замыкания позволяют описывать большие, но конечные множества с той же легкостью. В следующем разделе показана последовательность построений, создающая КА из РВ. В разделе 2.6 продемонстрирован алгоритм, который строит РВ по КА. В совокупности эти алгоритмы доказывают эквивалентность РВ и КА. Тот факт, что множество РВ замкнуто относительно альтернации, конкатенации и замыкания, играет основную роль в этих построениях.

Полный КА
КА, в который явно включены все переходы в состояние ошибки.

Эквивалентность между РВ и КА наводит на мысль и о других свойствах замкнутости. Например, по заданному полному КА можно построить КА, который распознает все слова *w*, не принадлежащие языку *L*(КА), – так называемое дополнение *L*(КА). Чтобы построить КА для дополнения, мы можем поменять обозначения допускающих и недопускающих состояний оригинального КА на противоположные. Поскольку КА и РВ эквивалентны, этот результат показывает, что РВ замкнуты относительно дополнения. И действительно, во многих системах, где используются РВ, имеется оператор дополнения, например ^ в `lex` и `flex`.

ИТОГИ РАЗДЕЛА

Регулярные выражения – краткая и мощная нотация для описания микросинтаксиса языков программирования. В основе РВ лежат три операции над конечным алфавитом: альтернация, конкатенация и замыкание Клини. Другие удобные операторы, такие как конечное замыкание, положительное замыкание и дополнение, выводятся из этих трех основных. Регулярные выражения и конечные автоматы эквивалентны; любое РВ можно реализовать конечным автоматом, а язык, допускаемый КА, можно описать с помощью РВ. В следующем разделе эта связь будет формализована.

ВОПРОСЫ

1. Шестизначный идентификатор можно описать конечным замыканием

   $([A\ldots Z] \mid [a\ldots z])([A\ldots Z] \mid [a\ldots z] \mid [0\ldots 9])^5$.

   Перепишите эту спецификацию, пользуясь только тремя основными операциями РВ: альтернацией, конкатенацией и замыканием Клини.

2. В языке PL/I строки начинаются и заканчиваются знаком кавычки, ". Между ними может быть нуль или более литер, взятых из некоторого алфавита Σ. Чтобы представить знак ", программист должен написать два таких знака подряд. Строку

   ```
   The quotation mark, ", should be typeset in italics.
   ```

   в программе на PL/I следовало бы записать так:

   ```
   "The quotation mark, "", should be typeset in italics."
   ```

   Спроектируйте РВ и КА для распознавания строк PL/I.

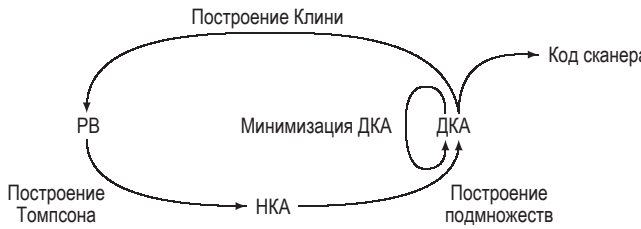


Рис. 2.3 Цикл построений

2.4. От регулярного выражения к сканеру

Цель нашей работы с конечными автоматами – автоматизировать вывод сканеров из набора РВ. В этом разделе мы разработаем построения для преобразования РВ в КА. Построения опираются на *недетерминированные* КА, или НКА, и на *детерминированные* КА, или ДКА. Построение Клини, кратко описанное в разделе 2.6.1, строит РВ по КА. Вместе эти построения образуют цикл, показанный на рис. 2.3.

Цикл построений демонстрирует, что РВ и КА имеют эквивалентную выразительную силу. То есть с помощью РВ можно выразить любой язык, распознаваемый КА, а КА может распознать любой язык, который можно описать с помощью РВ.

В разделе 2.4.1 объясняется разница между НКА и ДКА В разделе 2.4.2 представлено построение Томпсона, которое строит НКА непосредственно по РВ. В разделе 2.4.3 представлен алгоритм построения подмножеств, который строит ДКА, моделирующий НКА. В разделе 2.4.4 приведен алгоритм Хопкрофта минимизации ДКА; альтернативный алгоритм минимизации, предложенный Бжозовским, приведен в разделе 2.6.3.

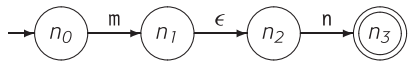
2.4.1. Недетерминированные конечные автоматы

Напомним, что мы определили пустую строку ϵ как РВ. Ни один КА, построенный нами вручную, не включал ϵ, но в некоторых РВ она встречалась. Какую роль ϵ играет в КА? Мы можем использовать переходы по ϵ, чтобы объединить КА и упростить построение КА из РВ. Например, предположим, что имеются КА для распознавания m и n, названные FA_m и FA_n.



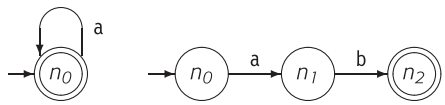
Мы можем построить КА для mn, добавив переход по ϵ из допускающего состояния FA_m в начальное состояние FA_n, перенумеровав состояния и воспользовавшись допускающим состоянием FA_n как допускающим состоянием нового КА.

ϵ-переход
переход по пустой строке ϵ, которая не продвигает вход.

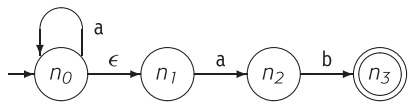


С появлением ϵ-перехода определение допустимости следует немного изменить, разрешив один или более ϵ-переходов между любыми двумя литерами входной строки. Например, в состоянии n_1 КА совершает переход $n_1 \xrightarrow{\epsilon} n_2$, не потребляя ни одной входной литеры. Изменение мелкое и интуитивно понятное. Однако оно означает, что КА с ϵ-переходами может совершать несколько переходов при чтении одной входной литеры.

Объединение двух КА с помощью ϵ-перехода может усложнить нашу модель работы КА. Рассмотрим КА для языков a^* и ab.



Мы можем объединить их ϵ-переходом и образовать КА для a^*ab.



ϵ-переход, по сути дела, наделяет КА двумя разными переходами из n_0 по букве a. Он может совершить переход $n_0 \xrightarrow{a} n_0$ или два перехода $n_0 \xrightarrow{\epsilon} n_1$ и $n_1 \xrightarrow{a} n_2$. Какой переход правильный? Рассмотрим строки aab и ab. КА должен допускать обе строки. Для aab он должен пройти по пути $n_0 \xrightarrow{a} n_0$, $n_0 \xrightarrow{\epsilon} n_1$, $n_1 \xrightarrow{a} n_2$ и $n_2 \xrightarrow{b} n_3$. Для ab он должен пройти по пути $n_0 \xrightarrow{\epsilon} n_1$, $n_1 \xrightarrow{a} n_2$, и $n_2 \xrightarrow{b} n_3$.

Недетерминированный КА
КА, в котором функция переходов может быть многозначной и включать ϵ-переходы.

Детерминированный КА
КА, в котором функция переходов однозначная и не включает ϵ-переходов.

Как показывает пример этих двух строк, какой переход из n_0 правилен, зависит от литер, следующих за a. На каждом шаге КА анализирует текущую литеру. Его состояние кодирует левый контекст, т. е. уже обработанные литеры. Поскольку КА должен совершить переход еще до анализа следующей литеры, такое состояние, как n_0, идет вразрез с нашим представлением о поведении последовательного алгоритма. КА, включающий состояния типа n_0, из которых исходит несколько переходов по одному символу, называется недетерминированным конечным автоматом (НКА). Напротив, КА, в котором из каждого состояния исходит только один переход, называется детерминированным конечным автоматом (ДКА).

Чтобы придать смысл НКА, нам понадобится набор правил, описывающий его поведение. Исторически для этой цели были предложены две разные модели.

1. Всякий раз, как НКА должен сделать недетерминированный выбор, он выбирает переход, который ведет в допускающее состояние для входной строки, если такой переход существует. Эта модель, в которой используется всеведущий НКА, привлекательна тем, что реализует (на первый взгляд) точно определенный допускающий алгоритм ДКА. По существу, НКА в каждой точке угадывает правильный переход.
2. Всякий раз, как НКА должен сделать недетерминированный выбор, он клонирует себя, чтобы испробовать все возможные переходы. Таким образом, для заданной входной литеры НКА и его клоны находятся в одном и том же множестве состояний. В этой модели НКА проходит по всем путям одновременно.

   В любой точке множество состояний, в которых НКА активен, называется его *конфигурацией*. Если НКА достигает конфигурации, в которой вход исчерпан, а один или несколько клонов перешли в допускающие состояния, то считается, что НКА допускает строку.

Конфигурация НКА
множество одновременно активных состояний НКА.

В обеих моделях НКА $(S, \epsilon, \delta, n_0, S_A)$ допускает входную строку $x = x_1 x_2 x_3 \ldots x_k$ тогда и только тогда, когда существует по меньшей мере один путь в диаграмме переходов, который начинается в n_0 и заканчивается в некотором n_m, таком что метки ребер, образующих этот путь, с опущенными ϵ совпадают с входной строкой. Иными словами, метка i-го ребра должна быть равна x_i. Это определение согласуется с обеими моделями поведения НКА.

Эквивалентность НКА и ДКА

С точки зрения выразительной силы, НКА и ДКА эквивалентны. Любой ДКА является частным случаем НКА. Следовательно, НКА по меньшей мере так же выразительны, как ДКА. Любой НКА можно смоделировать с помощью ДКА. Интуитивно идея проста, а построение, описанное в разделе 2.4.3, лишь немногим сложнее.

Рассмотрим состояние НКА в момент, когда он достиг некоторой точки входной строки. При второй модели поведения НКА имеет конечное множество работающих клонов. Множество состояний, занятых этими клонами, образует *конфигурацию* НКА. Каждая конфигурация – это подмножество N множества состояний НКА. Количество таких подмножеств конечно. Следовательно, НКА с N состояниями порождает не более $|2^N|$ конфигураций.

Степенное множество *N*
множество всех подмножеств N, обозначаемое 2^N.

Чтобы смоделировать поведение НКА, нам необходим ДКА, содержащий по одному состоянию для каждой допустимой конфигурации НКА. В результирующем ДКА может быть экспоненциально больше состояний, чем в НКА. И тем не менее количество конфигураций, а значит, и состояний ДКА конечно. ДКА по-прежнему совершает один переход на каждую входную литеру, поэтому время его работы пропорционально длине входной строки. Таким

образом, при моделировании НКА с помощью ДКА возможна проблема с памятью, но не со временем.

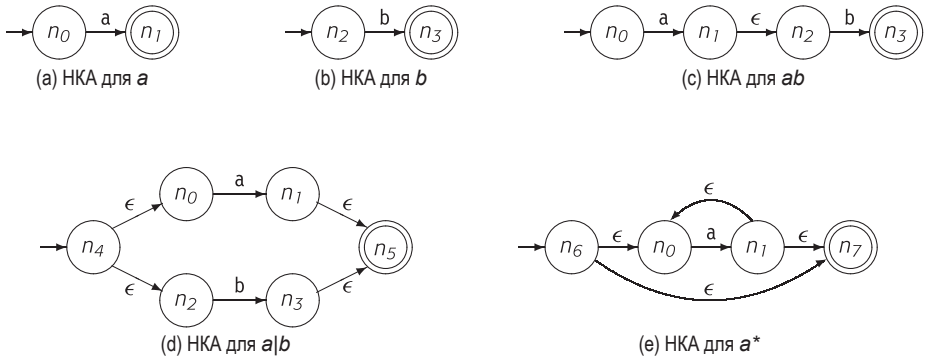
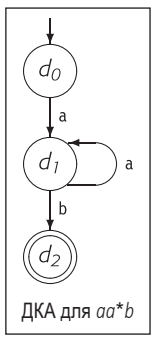


Рис. 2.4 Тривиальные НКА для операторов регулярных выражений



Поскольку НКА и ДКА эквивалентны, мы можем построить ДКА, распознающий a^*ab. В НКА, который мы видели раньше, было два перехода из n_0 по литере a. Очевидный способ избежать этого недетерминированного перехода – заметить, что a^*ab эквивалентно aa^*b. С этим РВ ассоциирован ДКА, показанный во врезке. Алгоритм построения подмножеств автоматизирует преобразование НКА в ДКА (см. раздел 2.4.3).

2.4.2. РВ в НКА: построение Томпсона

Чтобы поместить построение Томпсона в контекст, см. врезку на стр. 741 в разделе 14.2.

На первом шаге вывода сканера из РВ по этому РВ строится НКА, для чего применяется *построение Томпсона*. В построении используется простой управляемый шаблоном процесс формирования НКА из меньших НКА. Сначала строятся НКА для отдельных литер $s \in \Sigma$, а затем к получившимся НКА применяются преобразования, моделирующие конкатенацию, альтернацию и замыкание. На рис. 2.4 показаны тривиальные НКА для a и b, а также преобразования для получения ab, $a|b$ и a^* из НКА для a и b. Эти преобразования применимы к произвольным НКА.

На рис. 2.5 показаны шаги построения Томпсона для создания НКА по РВ $a(b|c)^*$. Сначала строятся тривиальные НКА для каждой литеры, встречающейся в РВ, как показано на рисунке (a). Затем применяются операторы в порядке предшествования. На рисунке (b) показано построение НКА для выражения в скобках, $(b|c)$. Затем на рисунке (c) показано построение замыкания. Наконец, на рисунке (d) показана конкатенация НКА для a с НКА для

$(b \mid c)^*$. Этот простой процесс порождает НКА для РВ, записанного в терминах трех базовых операторов.

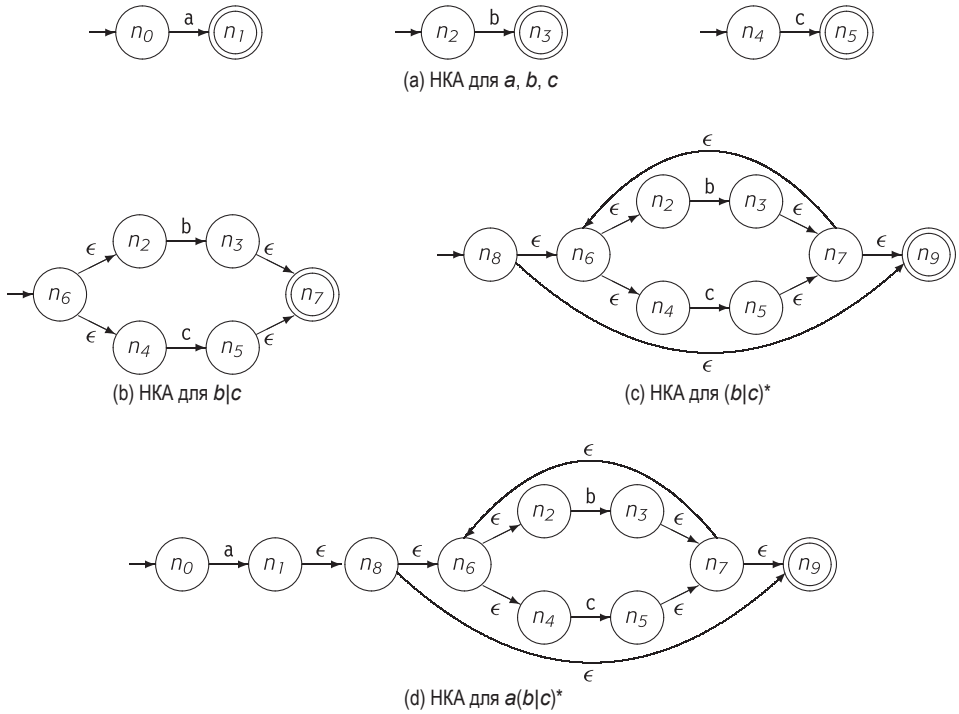
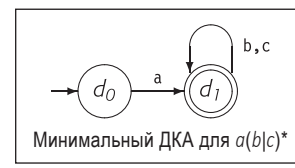


Рис. 2.5 Применение построения Томпсона к РВ $a(b|c)^*$

У НКА, выведенных с помощью построения Томпсона, есть несколько специальных свойств, упрощающих реализацию. У каждого НКА имеется одно начальное состояние и одно допускающее состояние. В начальное состояние не ведет никакой переход, кроме начального. Из допускающего состояния не исходит никакой переход. Наконец, для каждого состояния имеется не более двух входящих и двух исходящих ϵ-переходов и не более одного входящего и одного исходящего перехода по литере алфавита. В сочетании эти свойства упрощают представление НКА и операции с ними.

На рис. 2.5(d) показан НКА, созданный построением Томпсона для РВ $a(b \mid c)^*$. Комбинация построения подмножеств и минимизации ДКА должна преобразовать НКА на рис. 2.5(d) в ДКА, подобный показанному во врезке.



2.4.3. НКА в ДКА: построение подмножеств

Построение Томпсона порождает НКА для распознавания языка, описанного РВ. Поскольку исполнение ДКА смоделировать гораздо проще, чем исполнение НКА, следующий шаг построения распознавателя РВ заключается в пре-

образовании НКА в эквивалентный ДКА. Алгоритм построения ДКА по НКА называется *построением подмножеств*.

```
q0 ← FollowEpsilon({n0})
Q ← q0
WorkList ← {q0}

while (WorkList ≠ ∅) do
   удалить q из WorkList
   for each литеры c ∈ Σ do
      temp ← FollowEpsilon(Delta(q, c))
      if temp ≠ ∅ then
         if temp ∉ Q then
          добавить temp в Q и WorkList
         T[q, c] ← temp
```

Рис. 2.6 Построение подмножеств

Допустимая конфигурация
конфигурация НКА, которой можно достичь при какой-то допустимой входной строке.

Алгоритм построения подмножеств принимает на входе НКА, (N, Σ, δ_N, n_0, N_A). Он строит модель, которая запоминает все допустимые конфигурации, в которые НКА может попасть при обработке входных строк. Каждая конфигурация содержит одно или несколько состояний НКА, и все они достижимы из одного и того же набора входных строк. Модель состоит из двух множеств, Q и T. Каждый элемент q_i множества Q – это множество состояний НКА, которое представляет допустимую конфигурацию оригинального НКА. T моделирует переходы между конфигурациями, которые были обнаружены во время построения Q алгоритмом.

Чтобы построить ДКА по модели, мы создаем состояние ДКА d_i для каждого $q_i \in Q$. Переходы ДКА непосредственно вытекают из переходов, зарегистрированных в T. Если q_j был построен в результате рассмотрения того, какой переход НКА в конфигурации q_i совершил бы по литере c, то T содержит переход $q_i \xrightarrow{c} q_j$ и $\delta_N(d_i, c)$ должно быть равно d_j.

Алгоритм показан на рис. 2.6. Он начинает работу с одной конфигурацией q_0, построенной по n_0. Он добавляет q_0 в Q и помещает ее в *WorkList*. Рабочий список *WorkList* нужен для отслеживания тех q_i, которые должны быть обработаны в будущем. Алгоритм в цикле выбирает конфигурацию q_i из рабочего списка и удаляет ее. Затем для каждой $c \in \Sigma$ он вычисляет допустимую конфигурацию *temp*, достижимую по переходам из q_i по литере c. Если $temp \notin Q$, то она добавляется в Q и *WorkList*. Алгоритм запоминает переход из q_i в *temp* в множестве T. Алгоритм останавливается, когда рабочий список опустеет.

Каждое множество $q_i \in Q$ содержит базовые состояния и нуль или более небазовых состояний. Небазовые состояния достижимы из базового по одному или нескольким ϵ-переходам. В начальной конфигурации модели, q_0, имеется только одно базовое состояние – начальное состояние НКА n_0. Для построения q_0 алгоритм добавляет все состояния НКА, вытекающие из n_0, т. е. те, что

достижимы по путям, состоящим из ϵ-переходов. Функция *FollowEpsilon*(*s*) расширяет множество *s*, добавляя в него небазовые элементы.

Для вычисления конфигурации НКА, достижимой из q_i по литере *c*, алгоритм применяет δ_N к каждому $n_x \in q_i$. Функция *Delta*(*s*, *c*) вычисляет базу новой конфигурации по *s* и *c*. Если $\delta_N(n_x, c) = n_y$ и n_y не является состоянием ошибки, то n_y – базовое состояние новой конфигурации. Алгоритм вызывает *FollowEpsilon*, чтобы добавить вытекающие небазовые элементы.

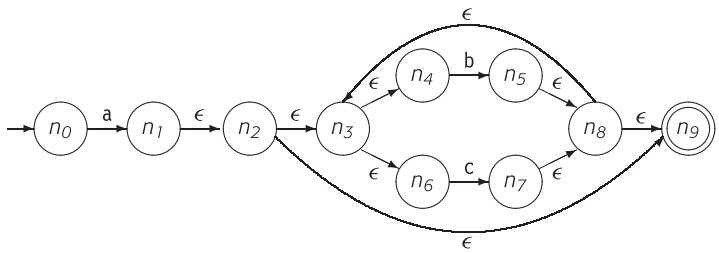
Когда построение завершится, *Q* будет содержать все допустимые конфигурации НКА, а *T* – все допустимые переходы между этими конфигурациями. Вместе они представляют ДКА, моделирующий исходный НКА. Чтобы создать экземпляр этого ДКА, мы создаем состояние d_i для каждого множества $q_i \in Q$. Если q_i содержит допускающее состояние НКА, т. е. такое n_j, что $n_j \in N_A$, то d_i, представляющее q_i, является допускающим состоянием, т. е. $d_i \in D_A$. Функция переходов ДКА строится непосредственно по *T* путем отображения множеств, принадлежащих *Q*, на их состояния ДКА. Наконец, q_0, множество, построенное по n_0, становится d_0, начальным состоянием ДКА.

Заметим, что *Q* растет монотонно. В цикле while множества добавляются в *Q*, но оттуда никогда не удаляются. Поскольку количество конфигураций НКА ограничено – каждое q_i является подмножеством 2^N (степенного множества *N*) – и каждое q_i встречается в рабочем списке ровно один раз, цикл while должен остановиться.

Q может оказаться большим – содержать $|2^N|$ различных состояний. Сколько будет расширений состояния, зависит от степени недетерминированности НКА. Напомним, однако, что результатом является ДКА, поэтому он будет совершать ровно один переход для каждой прочитанной литеры независимо от числа состояний. Таким образом, расширение, имеющее место при построении подмножеств, не влияет на асимптотическое время работы ДКА. Если структуры данных, используемые для представления ДКА, становятся достаточно большими, то может возникнуть проблема локальности памяти, которая ведет к снижению производительности. По счастью, минимизация ДКА и сжатие таблицы могут смягчить эти эффекты (см. разделы 2.4.4 и 2.5.4).

Пример

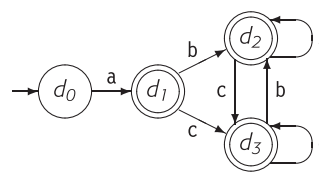
На рис. 2.7(a) показан НКА, являющийся результатом построения Томпсона для РВ $a(b \mid c)^*$, после перенумерации состояний слева направо. В таблице на рис. 2.7(b) перечислены шаги построения подмножеств для этого НКА. В первом столбце находится имя множества, принадлежащего *Q*, которое обрабатывается на данной итерации цикла while. Во втором столбце находится имя соответствующего состояния нового ДКА. В третьем столбце показано множество состояний НКА, содержащихся в текущем множестве из *Q*. В последних трех столбцах показан результат применения функции *FollowEpsilon(Delta(q_i, x))* к текущему множеству q_i и каждой литере $x \in \Sigma$.



(a) Исходный НКА

Имя множества	Состояние ДКА	Состояние НКА	FollowEpsilon(Delta(q, x)) a	b	c
q_0	d_0	$\{ n_0 \}$	$\{n_1, n_2, n_3, n_4, n_6, n_9\}$	– none –	– none –
q_1	d_1	$\{n_1, n_2, n_3, n_4, n_6, n_9\}$	– нет –	$\{n_5, n_8, n_9 n_3, n_4, n_6\}$	$\{n_7, n_8, n_9 n_3, n_4, n_6\}$
q_2	d_2	$\{n_5, n_8, n_9 n_3, n_4, n_6\}$	– нет –	q_2	q_3
q_3	d_3	$\{n_7, n_8, n_9 n_3, n_4, n_6\}$	– нет –	q_2	q_3

(b) Итерации построения подмножеств



(c) Результирующий ДКА

Рис. 2.7 Применение алгоритма построения подмножеств к НКА на рис. 2.5

Алгоритм выполняет следующие шаги.

1. На этапе инициализации q_0 присваивается значение *FollowEpsilon(*{n_0}*)*, каковым является {n_0}.
2. На первой итерации *Delta* и *FollowEpsilon* применяются к q_0 с литерами *a*, *b* и *c*. При использовании *a* получается q_1, которое содержит шесть состояний НКА. Использование *b* и *c* дает пустое множество.
3. На второй итерации цикла в цикле while анализируется q_1. Использование *a* дает пустое множество, использование *b* дает q_2, а использование *c* – q_3.
4. На третьей итерации цикла while анализируется q_2. Использование *a* дает пустое множество, а использование *b* и *c* повторно строит q_2 и q_3.

5. На четвертой итерации цикла while анализируется q_3. И снова повторно строятся q_2 и q_3. Алгоритм останавливается, потому что *WorkList* пуст.

На рис. 2.7(c) показан результирующий ДКА. Его состояния соответствуют множествам, принадлежащим Q; таблица и переходы взяты из T. Каждое из подмножеств q_1, q_2 и q_3 является допускающим состоянием ДКА, потому что все они содержат n_9, допускающее состояние НКА.

Вычисление неподвижной точки

Монотонная функция
функция f с областью определения D называется монотонной, если $\forall\ x, y \in D\ x \leqslant y \Rightarrow f(x) \leqslant f(y)$.

Алгоритм построения подмножеств – пример *вычисления неподвижной точки*, которое регулярно возникает во многих разделах информатики. Такие вычисления характеризуются повторяющимся применением некоторой монотонной функции к набору множеств, выбираемых из области с известной структурой. Вычисление заканчивается по достижении состояния, в котором следующая итерация дает тот же ответ, что предыдущая, т. е. по достижении «неподвижной точки» в пространстве последовательных итераций. Вычисления неподвижной точки играют важную роль в конструировании компиляторов.

Доказательство завершения алгоритма вычисления неподвижной точки обычно зависит от свойств области определения. В алгоритме построения подмножеств областью определения D является множество подмножества N. Алгоритм строит множество Q такое, что любой элемент $q_i \in Q$ является подмножеством N. Поскольку N конечно, 2^N тоже конечно и количество различных элементов Q ограничено.

Цикл while работает как монотонно возрастающая функция, определенная на Q. Он добавляет элементы в Q, так что последовательные итерации порождают все большие аппроксимации Q. Если i-я аппроксимация Q равна Q_i, то алгоритм гарантирует, что $Q_i \leqslant Q_{i+1}$. Поскольку число элементов Q не превышает $|2^N|$, цикл while остановится не позднее, чем после $|2^N|$ итераций. Конечно, он может достичь неподвижной точки и остановиться быстрее.

Проблемой алгоритмов вычисления неподвижной точки является единственность результата. Например, влияет ли порядок, в котором алгоритм выбирает q из рабочего списка на конечное множество Q? В этом алгоритме монотонная функция применяет к множествам оператор объединения. Поскольку операция объединения коммутативна ($a \cup b = b \cup a$) и ассоциативна ($(a \cup b) \cup c = a \cup (b \cup c)$), порядок добавления множеств в Q не меняет конечный результат. Индексы, присвоенные конкретным элементам $q_i \in Q$, могут меняться в зависимости от порядка извлечения из рабочего списка, но конечное Q всегда будет содержать одни и те же множества допустимых конфигураций НКА. Различные варианты Q могут отличаться только именами множеств q_i.

2.4.4. ДКА в минимальный ДКА

На последнем шаге построения РВ → ДКА мы можем воспользоваться алгоритмом минимизации числа состояний автомата. Алгоритм построения подмножеств может порождать ДКА с большим числом состояний. И хотя размер ДКА не влияет на его асимптотическую сложность, от него существенно зависит потребление памяти распознавателем. В современных компьютерах скорость доступа к памяти часто определяет скорость вычисления. Небольшой распознаватель может поместиться в самом нижнем уровне кеш-памяти, что уменьшит среднее время доступа.

Чтобы сократить число состояний ДКА, генератор сканеров может применить алгоритм минимизации ДКА. Самым известным и асимптотически самым быстрым является алгоритм Хопкрофта, который строит минимальный ДКА из произвольного, объединяя эквивалентные состояния в группы. Два состояния ДКА эквивалентны, если они порождают одинаковое поведение на любой входной строке. Алгоритм находит наибольшие возможные множества эквивалентных состояний; каждое такое множество становится состоянием минимального ДКА.

Разбиение множества
Разбиением множества S называется набор непересекающихся непустых подмножеств S, объединение которых совпадает с S.

Алгоритм строит разбиение $P = \{p_1, p_2, p_3, \ldots, p_m\}$ множества состояний ДКА. Каждое p_i содержит множество эквивалентных состояний ДКА. Более формально, он строит разбиение с минимальным числом множеств такое, что выполняются следующие два правила:

1) $\forall c \in \Sigma$, если $d_i, d_j \in p_s$, $d_i \xrightarrow{c} d_x$, $d_j \xrightarrow{c} d_y$ и $d_x \in p_t$, то $d_y \in p_t$;
2) если $d_i, d_j \in p_k$ и $d_i \in D_A$, то $d_j \in D_A$.

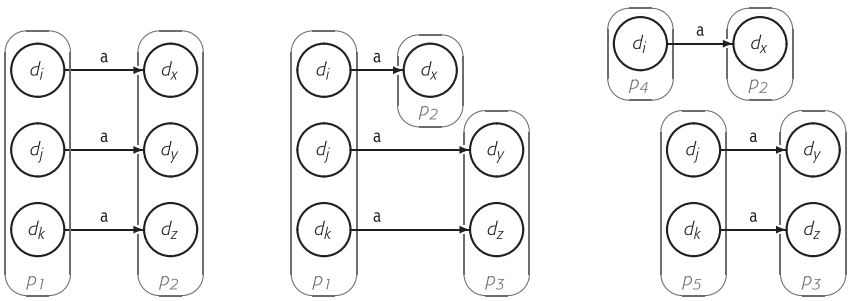
Правило 1 означает, что переходы из двух состояний, принадлежащих одному множеству, для каждой литеры $c \in \Sigma$ должны вести в состояния, которые сами являются элементами одного и того же подмножества разбиения. Правило 2 означает, что любое подмножество содержит либо допускающие состояния, либо недопускающие, но не те и другие вместе.

P_0 разбивает D на допускающие и недопускающие состояния, в этом фундаментальное отличие от поведения, выраженного правилом 2.

Эти два свойства не только налагают ограничение на конечное разбиение P, но также ведут к построению P. В начале работы создается самое грубое разбиение, $P_0 = \{D_A, \{D - D_A\}\}$. Затем алгоритм итеративно «измельчает» разбиение, пока оба свойства не станут истинными для каждого множества в P. Чтобы измельчить разбиение, алгоритм разбивает каждое множество на основе переходов из состояний ДКА, попавших в него.

На рис. 2.8 показано, как алгоритм использует переходы, чтобы измельчить множества разбиения. На рисунке (а) из всех трех состояний ДКА в множестве p_1 исходят переходы в состояния ДКА в p_2 по входной литере a. Конкретно $d_i \xrightarrow{a} d_x$, $d_j \xrightarrow{a} d_y$ и $d_k \xrightarrow{a} d_z$. Поскольку $d_i, d_j, d_k \in p_1$ и $d_x, d_y, d_z \in p_2$, то множества

p_1 и p_2 удовлетворяют правилу 1. Таким образом, состояния в p_1 с точки зрения поведения эквивалентны на литере a, поэтому a не побуждает алгоритм измельчить p_1.



(a) a не измельчает p_1; (b) a измельчает p_1 (c) множества на рисунке b после измельчения

Рис. 2.8 Измельчение множества по a

С другой стороны, на рисунке (b) показана ситуация, когда литера a индуцирует измельчение множества p_1. Как и раньше, $d_i \xrightarrow{a} d_x$, $d_j \xrightarrow{a} d_y$ и $d_k \xrightarrow{a} d_z$, но $d_x \in p_2$, а d_y, $d_z \in p_3$. Эта ситуация нарушает правило 1, поэтому a индуцирует измельчение p_1 на два множества, $p_4 = \{d_i\}$ и $p_5 = \{d_j, d_k\}$, показанных на рисунке (c).

Алгоритм, показанный на рис. 2.9, основан на этих идеях. Имея ДКА, он строит начальное разбиение множества его состояний. Начинает он с самого грубого разбиения, не нарушающего правило 2, $\{D_A, \{D - D_A\}\}$.

> Начать с наибольших возможных множеств и затем измельчать их – это *оптимистический* подход к построению множеств. Оптимизм обсуждается в разделе 9.3.6 и в работе [359].

Из такого выбора начального разбиения вытекают два следствия. Во-первых, поскольку каждое множество в конечном разбиении строится путем измельчения какого-то множества, вычисленного на предыдущей аппроксимации, гарантируется, что никакое множество в конечном разбиении не будет содержать одновременно допускающие и недопускающие состояния. Во-вторых, благодаря выбору наибольших множеств, согласованных с правилом 2, он налагает на процесс измельчения ограничения минимума, что, в свою очередь, может порождать разбиение на более крупные множества. (Чем больше множества, тем меньше состояний в конечном ДКА.)

Алгоритм оперирует рабочим списком состояний, начиная с начального разбиения $\{D_A, \{D - D_A\}\}$. Он в цикле выбирает множество s из рабочего списка и использует его для измельчения *Partition* путем разделения множеств на основе их переходов в s.

Чтобы идентифицировать множества состояний, которые необходимо измельчить из-за наличия перехода в множество s по некоторой литере c, алгоритм обращает функцию переходов. Он вычисляет *Image* как множество состояний ДКА, из которых можно достичь состояния, принадлежащего s, по переходу, помеченному c. Затем он систематически исследует каждое мно-

жество q, в котором имеется состояние из *Image*, чтобы понять, индуцирует ли *Image* измельчение q. Если *Image* разделяет q на непустые множества q_1 и q_2, то он заменяет q в *Partition* на q_1 и q_2.

```
Partition ← {D_A, {D – D_A}}
Worklist ← {DA, {D – D_A}}

while ( Worklist ≠ ∅ ) do
  выбрать множество s из Worklist и удалить его
  for each литера c ∈ Σ
  do
    Image ← { x | δ(x,c) ∈ s }
    for each множество q ∈ Partition, имеющее состояние в Image do
      q1 ← q ∩ Image
      q2 ← q – q1
      if q2 ≠ ∅ then                  // измельчить q по s и c
        удалить q из Partition
        Partition ← Partition ∪ q1 ∪ q2
        if q ∈ Worklist then          // и обновить Worklist
          удалить q из Worklist
          WorkList ← WorkList ∪ q1 ∪ q2
        else if |q1| ≤ |q2| then
          WorkList ← Worklist ∪ q1
        else WorkList ← WorkList ∪ q2
        if s = q then                 // нужно еще одно s
          break
```

Рис. 2.9 Алгоритм минимизации ДКА

Для завершения обработки q относительно c осталось только обновить рабочий список. Если q находится в списке, то алгоритм заменяет q на q_1 и q_2. Обоснование простое: q находилось в рабочем списке для достижения какого-то потенциального эффекта; возможно, этот эффект может быть достигнут в результате обработки какой-то литеры, отличной от c, поэтому все состояния ДКА, принадлежащие q, должны быть представлены в рабочем списке.

С другой стороны, если q не находится в рабочем списке, то единственный эффект, который измельчение q может оказать на другие множества, – это их измельчение. Предположим, что из какого-то множества r исходят переходы в q по литере e. Измельчение q могло бы создать необходимость разделения r на множества, из которых имеются переходы в q_1 и q_2. В таком случае либо q_1, либо q_2 индуцирует измельчение, так что алгоритм должен выбрать одно из них. При выборе меньшего множества алгоритм будет работать быстрее; например, вычисление *Image* занимает время, пропорциональное размеру множества.

Чтобы построить новый ДКА из конечного разбиения *Partition*, мы можем создать состояния для представления каждого множества $p_i \in$ *Partition* и добавить необходимые переходы между новыми репрезентативными состояниями. Для состояния, представляющего p_m, мы добавляем переход в состояние,

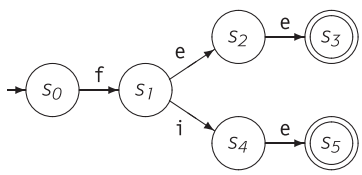
представляющее p_n, по литере c, если из какого-то $d_j \in p_m$ имеется переход в какое-то $d_k \in p_n$ по c. Из построения следует, что если $d_j \xrightarrow{c} d_k$, где $d_j \in p_m$ и $d_k \in p_n$, то из каждого состояния, принадлежащего p_m, имеется аналогичный переход по c в состояние, принадлежащее p_n. Если это условие не выполняется, то алгоритм измельчил бы p_m на основе переходов по c. Получившийся ДКА минимален, доказательство выходит за рамки книги.

Примеры

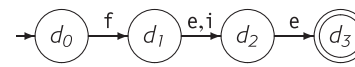
В качестве первого примера рассмотрим ДКА для *fee*|*fie*, показанный на рис. 2.10(b). На рисунке (a) показано продвижение алгоритма Хопкрофта для этого ДКА.

Шаг	Разбиение по входу	Worklist	s	c	Image	q	q_1	q_2	Действие
–	p_0: {s_3, s_5},	p_0, p_1	—	—	—	—	—	—	—
	p_1: {s_0, s_1, s_2, s_4}								
1	p_0: {s_3, s_5},	p_1	p_0	e	s_2, s_4	p_1	s_2, s_4	s_0, s_1	*разделить* $p_1 \rightarrow p_2, p_3$
	p_1: {s_0, s_1, s_2, s_4}	p_2, p_3	p_0	f	Ø	Ø	Ø	Ø	*нет*
		p_2, p_3	p_0	i	Ø	Ø	Ø	Ø	*нет*
2	p_0: {s_3, s_5}, p_2: {s_2, s_4},	p_3	p_2	e	s_1	p_3	s_1	s_0	*разделить* $p_3 \rightarrow p_4, p_5$
	p_3: {s_0, s_1}	p_4, p_5	p_2	f	Ø	Ø	Ø	Ø	*нет*
		p_4, p_5	p_2	i	s_1	Ø	Ø	Ø	*нет*
3	p_0: {s_3, s_5}, p_2: {s_2, s_4},	p_5	p_4	e	Ø	Ø	Ø	Ø	*нет*
	p_4: {s_1}, p_5: {s_0}	p_5	p_4	f	s_0	p_5	s_0	Ø	*нет*
		p_5	p_4	i	Ø	Ø	Ø	Ø	*нет*
4	p_0: {s_3, s_5}, p_2: {s_2, s_4},	Ø	p_5	e	Ø	Ø	Ø	Ø	*нет*
	p_4: {s_1}, p_5: {s_0}	Ø	p_5	f	Ø	Ø	Ø	Ø	*нет*
		Ø	p_5	i	Ø	Ø	Ø	Ø	*нет*

(a) Итерации алгоритма Хопкрофта применительно к ДКА для «fee | fie»



(b) Исходный ДКА для «fee | fie»



(c) Минимальный ДКА для «fee | fie»

Рис. 2.10 Применение алгоритма минимизации ДКА

В первой строчке показана начальная конфигурация алгоритма. *Partition* и *Worklist* содержат по два множества: {D_A, {$D - D_A$}}. D_A снабжено меткой p_0, а {$D - D_A$} – меткой p_1.

Алгоритм входит в цикл while и удаляет p_0 из *Worklist*; оно становится s. Алгоритм перебирает литеры алфавита Σ в порядке `e`, `f`, `i`. Для `e` p_0 разделя-

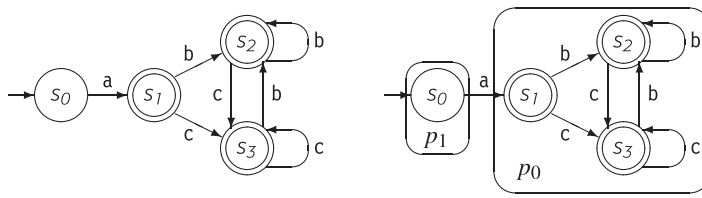
ет p_1 на два множества: p_2: {s_0, s_1} и p_3: {s_2, s_4}. Алгоритм удаляет p_1 из *Partition* и добавляет p_2 и p_3. Далее он удаляет p_1 из *Worklist* и добавляет p_2 и p_3. Для `f` и `i` ребер, входящих в p_0, не существует. Таким образом, *Image* пусто и никакого измельчения не происходит.

Вторая итерация аналогична. Алгоритм выбирает p_2. Литера `e` измельчает p_3 и вызывает обновление *Partition* и *Worklist*. Для `f` множество *Image* пусто. Для `i` множество *Image* содержит s_1. Поскольку алгоритм уже измельчил p_3 по `e`, эта ситуация не вызывает измельчения. Однако переход в конечный ДКА добавляется.

На третьей итерации из рабочего списка выбирается p_4. И `e`, и `i` порождают пустые множества *Image*. В случае `f` множество *Image* содержит s_0. Поскольку p_5, содержащее s_0, – одноэлементное множество, измельчить его невозможно. Но переход в конечный ДКА добавляется.

На последней итерации из рабочего списка выбирается p_5. Для каждой литеры `e`, `f`, `i` множество *Image* пусто. Следовательно, никаких измельчений и добавлений переходов не происходит, и рабочий список оказывается пустым. На рисунке (c) показан минимальный ДКА.

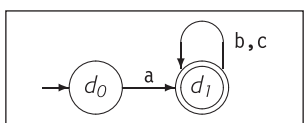
В качестве второго примера рассмотрим ДКА для РВ $a(b\,|\,c)^*$, порожденный построением Томпсона и алгоритмом построения подмножеств; он показан на рис. 2.11(a). Начальное разбиение имеет вид {p_0 : {s_1, s_2, s_3}, p_1 : {s_0}}.



(a) Исходный ДКА (b) Начальное разбиение

Рис. 2.11 ДКА для $a(b|c)^*$

Алгоритм сначала выбирает p_0 и по очереди исследует литеры `a`, `b` и `c`. Для `a` множество *Image* содержит s_0, которое принадлежит одноэлементному множеству p_1. Следовательно, `a` индуцирует переход в конечном ДКА, но не измельчение. Для `b` и `c` *Image* равно {s_1, s_2, s_3}, что в точности совпадает с p_0. Таким образом, q_2 пусто, и никакого измельчения не происходит.



Минимальный ДКА для $q(b|c)^*$

Далее алгоритм удаляет p_1 и исследует литеры `a`, `b` и `c`. Так как в p_1 ни один переход не входит, множество *Image* для всех литер пусто. Дополнительных измельчений нет. Первоначальное разбиение на два множества является окончательным. Конечный ДКА имеет два состояния, что показано во врезке. Напомним, что вывод именно такого ДКА мы ожидали от человека. Автоматический метод после минимизации дает такой же результат.

2.4.5. Использование ДКА в качестве сканера

Инструменты, рассмотренные в трех предыдущих разделах, прокладывают алгоритмический путь от РВ к ДКА. Как мы видели на рис. 2.2, ДКА можно смоделировать простой таблично управляемой структурой. В совокупности эти два факта позволяют предположить, что построение сканера можно автоматизировать, взяв РВ для всех слов языка программирования, объединив их в одно РВ и воспользовавшись получившимся ДКА. Реальность, однако, сложнее. Между сканерами и ДКА есть два принципиально важных отличия, которые влияют на то, как мы ставим и решаем задачу построения основанных на РВ сканеров.

Модель выполнения

ДКА читает весь свой ввод и допускает его, если последнее состояние допускающее. То есть ДКА пытается найти одно слово. С другой стороны, сканер читает достаточно текста, чтобы найти следующее слово во входном потоке. Сканер оставляет входной поток в состоянии, двигаясь из которого, он может найти следующее слово.

Это различие вынуждает искать новую модель выполнения. Вместо того чтобы исчерпывать входной поток, сканер моделирует ДКА, пока не столкнется с ошибкой, т. е. до тех пор, пока не окажется в состоянии d_i с входной литерой c такой, что $\delta(d_i, c) = s_e$, состоянию ошибки. Мы также определим $\delta(d_j, \texttt{eof}) = s_e \; \forall \; d_j \in D$.

Если d_i – допускающее состояние, $d_i \in D_A$, то сканер нашел слово. Если d_i – недопускающее состояние, то сканер, возможно, проходил через такое состояние на пути в d_i. Чтобы установить, так это или не так, сканер должен идти назад, по одной литере, пока либо не достигнет допускающего состояния, либо не исчерпает всю лексему.

Эта схема добавляет кое-какую новую работу. Сканер должен либо запоминать состояния, либо обратить δ. Любой подход занимает время, пропорциональное числу просканированных литер. Литера может сканироваться несколько раз; в разделе 2.5.1 описан метод, позволяющий избежать худшего случая при таком поведении.

Нахождение синтаксических категорий

ДКА возвращает бинарный ответ: вход либо допускается, либо отвергается. Сканер же возвращает токен, ⟨*лексема*, *категория*⟩, который содержит написание и синтаксическую категорию следующего слова. Если сканер обнаруживает ошибку, то может вернуть недопустимый токен.

Если мы построим такой ДКА, что каждое конечное состояние отображается в одну категорию, то сканер сможет найти категорию простым поиском в таблице.

> $d_i \in D_A$ однозначно отображается в категорию, но одна категория может отображаться в несколько d_i.

Однако такая схема требует строить ДКА, так чтобы сохранялось отображение конечных состояний в категории.

Идентификация ключевых слов

В большинстве языков программирования зарезервированы ключевые слова, обозначающие важные синтаксические конструкции, например `if`, `then` и `while`. В типичном сканере и парсере каждое ключевое слово имеет уникальную синтаксическую категорию, состоящую всего из одной лексемы. Автор компилятора стоит перед выбором: определить ключевое слово отдельным правилом или руководствоваться тем же правилом, что для всех идентификаторов, а распознавать ключевые слова каким-то другим способом. Оба подхода работают и приводят к эффективным сканерам.

Если для каждого ключевого слова определено свое правило, то сканер может вернуть подходящую категорию, пользуясь тем же механизмом, что для других категорий, таких как *число* или *идентификатор*. Дополнительные правила могут немного увеличить издержки генерирования сканера, а в ДКА может оказаться больше состояний. Но поскольку порождается ДКА, сложность результирующего сканера все равно будет равна O(1) на один символ.

Но большинство сканеров строят таблицу имен идентификаторов. Она служит отправной точкой для *таблицы символов* компилятора и позволяет отобразить имена идентификаторов в небольшие числа, чтобы их можно было эффективно представлять и сравнивать (см. раздел 4.5). Если автор компилятора заранее поместит в таблицу символов ключевые слова и их синтаксические категории, то сканер будет находить ключевые слова как ранее встречавшиеся и уже классифицированные идентификаторы и возвращать категорию каждого.

Большинство генераторов сканеров принимают на входе список РВ r_1, r_2, ..., r_k, каждое из которых определяет написание некоторой категории. Очевидный способ получить один ДКА – построить его по объединенному РВ, $(r_1 \mid r_2 \mid \ldots \mid r_k)$. Однако построение Томпсона первым делом сольет конечные состояния, что нарушит отображение $d_i \in D_A$ в категории.

Чтобы сохранить отображение конечных состояний в уникальные категории, генератор сканеров может строить отдельный НКА для каждого правила, применяя построение Томпсона. Затем он соединит эти НКА в один НКА с новым начальным состоянием с ϵ-переходами и воспользуется алгоритмом построения подмножеств, чтобы построить ДКА, моделирующий этот НКА. В результирующем ДКА может быть больше состояний, но каждое конечное состояние соответствует одному правилу, а значит, одной синтаксической категории.

Эта ситуация является признаком неоднозначности спецификации. Например, ключевое слово `then` может также отвечать правилу для идентификатора.

И `flex`, и `lex` назначают больший приоритет правилу, которое находится раньше в списке правил.

Если два правила пересекаются, то алгоритм построения подмножеств может объединить их конечные состояния. Если это происходит, то генератор сканеров должен решить, какую синтаксическую категорию возвращать для конечного состояния. На практике генераторы сканеров дают автору компилятора возможность задать приори-

тет синтаксической категории и назначают конечному состоянию категорию с наибольшим приоритетом.

Минимизация поднимает еще одну проблему. Алгоритм Хопкрофта сразу объединяет все конечные состояния в один раздел разбиения, сводя тем самым на нет свойство отображения конечных состояний в синтаксические категории. Но если генератор сканеров строит начальное разбиение, так чтобы конечные состояния для каждой синтаксической категории попадали в разные разделы, то при дальнейшей работе алгоритм сохраняет это свойство.

Алгоритм Хопкрофта измельчает разбиения, но никогда не укрупняет их.

Результирующий ДКА может оказаться больше минимального, получающегося, когда все конечные состояния группируются в один раздел разбиения. Однако больший ДКА обладает тем свойством, которое необходимо компилятору: каждое конечное состояние отображается в одну синтаксическую категорию.

Роль разделителей

Программисты часто называют пробелы и табуляторы, служащие для форматирования кода, *разделителями* (whitespace). В большинстве языков у разделителей нет какого-то специального смысла. Сканеры таких языков обычно распознают и отбрасывают их. Главная особенность разделителей заключается в том, что они не встречаются в РВ, определяющих слова языка.

Например, тот факт, что РВ для идентификатора или ключевого слова не включает ни пробелов, ни табуляторов, заставляет основанный на РВ сканер разделять `do` и `i` в таком предложении, как

```
do i = 1 to 100
```

По тем же причинам РВ для идентификаторов не содержит знаков `+`, `-`, `*` и `/`. Это гарантирует, что «`a * b`» сканируется так же, как «`a*b`».

FORTRAN 66

В языке FORTRAN 66 пробелы не значимы, т. е. «`n a m e`» и «`name`» – один и тот же идентификатор. Это правило усложняет сканирование. Заголовок цикла `do` в FORTRAN может выглядеть так:

```
do 10 i = 1,100
```

где `10` – метка последнего предложения в теле цикла, а `i` – индексная переменная цикла, принимающая значения от `1` до `100`. (Шаг по умолчанию равен 1, если явно не указано противное.)

Конечно, `do10i` – допустимое имя переменной. Чтобы различить предложения

```
do 10 i = 1
do 10 i = 1,100
```

сканер должен читать текст после `=` и `1` до запятой. Запятая показывает, что второе предложение – заголовок цикла, и сканер может разделить `do10i` на

три слова: `do`, `10` и `i`. Мало найдется языков (если вообще такие существуют), которые последовали за FORTRAN'ом в этом отношении.

PYTHON

В PYTHON принят противоположный подход: мало того, что пробелы значимы, так еще и количество пробелов в начале строке определяет смысл программы на PYTHON. Вместо того чтобы использовать фигурные скобки или операторные скобки `begin` и `end` для обозначения блочной структуры, PYTHON полагается на величину отступа.

Чтобы обработать начальные пробелы в PYTHON, можно просто добавить правило, которое распознает конец строки, за которым следует нуль или более пробелов. Тогда сканер может проверить длину лексемы. Если ее длина такая же, как в предыдущем токене этой категории, то он возвращает результат, понуждающий вызвать сканер еще раз. Если длины различаются, то сканер может вернуть категорию, обозначающую начало или конец блока, в зависимости от ситуации.

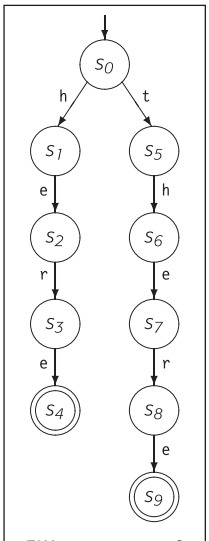
ИТОГИ РАЗДЕЛА

По заданному регулярному выражению мы можем вывести ДКА, распознающий описываемый им язык. Для этого нужно выполнить два шага: (1) применить построение Томпсона, порождающее НКА для РВ, и (2) воспользоваться алгоритмом построения подмножеств, который создает ДКА, моделирующий этот НКА. В результате получается быстрый и эффективный ДКА.

Для построения сканера, распознающего несколько категорий слов, каждое из которых описывается своим РВ, мы можем применить двухшаговый процесс, чтобы создать минимальный ДКА для каждой категории, а затем объединить эти ДКА в один НКА, добавив новое начальное состояние, из которого ϵ-переходы ведут в начальные состояния индивидуальных минимальных ДКА. Затем алгоритм построения подмножеств преобразует этот НКА в ДКА с различными конечными состояниями для каждой категории.

ВОПРОСЫ

1. Рассмотрим РВ *who* | *what* | *where*. Применив построение Томпсона, постройте НКА по этому РВ. Воспользовавшись алгоритмом построения подмножеств, постройте ДКА по этому НКА. Минимизируйте ДКА.
2. Минимизируйте ДКА, показанный во врезке.



ДКА для вопроса 2

2.5. Реализация сканеров

Генераторы сканеров применяют теорию формальных языков непосредственно к задаче создания эффективных токенизаторов для языков программирования. С помощью методов, описанных в предыдущем разделе, эти инструменты строят модели микросинтаксиса языка в виде ДКА и преобразуют их в исполняемый код.

В этом разделе мы обсудим три стратегии преобразования ДКА в исполняемый код. Первые две, таблично управ-

ляемый и непосредственно кодируемый сканер, основаны на *генераторах сканеров*, которые транслируют набор РВ в сканер. Третья стратегия – написание сканера вручную автором компилятора. Все подходы могут дать надежный и эффективный сканер.

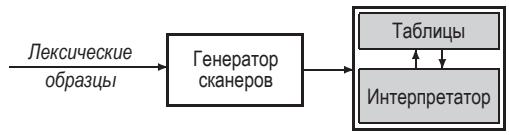
Сгенерированные сканеры моделируют ДКА, описанные в разделе 2.4.5. Они начинают работу в начальном состоянии и совершают ряд переходов на основе текущего состояния и входной литеры. Если из текущего состояния нет перехода по данной литере, то сканер возвращается назад, пока не найдет допускающее состояние. Если найти допускающее состояние не удается, то сканер сообщает о лексической ошибке.

В трех следующих подразделах обсуждаются различия в реализации таблично управляемых, непосредственно кодируемых и написанных вручную сканеров. Стратегии различаются способом моделирования структуры переходов ДКА и его работы. Каждый подход порождает сканер со сложностью $O(1)$ на литеру, но из-за различий между ними константы могут быть разными. В последнем подразделе мы рассмотрим два разных подхода к обработке зарезервированных слов.

2.5.1. Таблично управляемые сканеры

При таблично управляемом подходе используется заготовка сканера для управления и набор сгенерированных таблиц для кодирования знаний о конкретном языке. Как правило, таблично управляемые сканеры порождают генераторы сканеров. На этапе проектирования автор компилятора создает набор РВ. На этапе сборки генератор сканеров создает набор таблиц для реализации ДКА и прикомпилирует их к заготовке сканера.

Концептуально процесс выглядит следующим образом:



На рис. 2.12 показан таблично управляемый сканер для РВ $r[0...9]^+$ из раздела 2.3.2. На рисунке (a) мы видим код заготовки сканера, а на рисунках (b), (c) и (d) – таблицы, кодирующие ДКА, показанный на рисунке (e). Код более детальный, чем на рис. 2.2 выше, но по существу работает аналогично.

Заготовка сканера состоит из четырех секций: инициализация, цикл сканирования, моделирующий поведение ДКА, цикл отката для нахождения конечного состояния и последняя секция для интерпретации и вывода результатов. В цикле сканирования повторяются два основных действия сканера: читать литеру и выполнить переход. Цикл останавливается, когда ДКА попадает в состояние ошибки, s_e. Таблица переходов δ представляет диаграмму переходов ДКА. Одинаковые столбцы δ объединены, так что сканер пользуется таблицей

Записи в таблице для *0...9* объединены в один столбец.

Сжатие таблицы обсуждается в разделе 2.5.4.

CharClass для отображения входной литеры в индекс столбца δ. В цикле отката используется стек состояний для возврата сканера в последнее допускающее состояние.

```
state ← s0;
lexeme ← "";
очистить стек;
push(bad);

while (state ≠ se) do
    char ← NextChar();
    lexeme ← lexeme + char;
    if state ∈ SA then
        clear stack;
        push(bad);
    push(state);
    col ← CharClass[char];
    state ← δ[state,col];

while (state ∉ SA and state ≠ bad) do
    state ← pop();
    if state ≠ bad then
        обрезать лексему;
        RollBack();

if state ∈ SA
    then return Type[state];
    else return invalid;
```

(a) Код интерпретации таблиц

r	0...9	Прочие
Register	*Digit*	*Other*

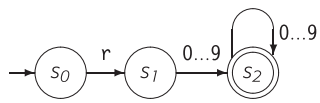
(b) Таблица классификации, *CharClass*

	Register	Digit	Other
s_0	s_1	s_e	s_e
s_1	s_e	s_2	s_e
s_2	s_e	s_2	s_e
s_e	s_e	s_e	s_e

(c) Таблица переходов δ

s_0	s_1	s_2	s_e
invalid	*invalid*	*register*	*invalid*

(d) Таблица типов токенов *Type*



(e) Исходный ДКА

Рис. 2.12 Таблично управляемый сканер для имен регистров

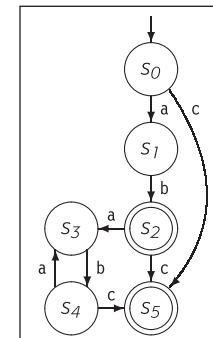
Функция *NextChar* возвращает следующую литеру из входного потока. Соответствующая ей функция *RollBack* возвращается на одну литеру входного потока (см. раздел 2.5.4).

Когда цикл сканирования останавливается, текущим является состояние ошибки s_e. Сканер возвращается назад, пока не найдет допускающее состояние или не докажет, что такого не существует. В большинстве языков величина такого отката ограничена. Можно построить РВ, требующее квадратичного отката на определенных входах. Если от сканера требуется обрабатывать такие РВ, то следует использовать альтернативную реализацию, например описанную в следующем подразделе. Однако в большинстве языков программирования величина отката невелика.

Как избежать слишком длинного отката

Некоторые РВ вынуждают сканер на рис. 2.12(а) совершать квадратичный откат. Проблема – в стремлении сканера вернуть самое длинное слово, являющееся префиксом входного потока.

Рассмотрим РВ *ab* | (*ab*)**c*. Соответствующий ДКА, показанный во врезке, распознает либо *ab*, либо любое число повторений *ab*, за которым следует *c*. На входной строке abababab c сканер, построенный на основе этого ДКА, прочитает все литеры и вернет одно слово – строку целиком. Но если на вход подана строка abababab, то сканеру придется просмотреть все литеры, прежде чем он установит, что самым длинным префиксом является ab. При следующем вызове он будет сканировать строку abab и вернет ab. При третьем вызове он сканирует строку abab и возвращает ab, а при последнем просто возвращает ab, не совершая откат. В худшем случае откат влечет за собой сложность $O(n^2)$.



ДКА для *ab*|(*ab*)**c*

Сканер с максимальным совпадением избегает такого патологического поведения, помечая тупиковые переходы в момент извлечения их из стека. Таким образом, со временем он накапливает пары ⟨*состояние*, *входная позиция*⟩, которые не ведут в допускающее состояние. В цикле сканирования проверяются все такие пары, и при попытке совершить тупиковый переход происходит выход из цикла.

```
state ← s0 ;
lexeme ← "";
очистить стек;
push(⟨bad,-1⟩);

while (state ≠ se) do
    if Failed[state,InputPos] then
         ⟨state, InputPos⟩ ← pop();
         обрезать лексему;
         break;
    char ← Input[InputPos];
    lexeme ← lexeme + char;
    if state ∈ SA then
        очистить стек;
        push(⟨bad,-1⟩);
    push(⟨state, InputPos⟩);
    col ← CharClass[char];
    state ← δ[state, col];
    InputPos ← InputPos + 1;

while (state ∉ SA and state ≠ bad) do
    if state ≠ se then
        Failed[state, InputPos] ← true;
    ⟨state, InputPos⟩ ← pop();
    if state ≠ bad then
    обрезать лексему;

if state ∈ SA
    then return TokenType[state];
    else return invalid;
```

Рис. 2.13 Сканер с максимальным совпадением

На рис. 2.13 показан сканер с максимальным совпадением. В глобальном счетчике `InputPos` хранится позиция во входном потоке. Для запоминания тупиковых переходов служит битовый массив `Failed`, в котором имеется строка для каждого состояния и столбец для каждой литеры из входного потока. Когда сканер вынужден откатиться, он помечает соответствующий переход в массиве `Failed`, чтобы не идти по тому же ведущему в тупик пути при следующих вызовах.

Код на рис. 2.13 выполняется при каждом вызове сканера. Перед первым вызовом необходимо инициализировать `Failed` и `InputPos`. Всем битам `Failed` присваивается значение `false`, а переменной `InputPos` – значение 1.

Небольшая оптимизация может уменьшить размер `Failed`. Например, если в сканере используется конечный входной буфер (см. раздел 2.5.4), то число столбцов в `Failed` можно уменьшить до размера этого буфера.

В большинстве языков программирования микросинтаксис настолько прост, что такой квадратичный откат вообще невозможен. Но если вы разрабатываете сканер для языка, в котором эта проблема актуальна, то описанная техника позволит избежать ее ценой небольших дополнительных затрат на каждый символ.

2.5.2. Непосредственно кодируемые сканеры

Чтобы повысить производительность таблично управляемого сканера, мы должны уменьшить стоимость одного или обоих главных действий: чтение символа и вычисление следующего перехода в ДКА. Непосредственно кодируемые сканеры уменьшают стоимость вычисления переходов ДКА путем замены явного представления состояний ДКА и функции переходов неявным. Неявное представление упрощает двухшаговое вычисление с табличным поиском. Оно исключает обращения к памяти, присущие такому вычислению, и допускает другие специализации. Функциональность получающегося сканера такая же, как у таблично управляемого сканера, но с меньшими накладными расходами на одну литеру. Генерировать непосредственно кодируемый сканер не труднее, чем эквивалентный таблично управляемый.

Таблично управляемый сканер тратит большую часть времени в главном цикле while; таким образом, весь смысл непосредственно управляемого сканера заключается в альтернативной реализации этого цикла. Отвлекаясь от деталей, этот цикл выполняет следующие действия:

```
while (state ≠ s_e) do
    char  ← NextChar();
    col   ← CharClass[char];
    state ← δ[state,col];
```

Здесь `state` явно представляет текущее состояние ДКА, а двумерный массив δ – его функцию переходов. Одинаковые столбцы δ объединяются (см. раздел 2.5.4). `CharClass` – вектор, отображающий входную литеру на индекс столбца δ.

Уменьшение затрат на табличный поиск

Для каждой литеры таблично управляемый сканер обращается к двум массивам: δ и `CharClass`. Хотя оба поиска занимают время **O**(1), с ними сопряжены постоянные затраты, которых в непосредственно кодируемом сканере можно избежать. Чтобы обратиться к *i*-му элементу `CharClass`, код должен вычислить его адрес, равный

```
@CharClass0 + i × w
```

где $@CharClass_0$ – константа, связанная с адресом *CharClass* в памяти, а *w* – размер элемента *CharClass* в байтах. Затем код загружает индекс столбца, находящийся по этому адресу.

> Детальное обсуждение кода вычисления адреса элемента массива см. в разделе 7.3.2.

Далее сканер ищет состояние в δ. Поскольку массив δ двумерный, вычисление адреса δ[state,col] сложнее:

```
@δ0 + (state × число столбцов δ + col) × w
```

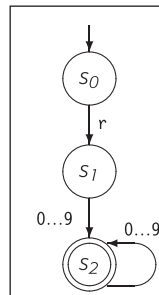
где $@\delta_0$ – константа, связанная с адресом начала δ в памяти, а `w` – размер одного элемента δ в байтах. И снова сканер должен выполнить операцию загрузки для извлечения данных, хранящихся по этому адресу.

Таким образом, таблично управляемый сканер вычисляет два адреса и выполняет две операции загрузки для каждой входной литеры. Некоторое повышение скорости работы непосредственно кодируемого сканера проистекает из уменьшения этих затрат.

Замена цикла while в таблично управляемом сканере

Таблично управляемый сканер представляет состояние ДКА и диаграмму переходов явно, чтобы можно было использовать один и тот же код для реализации каждого состояния. С другой стороны, непосредственно кодируемый сканер представляет состояние и диаграмму переходов неявно. Для реализации каждого состояния в нем используется особый фрагмент кода. А переходы из одного состояния в другое моделируются переходами на соответствующий фрагмент.

На рис. 2.14 показан непосредственно кодируемый сканер для регулярного выражения $r[0...9]^+$; ДКА показан во врезке, а таблично управляемый сканер – на рис. 2.12. Выполнение начинается с метки s_0, где сканер инициализируется и выполняются действия в состоянии ДКА s_0.



ДКА для $r[0...9]^+$

Рассмотрим код для состояния s_1. Он читает литеру, конкатенирует ее с текущей лексемой и помещает s_1 во внутренний стек. Если *char* – цифра, то сканер переходит в состояние s_2, в противном случае – в состояние s_{out}. В коде нет никаких сложных вычислений адресов. Он работает с минимальным набором значений – *char*, *lexeme* и *state*, – которые можно хранить в регистрах. Реализация остальных состояний столь же проста.

Генератор сканеров может непосредственно выдать код, похожий на тот, что показан на рис. 2.14. В каждом состоянии имеется два стандартных действия, а за ними – ветвление, реализующее переходы из этого состояния. Если из какого-то состояния исходит слишком много переходов, то хорошая реализация могла бы построить небольшую таблицу переходов и использовать схему «вычисляемого» ветвления – поиск в таблице меток. В отличие от таблично управляемого сканера, для каждого набора РВ код будет другим. Но поскольку этот код генерируется непосредственно по РВ, для автора компилятора разницы никакой нет.

```
s0 : очистить стек;
     push(bad);
     char ← NextChar();
     lexeme ← char;
     push(s0);
     if (char = 'r')
         then goto s1;
         else goto sout;

s1: char ← NextChar();
     lexeme ← lexeme + char;
     push(s1);
     if ('0'≤char≤'9')
         then goto s2;
         else goto sout;

s2 :  char ← NextChar();
      lexeme ← lexeme + char;
      очистить стек;
      push(s2);
      if '0'≤char≤ '9'
          then goto s2;
          else goto sout;

sout : state ← se;
       while (state ∉ SA and state ≠ bad) do
           state ← pop();
           if state ≠ bad then
               обрезать лексему;
               RollBack();
       end;

       if state ∈ SA
           then return Type[state];
           else return invalid;
```

Рис. 2.14 Непосредственно кодируемый сканер для $r[0...9]^+$

Код, написанный в стиле, показанном на рис. 2.14, часто называют *спагетти* за запутанность потока управления.

Конечно, этот сгенерированный код нарушает многие предписания структурного программирования. Небольшие примеры еще понятны, но для сложного набора РВ код труден для восприятия. Однако поскольку он генерируется, человеку ни читать, ни отлаживать его не придется. Повышенная скорость работы непосредственно кодируемого сканера делает его привлекательным, особенно если учесть, что от автора компилятора никакой дополнительной работы не требуется. Весь труд возлагается на реализацию генератора сканеров.

2.5.3. Сканеры, написанные вручную

Мы подозреваем, что рукописные сканеры продолжают существовать по одной простой причине: это небольшие, простые программы, которые интересно писать.

В сгенерированных сканерах, будь то таблично управляемый или непосредственно кодируемый, время обработки одной литеры невелико

и постоянно. Но, несмотря на это, во многих компиляторах используются сканеры, написанные вручную. В ходе неформального обзора групп пользователей коммерческих компиляторов мы обнаружили на удивление много компиляторов, в которых сканеры написаны вручную. И так же обстоит дело во многих популярных компиляторах с открытым исходным кодом. Например, генератор сканеров `flex` предположительно был создан для поддержки проекта GCC, но в нескольких фазах анализа GCC 4.0 используются рукописные сканеры.

Непосредственно кодируемый сканер уменьшает затраты на моделирование ДКА; рукописный сканер предлагает изобретательному автору компилятора дополнительные возможности повысить производительность. Код некоторых ветвей можно оптимизировать.

Например, сканер на рис. 2.14 реализует ДКА с единственным допускающим состоянием. Следовательно, от стека для отслеживания допускающих состояний можно отказаться. Переходы $s_0 \rightarrow s_{out}$ и $s_1 \rightarrow s_{out}$ можно заменить кодом, который просто возвращает *invalid*.

Аналогично можно улучшить интерфейс между сканером и парсером. Таблично управляемый и непосредственно кодируемый сканер для РВ $r[0...9]^+$ возвращают лексему в виде строки литер. Если в парсере имеется синтаксическая категория *имя регистра*, то сканер мог бы вернуть фактический номер регистра, а не содержащую его строку, – и тем самым избежать преобразования в парсере и нескольких конкатенаций в сканере.

2.5.4. Практические вопросы реализации

В этом разделе рассматриваются два практических вопроса, возникающих при построении сканера: обработка входного потока способом, обеспечивающим как эффективное политерное сканирование, так и откат, и сжатие таблицы переходов, чтобы она занимала меньше места.

Буферизация входного потока

Хотя политерный ввод-вывод приводит к чистым алгоритмам, накладные расходы на вызов функции для каждой литеры велики по сравнению со стоимостью моделирования ДКА. Чтобы уменьшить их, автор компилятора может воспользоваться буферизованным вводом-выводом, когда каждая операция чтения возвращает более длинную строку литер в буфере, а сканер обходит этот буфер. Сканер хранит указатель на позицию буфера. Ответственность за заполнение буфера, отслеживание текущей позиции и распознавание конца файла возлагаются на функцию `NextChar`. Эти операции можно встроить, часто они оформляются в виде макроса, чтобы избежать загромождения кода операциями разыменования указателя и инкремента.

Накладные расходы на чтение полного буфера складываются из двух частей: большие фиксированные затраты и небольшая стоимость для каждой литеры. Схема с буфером и указателем раскладывает фиксированные затра-

ты на много операций выборки литеры. Чем больше буфер, тем реже сканеру приходится нести эти затраты, так что накладные расходы в расчете на одну литеру уменьшаются.

Использование буфера и указателя также приводит к простой и эффективной реализации операции `RollBack`. Чтобы вернуться назад на одну литеру, сканер может просто уменьшить указатель буфера на единицу. Это работает, пока указатель не дойдет до начала буфера. А в этот момент сканеру все же понадобится доступ к предыдущему содержимому буфера.

```
NextChar( ) {
    Char ← Buffer[Input];
    if Char ≠ eof then
        Input ← (Input + 1) mod 2n;
        if (Input mod n = 0) then
            fill Buffer[Input : Input + n - 1];
            Fence ← (Input + n) mod 2n;
    return Char;
}
```

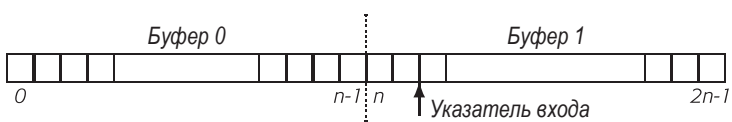
```
RollBack( ) {
    if (Input = Fence) then
        сигнализировать об ошибке отката;
    else
        Input ← (Input - 1) mod 2n;
}

Initialize() {
    Input ← 0;
    Fence ← 0;
    fill Buffer[0 : n-1];
}
```

Рис. 2.15 Реализация *NextChar* и *RollBack*

Двойная буферизация
Схему, в которой используются два входных буфера с инкрементом по модулю, чтобы реализовать ограниченный откат, часто называют *двойной буферизацией*.

На практике автор компилятора может ограничить расстояние, на которое сканеру разрешено откатиться. В этом случае сканер может просто использовать два соседних буфера и производить инкремент указателя по модулю, как показано ниже.



Чтобы прочитать литеру, сканер инкрементирует указатель по модулю *2n* и возвращает указатель в этой позиции. Чтобы откатиться на одну литеру назад, программа декрементирует входной указатель по модулю *2n*. Она также должна управлять содержимым буфера, читая дополнительные литеры из входного потока по мере необходимости.

Реализации `NextChar` и `RollBack` просты и эффективны, как показано на рис. 2.15. При каждом вызове `NextChar` загружается одна литера, инкрементируется указатель *Input* и проверяется, не пора ли заполнять буфер. Буфер заполняется после чтения каждых *n* литер. Код настолько мал, что его можно встроить, быть может, сгенерировать макросом. При такой схеме стоимость заполнения буфера раскладывается на *n* литер. Выбрав разумное значение

n, например 2048, 4096 или больше, автор компилятора может поддерживать накладные расходы на ввод-вывод на низком уровне.

`RollBack` обходится еще дешевле. Она проверяет, что содержимое буфера действительно, после чего декрементирует указатель. И снова реализация настолько проста, что допускает встраивание. На этапе инициализации только задается известное непротиворечивое начальное состояние.

δ	0	1	2	3	4	5	6	7	8	9	Другое
s_0	s_1	s_2	s_2	s_2	s_2	s_2	s_2	s_2	s_2	s_2	s_e
s_1	s_e	s_e	s_e	s_e	s_e	s_e	s_e	s_e	s_e	s_e	s_e
s_2	s_2	s_2	s_2	s_2	s_2	s_2	s_2	s_2	s_2	s_2	s_e
s_e	s_e	s_e	s_e	s_e	s_e	s_e	s_e	s_e	s_e	s_e	s_e

(a) Полная таблица переходов для 0 | [1…9] [0…9]*

δ	0	1…9	Другое
s_0	s_1	s_2	s_e
s_1	s_e	s_e	s_e
s_2	s_2	s_2	s_e
s_e	s_e	s_e	s_e

(b) Сжатая таблица

Рис. 2.16 Пример сжатия таблицы переходов

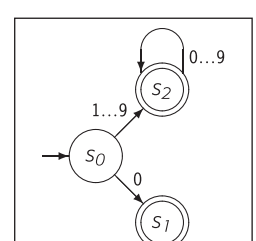
Из-за использования конечных буферов `RollBack` имеет ограниченную историю во входном потоке. Чтобы воспрепятствовать декременту указателя за начало этого контекста, `NextChar` и `RollBack` кооперируются. Указатель `Fence` всегда указывает на начало действительного контекста. `NextChar` устанавливает `Fence` всякий раз, как заполняет буфер. `RollBack` проверяет `Fence` всякий раз, как пытается декрементировать указатель `Input`.

После длинной серии операций `NextChar`, скажем больше *n*, где *n* – размер серии, `RollBack` всегда может откатиться по меньшей мере на *n* литер. Однако последовательность вызовов `NextChar` и `RollBack`, которые перемещаются вперед и назад по буферу, может привести к ситуации, когда расстояние между `Input` и `Fence` окажется меньше *n*. При больших значениях *n* вероятность такой ситуации уменьшается.

Если *n* выбрано с расчетом на повышение эффективности ввода-вывода, скажем 2048 или 4096 байт для каждой половины буфера, то величина отката в реальных программах должна быть достаточной. Величина отката, необходимая при конкретном входе, ограничена самой длинной последовательностью литер, не содержащей разделителей. Мало найдется программ, где идентификаторы насчитывают более 2048 литер.

Сжатие таблицы переходов

Таблица переходов для ДКА содержит |число состояний| · |Σ| элементов. Для реального языка программирования и |число состояний|, и |Σ| могут быть велики. Если размер таблицы оказывается больше размера кеша первого уровня, то могут возникнуть проблемы с производительностью.



ДКА для *0 | [1…9][0…9]**

Таблица переходов в сканере языка программирования часто содержит одинаковые столбцы. (Если ДКА минимален, то одинаковых строк быть не может.) Рас-

смотрим, к примеру, ДКА для РВ *0* | [*1*...*9*][*0*...*9*]* из раздела 2.2.2 (он повторен во врезке). На рис. 2.16(a) показано наивное представление таблицы, а на рис. 2.16(b) – та же таблица, в которой столбцы для литер от 1 до 9 сжаты в один. Заготовки кода в разделах 2.12 и 2.13 открывают еще одну возможность: на строку для состояния s_e не может быть ссылок, а значит, и представлять ее не нужно.

```
for i ← 1 to NumCols do
  MapTo[i] ← i
for i ← 1 to NumCols - 1 do
  if MapTo[i] = i then
    for j ← i + 1 to NumCols do
      if MapTo[j] = j then
        same ← true
        for k ← 1 to NumRows
          if δ(k,i) ≠ δ(k,j) then
            same ← false
            break
      if same then
        MapTo[j] ← i
```

Рис. 2.17 Нахождение одинаковых столбцов в таблице переходов δ

Чтобы использовать сжатую таблицу, сканер должен отобразить реально встречающиеся литеры на столбцы таблицы переходов. Простой и эффективный способ реализовать такое отображение дает таблица классификации, которая отображает входной символ на индекс столбца. Главный цикл сканера можно абстрагировать в виде кода, показанного слева на следующем рисунке:

```
while (state ≠ se) do
    char ← NextChar();
    state ← δ[state,char];
```

```
while (state ≠ se) do
    char ← NextChar();
    col ← CharClass[char];
    state ← δ[state,col];
```

Соответствующий код с классификацией показан справа. В нем имеется одно дополнительное обращение к памяти для доступа к `CharClass`, но взамен сканер может значительно сократить размер таблицы.

Разумеется, генератор сканеров должен порождать как классификатор, так и сжатую таблицу. На рис. 2.17 показан очевидный алгоритм нахождения одинаковых столбцов. Предполагается, что таблица переходов δ содержит *NumCols* столбцов и *NumRows* строк. По завершении алгоритма если *MapTo[i] = j* для $i \neq j$, то столбцы i и j одинаковы и могут быть сжаты в один. Оставляем построение таблицы классификации и сжатой версии δ в качестве упражнения для читателя (см. упражнение 2.13).

Приведенный алгоритм выполняет $O(|\Sigma|^2)$ сравнений в худшем случае. Каждое сравнение занимает время O(|states|). Мы можем уменьшить квад-

ратичный член, если будем запоминать для каждого столбца δ счетчик неошибочных состояний в этом столбце. Поскольку в одинаковых столбцах и счетчики должны быть одинаковы, алгоритм может построить множество индексов и поразрядно отсортировать его по этому счетчику. Тогда два столбца нужно будет сравнивать, только если после поразрядной сортировки они оказались в одной категории. Эта стратегия должна приводить к нескольким группам, каждая из которых требует $O(n^2)$ сравнений, но каждое n должно быть меньше $|\Sigma|$.

Счетчик неошибочных состояний – простая сигнатура столбца. Можно придумать более сложные и, возможно, более дорогие сигнатуры. В лучшем случае они могут уменьшить стоимость сравнений до $O(1)$. Однако приходится балансировать стоимость сравнения и стоимость вычисления сигнатуры – ответственный разработчик должен обращать на это внимание.

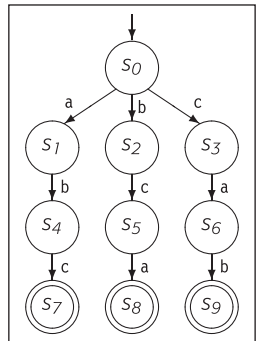
ИТОГИ РАЗДЕЛА

Автоматизированные методы позволяют без труда строить сканеры по минимальным ДКА. Генератор сканеров может выбрать либо таблично управляемый подход с общей заготовкой сканера и языково-зависимыми таблицами, либо непосредственно кодируемый сканер, который связывает с каждым состоянием ДКА явную ветвь. Вообще говоря, у непосредственно кодируемых сканеров накладные расходы на одну литеру меньше, чем у таблично управляемых.

Несмотря на то что у всех основанных на ДКА сканеров затраты на одну литеру постоянны и невелики, многие авторы компиляторов предпочитают писать сканер вручную. Этот подход позволяет экономно реализовать интерфейсы между сканером и системой ввода-вывода, с одной стороны, и парсером, с другой стороны.

ВОПРОСЫ

1. Для ДКА во врезке выполните следующие задания:
   - **a.** Постройте таблицу переходов на основе диаграммы переходов и вашего классификатора литер.
   - **b.** Напишите эквивалентный непосредственно кодируемый сканер.
2. Альтернативная реализация могла бы использовать ДКА для $(a|b|c)(a|b|c)(a|b|c)$ с последующим поиском в таблице, содержащей три слова: `abc`, `bca` и `cab`.
   - **a.** Нарисуйте эскиз ДКА для этого языка.
   - **b.** Сравните стоимость этого подхода со стоимостью использования ДКА из вопроса 1.



ДКА к вопросу 1

2.6. Дополнительные темы

В этом разделе приводятся дополнительные сведения по теме раздела 2.4. Комбинация построения Томпсона и построения подмножеств показывает, что ДКА способен выразить любое вычисление, которое можно выразить с помощью РВ. Чтобы замкнуть цикл и показать, что РВ может выразить лю-

бое вычисление, выразимое с помощью ДКА, нам необходимо построение, которое создает РВ, представляющее множество слов, допускаемых произвольным ДКА. В разделе 2.6.1 приведен набросок такого построения, которое часто называют алгоритмом Клини.

В разделе 2.6.2 рассматривается интересный частный случай задачи сведения РВ к ДКА: РВ, свободные от замыканий. ДКА для такого РВ ациклический. В разделе 2.6.2 приведен набросок алгоритма, который строит такой ДКА прямо и инкрементно.

Наконец, в разделе 2.4.4 представлен алгоритм Хопкрофта минимизации ДКА. В разделе 2.6.3 описан алгоритм Бжозовского, в котором используется алгоритм построения подмножеств.

2.6.1. Преобразование ДКА в регулярное выражение

Последний шаг в цикле построений, показанном на рис. 2.3, – построение РВ по ДКА. Комбинация построения Томпсона и алгоритма построения подмножеств дает конструктивное доказательство того, что ДКА по меньшей мере столь же мощны, сколь РВ. В этом разделе представлено построение Клини регулярного выражения, которое описывает множество строк, допускаемых произвольным ДКА. Этот алгоритм доказывает, что РВ по меньшей мере столь же мощны, сколь ДКА. А следовательно, РВ и ДКА эквивалентны.

Рассмотрим диаграмму переходов ДКА в виде графа с помеченными ребрами. Проблема вывода РВ, описывающего язык, допускаемый ДКА, соответствует проблеме перечисления путей в диаграмме переходов ДКА. Множество строк языка L(ДКА) состоит из множества реберных меток на каждом пути из d_0 в d_j $\forall$ $d_j \in D_A$. Для любого ДКА с циклическим графом переходов множество таких путей бесконечно. Однако для РВ определен оператор замыкания Клини, который решает эту проблему и позволяет кратко записать полное множество подпутей, порождаемое циклом.

На рис. 2.18 показан один из возможных алгоритмов вычисления этих путевых выражений. Предполагается, что состояния ДКА занумерованы числами от 0 до $|D| - 1$, а d_0 – начальное состояние. Для каждой пары вершин в диаграмме алгоритм генерирует выражение, представляющее метки вдоль всех путей между ними. На последнем шаге он комбинирует выражения для каждого пути, исходящего из d_0 и достигающего некоторого допускающего состояния $d_j \in D_A$. Путевые выражения строятся для всех путей путем перебора i, j и k.

Нотация R_{ij}^k обозначает РВ, описывающее все пути из d_i в d_j, которые не проходят через состояния с номерами, большими d_k. Но здесь «через» означает, что путь входит в состояние с номером, большим d_k, и выходит из него. В ДКА с переходом $d_1 \rightarrow d_{16}$ выражение $R_{1,16}^2$ было бы непустым, потому что путь входит в d_{16}, но не проходит через него.

$$
\begin{aligned}
&\textit{for } i = 0 \textit{ to } |D| - 1 \textit{ do}\\
&\quad \textit{for } j = 0 \textit{ to } |D| - 1 \textit{ do}\\
&\qquad R_{ij}^{-1} \leftarrow \{\, a \mid \delta(d_i, a) = d_j \,\}\\
&\qquad \textit{if } (i = j) \textit{ then}\\
&\qquad\quad R_{ij}^{-1} \leftarrow R_{ij}^{-1} \mid \{\epsilon\}\\
&\textit{for } k = 0 \textit{ to } |D| - 1 \textit{ do}\\
&\quad \textit{for } i = 0 \textit{ to } |D| - 1 \textit{ do}\\
&\qquad \textit{for } j = 0 \textit{ to } |D| - 1 \textit{ do}\\
&\qquad\quad R_{ij}^{k} \leftarrow R_{ik}^{k-1} (R_{kk}^{k-1}) * R_{kj}^{k-1} \mid R_{ij}^{k-1}\\
&L \leftarrow \mid_{d_j \in D_A} R_{0j}^{|D|-1}
\end{aligned}
$$

Рис. 2.18 Вывод регулярного выражения из ДКА

В традиционных формулировках этого алгоритма предполагается, что вершины пронумерованы от 1 до n, а не от 0 до $n - 1$. Поэтому прямые пути помещаются в R_{ij}^{0}.

Сначала алгоритм помещает все прямые пути из d_i в d_j в R_{ij}^{-1}. Он добавляет $\{\epsilon\}$ в каждое выражение, где $i = j$. На успешных итерациях он строит более длинные пути; R_{ij}^{k} вычисляется на основе R_{ij}^{k-1} путем добавления тех путей, которые проходят через d_k по дороге от d_i к d_j. Алгоритм вычисляет этот дополнительный компонент как (1) множество путей из d_i в d_k, которые не проходят ни через какое состояние с номером, большим *k-1*, конкатенированных со (2) всеми путями из d_k в себя, которые не проходят ни через какое состояние с номером, большим *k-1*, конкатенированными с (3) множеством путей из d_k в d_j, которые не проходят ни через какое состояние с номером, большим *k-1*. Присваивание во внутреннем цикле

$$R_{ij}^{k} \leftarrow R_{ik}^{k-1} (R_{kk}^{k-1}) * R_{kj}^{k-1} \mid R_{ij}^{k-1}$$

улавливает эти пути и с помощью оператора альтернации добавляет их в РВ, описывающее пути из R_{ij}^{k-1}. Следовательно, на каждой итерации внутреннего цикла добавляются пути, которые проходят через d_k в R_{ij}^{k-1}, и таким образом формируется R_{ij}^{k}.

Когда цикл по k завершается, различные выражения R_{ij}^{k} учитывают все пути в графе. На последнем шаге вычисляется множество путей, которые начинаются в d_0 и заканчиваются в каком-нибудь допускающем состоянии $d_j \in D_A$; это делается посредством применения оператора альтернации к путевым выражениям.

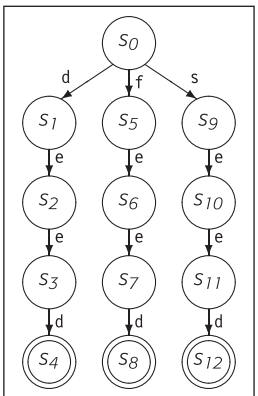
2.6.2. Регулярные выражения, свободные от замыканий

Один из подклассов регулярных языков, практические применения которого не исчерпываются сканированием, – множество языков, описываемых регулярными выражениями, свободными от замыканий. Такие РВ имеют вид $w_1 \mid w_2 \mid w_3 \mid \ldots w_n$, где отдельные слова w_i – просто конкатенации литер алфавита Σ. Такие РВ обладают тем свойством, что порождаемый ими ДКА ациклический.

Эти простые регулярные языки представляют интерес по двум причинам. Во-первых, многие задачи распознавания образцов можно описать свободным от замыканий РВ. Примерами могут служить слова в словаре, URL-адреса, подлежащие фильтрации, и ключи хеш-таблицы. Во-вторых, ДКА для свободного от замыканий РВ можно построить особенно эффективным способом.

Построение ДКА начинается с начального состояния s_0. Чтобы добавить слово в существующий ДКА, алгоритм следует по пути для нового слова, пока либо не исчерпает образец, либо не наткнется на переход в s_e. В первом случае он сопоставляет новому слову конечное состояние в качестве допускающего, а во втором добавляет путь для оставшегося суффикса нового слова. Результирующий ДКА можно закодировать в табличной или в непосредственной кодируемой форме (см. раздел 2.5.2). В любом случае распознавателю требуется постоянное время для обработки каждой литеры из входного потока.

В этом алгоритме стоимость добавления нового слова в существующий ДКА пропорциональна длине этого слова. Кроме того, алгоритм работает инкрементно; приложение может без труда добавлять новые слова в уже используемый ДКА. Это свойство делает ациклический ДКА интересной альтернативой реализации совершенной хеш-функции. При небольшом множестве ключей эта техника порождает эффективный распознаватель. По мере роста числа состояний (в непосредственно кодируемом распознавателе) или длины ключа (в таблично управляемом распознавателе) скорость работы реализации может падать из-за ограничений на размер кеша. В какой-то момент стоимость непопаданий в кеш делает эффективную реализацию более традиционной хеш-функции более привлекательным решением, чем инкрементное построение ациклического ДКА.



ДКА для *deed*, *feed* и *seed*

Минимальность так построенных ДКА не гарантируется. Рассмотрим ациклический ДКА, который был бы построен для РВ *deed*, *feed* и *seed* (показан во врезке). В нем есть три пути, каждый из которых распознает суффикс *eed*. Очевидно, что эти пути можно объединить, уменьшив тем самым число состояний и переходов в ДКА. Алгоритм будет строить ДКА, минимальный относительно префиксов слов языка, по аналогии с ДКА, порождаемыми алгоритмом построения подмножеств (см. раздел 2.4.3). Полная минимизация также объединила бы суффиксы с целью порождения меньшего ДКА.

2.6.3. Альтернатива алгоритму минимизации ДКА

Алгоритм построения подмножеств преобразовывал НКА в ДКА путем систематического исключения ϵ-переходов и объединения путей на диаграмме переходов НКА. Если применить его к НКА, в котором имеется несколько путей из начального состояния для некоторого префикса, то эти пути будут объединены в один. В результирующем ДКА нет дублирующихся префиксов. Воспользовавшись этим наблюдением, Бжозовский предложил альтернатив-

ный алгоритм минимизации, который непосредственно строит минимальный ДКА по НКА или ДКА.

Для НКА, созданных построением Томпсона, проблема достижимости не возникает. Но она может возникнуть при минимизации произвольного НКА.

Для НКА *n* обозначим *reverse(n)* НКА, полученный следующей процедурой: направления всех переходов изменяются на противоположные, начальное состояние делается конечным, добавляется новое начальное состояние, и с ним соединяются все состояния, бывшие конечными в *n*. Обозначим *reachable(n)* функцию, которая возвращает множество состояний и переходов в *n*, которые достижимы из его начального состояния. Наконец, обозначим *subset(n)* ДКА, порожденный применением алгоритма построения подмножеств к *n*.

Теперь минимальный ДКА, эквивалентный заданному НКА *n*, имеет вид

reachable(subset(reverse(reachable(subset(reverse(n)))))).

Внутреннее применение *subset* и *reverse* исключает дублирующиеся суффиксы в оригинальном НКА. Затем *reachable* отбрасывает все состояния и переходы, ставшие неинтересными. Наконец, внешнее применение *reachable*, *subset* и *reverse* устраняет дублирующиеся префиксы в НКА. (Применение *reverse* к ДКА может породить НКА.)

На рис. 2.19 показаны шаги, которые алгоритм выполняет, чтобы минимизировать ДКА для РВ $a(b|c)^*$, порожденный в двух предыдущих разделах. На рисунке (a) повторен ДКА, показанный на рис. 2.7. Применение *reverse* к этому ДКА порождает НКА, изображенный на рисунке (b). Затем алгоритм снова применяет *subset* к этому НКА и порождает ДКА на рисунке (c). Применение *reverse* к ДКА на рисунке (c) дает НКА на рисунке (d). И наконец, применение *subset* к этому НКА порождает окончательный минимальный ДКА, показанный на рисунке (e). Заметим, что в этом примере достижимость не играла никакой роли.

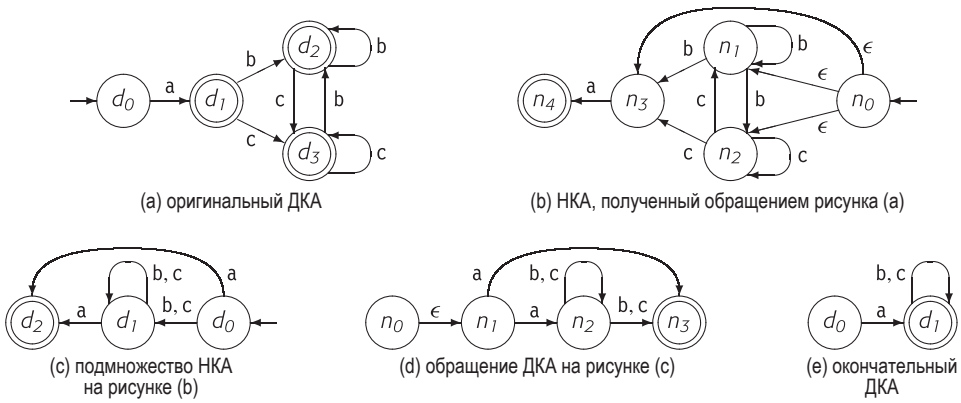


Рис. 2.19 Применение алгоритма Бжозовского к ДКА для $a(b|c)^*$

Алгоритм Бжозовского может обходиться дорого из-за того, что алгоритм построения подмножеств может создавать экспоненциально большое мно-

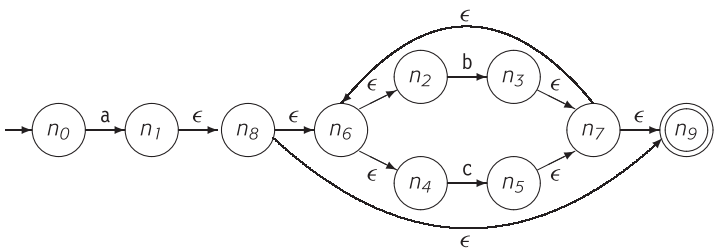
жество состояний. Асимптотическая сложность алгоритм Хопкрофта меньше: $O(|N| \, |\Sigma| \log_2(|N|))$, где N – множество состояний входного КА.

Выбрать компромисс между двумя алгоритмами не так-то просто. Однако изучение различных методов минимизации КА показывает, что время работы зависит от свойств конкретного КА. На практике алгоритм Бжозовского, похоже, работает достаточно хорошо.

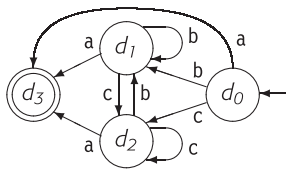
Первое применение функции *subset* обходится, по сути дела, бесплатно.

Кроме того, реализовать алгоритм Бжозовского почти наверняка будет проще, чем алгоритм Хопкрофта. Поскольку алгоритм Бжозовского порождает ДКА, его можно применить непосредственно к результату построения Томпсона, исключив дополнительное применение алгоритма построения подмножеств.

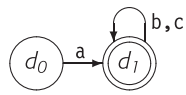
На рис. 2.20 показаны шаги, которые алгоритм выполняет, будучи применен непосредственно к НКА, созданному построением Томпсона для РВ *a*(*b*|*c*)*. На рисунке (a) представлен исходный НКА. На рисунке (b) показан ДКА, построенный применением *reverse*, а затем *subset* к этому НКА. На рисунке (c) видим окончательный ДКА. Построение обоих обращенных НКА оставляем в качестве упражнения для читателя.



(a) НКА для *a*(*b*|*c*)*



(b) подмножество обращения НКА на рисунке (a)



(c) подмножество обращения НКА на рисунке (b)

Рис. 2.20 Алгоритм Бжозовского применительно к НКА на рис. 2.5

Использование в генераторе сканеров

Поскольку в алгоритме Бжозовского используется построение подмножеств, первый его шаг объединяет все конечные состояния в одно репрезентативное состояние. Как описано в разделе 2.4.5, сканеру необходим ДКА, в котором каждое конечное состояние отображается на одну синтаксическую категорию. Алгоритм Хопкрофта можно модифицировать, так чтобы он хранил это отображение; для алгоритма Бжозовского такой модификации не существует.

Применение алгоритма Бжозовского к НКА порождает ДКА.

Чтобы использовать алгоритм Бжозовского в генераторе сканеров, инструмент должен был бы строить НКА для каждого правила и применять алгоритм Бжозовского к отдельным НКА. Затем он мог бы объединить эти ДКА ϵ-переходами и использовать алгоритм построения подмножеств для порождения окончательного ДКА. Как и раньше, если это применение алгоритма построения подмножеств объединяет конечные состояния, то необходимо назначить новое конечное состояние синтаксической категории правила с более высоким приоритетом. Получающийся ДКА не будет минимальным. Однако он будет меньше ДКА, порождаемого без минимизации.

2.7. Итоги и перспективы

Широкое распространение РВ для поиска и сканирования – одна из историй успеха современной информатики. Эти идеи были развиты на заре создания теории формальных языков и автоматов. Сегодня они повсеместно применяются в различных инструментах – от текстовых редакторов до программ фильтрации URL-адресов и компиляторов – как средство краткого описания групп строк, образующих регулярный язык. Всякий раз, как требуется распознавать конечное множество слов, следует обратить серьезное внимание на распознаватели на основе ДКА.

Теория РВ и конечных автоматов позволила разработать методы распознавания регулярных языков за время, пропорциональное длине входного потока. Методы автоматического вывода ДКА по РВ и минимизации ДКА открыли возможность построения надежных инструментов, генерирующих основанные на ДКА распознаватели. В современных авторитетных компиляторах используются как сгенерированные, так и написанные вручную сканеры. В любом случае хорошая реализация должна работать за время, пропорциональное длине входного потока, с небольшими затратами на одну литеру.

Замечания к этой главе

Первоначально причиной отделения лексического анализа, или сканирования, от синтаксического анализа, или парсинга, была эффективность. Поскольку стоимость сканирования линейно зависит от числа литер, а постоянные затраты невелики, вынесение лексического анализа из парсера в отдельный сканер снижало стоимость компиляции. Изобретение эффективных методов синтаксического анализа сделало этот аргумент менее убедительным, но практика построения сканеров сохранилась, потому что обеспечивает четкое разделение обязанностей между лексической и синтаксической структурами.

Поскольку построению сканера отведена небольшая роль в создании полного компилятора, мы старались сделать эту главу короткой. Поэтому мы опустили многие теоремы о регулярных языках и конечных автоматах, которые могли бы доставить удовольствие амбициозному читателю. Есть немало хоро-

ших книг по этому предмету, где конечные автоматы и РВ вместе со многими их полезными свойствами рассматриваются более глубоко [206, 241, 327].

В работе Клини [235] установлена эквивалентность РВ и ДКА. Замыкание Клини и алгоритм преобразования ДКА в РВ носят его имя. Макнафтон и Ямада продемонстрировали построение, связывающее РВ с НКА [270]. Построение, приведенное в этой главе, основано на работе Томпсона [345]; чтобы добавить поиск по регулярному выражению в текстовый редактор QED, он разработал небольшой компилятор, транслирующий РВ в машинный код для компьютера IBM 7094. Джонсон первым описал применение этих идей к автоматизации построения сканера [218]. Алгоритм построения подмножеств выводится из определения 11 в работе Рабина и Скотта [302].

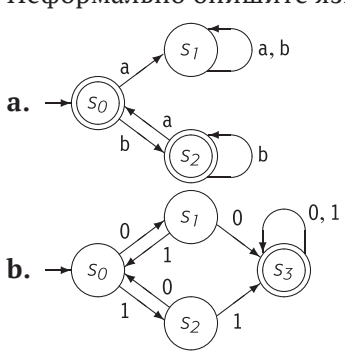
Хопкрофт опубликовал свой алгоритм минимизации ДКА, представленный в разделе 2.4.4, в 1971 году [205]. Он нашел применение при решении разнообразных задач, в т. ч. определения того, всегда ли две переменные программы принимают одно и то же значение [23]. Альтернативный алгоритм, описанный в разделе 2.6.3, был опубликован Бжозовским в 1962 году [66]. Несколько авторов сравнивали методы минимизации ДКА и их производительность [340, 355]. В нескольких работах рассматривается построение и минимизация ациклических ДКА [122, 356, 357].

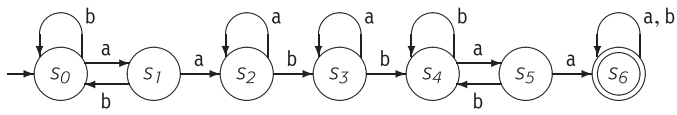
Сканер с максимальным совпадением описан в работе Репса [307]. Идея генерирования кода вместо таблиц с целью породить непосредственно кодируемый сканер, похоже, восходит к работам Уэйта [353] и Хуринга [201]. Они сообщают о пятикратном повышении производительности по сравнению с таблично управляемыми реализациями. В работе Нгассама и др. описываются эксперименты, характеризующие возможное ускорение в рукописных сканерах [283]. Несколько авторов исследовали компромиссы, присущие реализациям сканеров. Джонс [219] выступает в защиту непосредственного кодирования, но предлагает структурный подход к потоку управления вместо кода типа спагетти, показанного в разделе 2.5.2. В работе Брауэра и др. сравнивается скорость работы 12 различных реализаций сканера; выяснилось, что самая быстрая превосходит самую медленную в 70 раз [65].

Упражнения

К разделу 2.2

1. Неформально опишите языки, допускаемые следующими КА:



c. 

2. Постройте КА, допускающие каждый из следующих языков:
   a) {$w \in \{a, b\}^* \mid w$ начинается с «a» и содержит подстроку «$baba$»};
   b) {$w \in \{0, 1\}^* \mid w$ содержит подстроку «111» и не содержит подстроку «00»};
   c) {$w \in \{a, b, c\}^* \mid$ количество литер a по модулю 2 в слове w равно количеству литер b по модулю 3 в том же слове}.

3. Постройте КА для распознавания (a) строк, представляющих комплексные числа, и (b) строк, представляющих десятичные вещественные числа в научной нотации.

К разделу 2.3

4. В разных языках программирования используются разные нотации для представления целых чисел. Постройте регулярные выражения для каждого из следующих случаев:
   a) неотрицательные целые числа в языке C, представленные в системах счисления с основанием 10 и 16;
   b) неотрицательные целые числа в языке VHDL, которые могут включать знаки подчеркивания (но не на первом и не на последнем месте);
   c) денежные суммы в долларах, представленные положительным десятичным числом, округленным до двух знаков после точки. Такие числа начинаются знаком $, группы по три цифры слева от десятичной точки разделяются запятыми, а в конце, после десятичной точки, находятся две цифры. Примеры: $8,937.43 и $7,777,777.77.

5. Напишите регулярные выражения для каждого из следующих языков:
   a) дан алфавит Σ = {0, 1}, а L – множество всех строк, состоящих из чередующихся пар нулей и пар единиц;
   b) дан алфавит Σ = {0, 1}, а L – множество всех строк нулей и единиц, содержащих четное количество нулей или четное количество единиц;
   c) дан алфавит, состоящий из строчных английских букв, а L – множество всех строк, в которых буквы встречаются в порядке возрастания по алфавиту.

6. Напишите регулярные выражения для каждой из следующих языковых конструкций:
   a) любая последовательность табуляторов и пробелов (т. е. разделителей);
   b) комментарии на языке программирования C;
   c) строковые константы (без управляющих символов);
   d) числа с плавающей точкой.

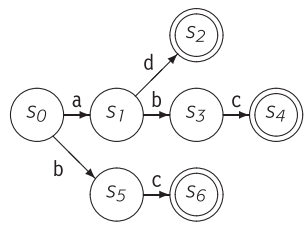
К разделу 2.4

7. Рассмотрим три регулярных выражения:

   $(ab \mid ac)^*$
   $(0 \mid 1)^*\ 1100\ 1^*$
   $(01 \mid 10 \mid 00)^*\ 11$

a) примените построение Томпсона к созданию НКА для каждого РВ;
b) преобразуйте НКА в ДКА;
c) минимизируйте ДКА.

8. Примените алгоритм минимизации Хопкрофта к показанному ниже ДКА.



9. Покажите, что множество регулярных языков замкнуто относительно пересечения.

К разделу 2.5

10. Постройте ДКА для каждой из следующих конструкций языка C, а затем постройте соответствующую таблицу для таблично управляемой реализации каждого из них:
a) целые константы;
b) идентификаторы;
c) комментарии.

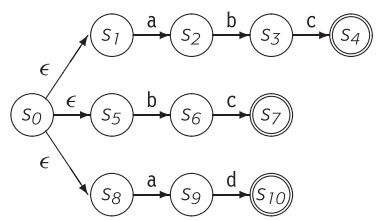
11. Для каждого ДКА из предыдущего упражнения напишите непосредственно управляемый сканер.

12. В этой главе описано несколько способов реализации ДКА. Еще одна альтернатива – использовать для реализации сканера взаимно рекурсивные функции. Обсудите преимущества и недостатки такой реализации.

13. На рис. 2.17 показан алгоритм, который обнаруживает одинаковые столбцы в функции переходов δ. Придумайте алгоритм, который конструирует классификатор литер, т. е. отображение литер на номера столбцов, а также сокращенную форму δ.
Можете предполагать, что столбец i функции δ соответствует i-й литере алфавита Σ, которая обозначается Σ_i.

К разделу 2.6

14. Примените алгоритм Бжозовского к следующему НКА:



Глава 3

Парсеры

АННОТАЦИЯ

Задача парсера – определить, является ли входная программа, представленная потоком классифицированных слов, порожденных сканером, правильным предложением языка программирования. Для этого парсер пытается построить вывод входной программы в грамматике, описывающей язык программирования.

В этой главе мы познакомимся с контекстно-свободными грамматиками для задания синтаксиса языка программирования. Будут разработаны нисходящие и восходящие методы. Описываются способы автоматизации построения парсеров обоих типов. Наконец, изучается ряд практических вопросов, возникающих при конструировании парсера.

КЛЮЧЕВЫЕ СЛОВА

Разбор, грамматика, LL(1), LR(1), рекурсивный спуск

3.1. Введение

Синтаксический разбор (парсинг) – вторая стадия фазы анализа компилятора. Парсер работает с программой, которая была разбита на токены сканером; он видит поток слов, аннотированных синтаксическими категориями (аналогами частей речи). Парсер выводит синтаксическую структуру программы, сопоставляя слова с грамматической моделью исходного языка программирования. Если парсер определяет, что входной поток является правильной программой, то он строит ее конкретную модель – промежуточное представление, которая будет использоваться остальными частями компилятора. Если же парсер обнаруживает ошибку, то сообщает о проблеме и ее месте пользователю.

Разбор и сканирование похожи. Как и сканирование, разбор был предметом всестороннего изучения; современные парсеры построены на солидном теоретическом фундаменте. Скорость имеет значение; все рассматриваемые нами методы занимают время, пропорциональное размеру программы

и ее представления. На производительность влияют низкоуровневые детали; в процессе реализации возникают те же компромиссы, что при сканировании. Описанные в этой главе методы могут быть реализованы как таблично управляемые, непосредственно кодируемые и рукописные парсеры. В отличие от сканеров, где написание кода вручную – обычное дело, парсеры гораздо чаще генерируются инструментально, чем пишутся вручную.

Концептуальная дорожная карта

Главная задача парсера – определить, является ли входная программа синтаксически правильным предложением исходного языка. Прежде чем строить парсеры, отвечающие на этот вопрос, нам нужен формальный механизм для определения синтаксиса исходного языка и систематический способ установления принадлежности этому формально описанному языку. Ограничившись множеством так называемых контекстно-свободных языков, мы можем разработать гарантированно эффективный алгоритм, отвечающий на вопрос о принадлежности. Нотацией для описания контекстно-свободных языков являются контекстно-свободные грамматики (КСГ).

Для ответа на вопрос о принадлежности в случае КСГ было предложено много алгоритмов. В этой главе рассматриваются два подхода к решению задачи: нисходящие и восходящие парсеры. Эти два вида парсеров существенно различаются по подходу и реализации. Но оба обрабатывают большой класс грамматик, включающий большинство конструкций, встречающихся в современных языках программирования. Не менее важно и то, что автору компилятора широкодоступны инструментальные средства, помогающие в создании нисходящего или восходящего парсера. В этой главе мы будем изучать как методы разбора, так и методы автоматизации конструирования парсеров.

Краткий обзор

Разбор
Для заданных потока слов *s* и грамматики *G* требуется найти в *G* вывод, порождающий *s*.

Основная обязанность парсера компилятора – распознавание синтаксиса, т. е. определение того, является ли компилируемая программа правильным предложением в синтаксической модели языка программирования. Эта модель выражается в виде формальной грамматики *G*; если некоторая строка слов *s* принадлежит языку, определенному *G*, то мы говорим, что *s выводится в G*. Для потока слов *s* и грамматики *G* парсер старается построить конструктивное доказательство того, что *s* может быть выведена в *G*, – этот процесс называется (синтаксическим) *разбором*.

Алгоритмы разбора можно отнести к двум общим категориям. Нисходящие парсеры пытаются сопоставить входной поток с продукциями грамматики и предсказать следующее слово (в каждой точке). Для ограниченного класса грамматик такое предсказание можно выполнить точно и эффектив-

но. В разделе 3.3 изучаются детали работы нисходящих парсеров и методы их создания. Рассматривается конструирование парсеров рекурсивного спуска и LL(1)-парсеров. Восходящие парсеры работают, начиная с низкоуровневых деталей – фактической последовательности слов, – и аккумулируют контекст, пока вывод не станет очевидным. И снова существует ограниченный класс грамматик, для которых можно сгенерировать эффективные восходящие парсеры. В разделе 3.4 изучается один конкретный вид восходящего парсера, таблично управляемый LR(1)-парсер, а также методы генерирования таких высокоэффективных парсеров. В последних разделах обсуждается ряд практических вопросов, возникающих при конструировании парсера.

Парсеры рекурсивного спуска
Так называются написанные вручную нисходящие парсеры. Они могут быть компактными и эффективными.

LL(1)-парсеры
LL(1)-парсеры являются таблично управляемыми и нисходящими. Они распознают класс грамматик, включающий наиболее интересные средства языков программирования.

LR(1)-парсеры
LR(1)-парсеры являются таблично управляемыми и восходящими. Они распознают более широкий класс грамматик, чем LL(1)-парсеры.

Несколько слов о времени

Проектирование, сборка и использование парсера охватывают весь процесс компиляции. На этапе проектирования автор компилятора выбирает модель разбора и комплект инструментов. Затем он создает КСГ для исходного языка во входном формате выбранного инструментария.

На этапе сборки с помощью инструментов разработки строится исполняемый парсер. Код рукописного парсера компилируется непосредственно. Для генерируемого парсера вызывается генератор, который строит парсер по КСГ и его аннотациям; затем этот код компилируется с целью создания исполняемого парсера.

Наконец, на этапе компиляции парсер анализирует разбитую на токены версию исходной программы. Он отображает поток классифицированных слов на КСГ и находит расхождения (если таковые имеются) между исходной программой и КСГ. Если входная программа правильна, то парсер генерирует промежуточное представление, которым будут пользоваться остальные части компилятора. Если вход содержит ошибки, то парсер сообщает о них программисту.

3.2. Выражение синтаксиса

Задача парсера – определить, отвечает ли некоторый поток слов синтаксису входного языка. В этом описании неявно подразумевается, что мы в состоянии описать синтаксис и проверить его; на практике необходима нотация для описания языков, которой люди могли бы пользоваться при програм-

мировании компьютеров. В главе 2 мы работали с одной такой нотацией, регулярными выражениями. РВ предлагают краткую нотацию для описания ограниченных видов синтаксиса. Описания на языке РВ позволяют создавать эффективные распознаватели. К сожалению, РВ недостаточно мощны для описания полного синтаксиса большинства языков программирования.

Для большинства языков программирования синтаксис выражается с помощью КСГ. В этом разделе мы познакомимся с КСГ и изучим их применение для проверки синтаксиса. Будет показано, как закодировать смысл в виде синтаксиса и структуры. Наконец, мы начнем знакомство с идеями, которые лежат в основе эффективных методов разбора, описанных в последующих разделах.

3.2.1. Почему бы не использовать регулярные выражения?

Чтобы обосновать применение КСГ, рассмотрим проблему распознавания алгебраических выражений, включающих имена и операторы +, -, × и ÷. Мы можем определить «имя» как любую строку, отвечающую РВ $[a\ldots z]([a\ldots z]|[0\ldots 9])^*$; это упрощенная версия идентификатора Algol, содержащая только строчные буквы. Теперь мы можем определить выражение следующим образом:

$$[a\ldots z]([a\ldots z]\,|\,[0\ldots 9])^*((+|-|\times|\div)[a\ldots z]([a\ldots z]\,|\,[0\ldots 9])^*)^*$$

Это РВ сопоставляется со строками «`a+b×c`» и «`e÷f×g`». Ничто в нем не предполагает какого-либо понятия предшествования операторов: какой оператор выполняется первым в выражении «`a+b×c`», + или ×? Стандартное правило алгебры говорит, что × и ÷ предшествуют + и -. Если мы хотим, чтобы порядок выполнения был иным, то должны расставить скобки.

Мы подчеркиваем (и), чтобы они визуально отличались от (и), которые используются для группировки в РВ.

Можем мы ли мы добавить скобки в РВ для выражений в местах, где они были бы допустимы? Выражение может начинаться скобкой (, поэтому РВ нуждается в факультативной начальной (. Аналогично необходима факультативная конечная).

$$(\underline{(}\,|\,\epsilon)[a\ldots z]([a\ldots z]\,|\,[0\ldots 9])^*$$
$$((+|-|\times|\div)[a\ldots z]([a\ldots z]\,|\,[0\ldots 9])^*)^*(\underline{)}\,|\,\epsilon)$$

Это РВ может породить выражение, заключенное в скобки, но не выражение с внутренними скобками, обозначающими предшествование. Все внутренние экземпляры (встречаются перед именем, тогда как все внутренние экземпляры) встречаются после имени, что предполагает следующее РВ:

$$\underline{(}^*[a\ldots z]([a\ldots z]\,|\,[0\ldots 9])^*\underline{)}^*$$
$$((+|-|\times|\div)\underline{(}^*[a\ldots z]([a\ldots z]\,|\,[0\ldots 9])^*\underline{)}^*)^*$$

Это РВ сопоставляется с «a+b×c» и «(a+b)×c». Оно сопоставляется с любым выражением, содержащим имена и четыре оператора с правильно расставленными скобками. К сожалению, оно сопоставляется также со многими синтаксически неправильными выражениями, такими как «a+(b×c» и «a+b)×c)». На самом деле мы вообще не можем написать РВ, которое сопоставлялось бы со всеми выражениями со сбалансированными скобками. (Парные конструкции, такие как `begin` и `end` или `then` и `else`, играют важную роль в большинстве языков программирования.)

Неспособность сопоставления с парными скобками, будь то ( ), { } или `begin` и `end`, – фундаментальное ограничение РВ; соответствующие распознаватели не могут посчитать скобки, поскольку имеют лишь конечное множество состояний. Язык $(^n)^n$ не является регулярным. ДКА в принципе не умеют считать. Для микросинтаксиса они работают хорошо, но для описания некоторых важных конструкций языков программирования не приспособлены.

> $a^n b^n$ – классический пример, который приводят при изложении леммы о разрастании в теории автоматов.

3.2.2. Контекстно-свободные грамматики

Для описания синтаксиса языка программирования нам необходима нотация, более мощная, чем регулярные выражения, но по-прежнему порождающая эффективные распознаватели. Традиционное решение – использовать КСГ. По счастью, большие подклассы КСГ обладают таким свойством.

> **Контекстно-свободная грамматика**
> Для языка *L* его КСГ определяет множество строк символов, допустимых в *L*.
>
> **Продукция**
> Каждое правило КСГ называется *продукцией*.

Форма Бэкуса–Наура

Традиционно для описания контекстно-свободных грамматик в информатике использовалась *форма Бэкуса–Наура*, или БНФ. В ней нетерминальные символы заключались в угловые скобки, например ⟨SheepNoise⟩. Терминальные символы подчеркивались, например baa. Символ ::= означал «выводится как» (или в русской традиции произношения «это»), а символ | – «также выводится как». Запись грамматики для звуков, издаваемых овцой, в БНФ выглядит так:

```
⟨SheepNoise⟩ ::= baa ⟨SheepNoise⟩
               | baa
```

БНФ датируется концом 1950-х – началом 1960-х годов [282]. Синтаксические соглашения об угловых скобках, подчеркивании, ::= и | проистекали из-за недостаточности типографических средств, доступных в то время авторам компиляторов. (См., например, книгу Дэвида Гриса «Compiler Construction for Digital Computers» [181].) В этой книге мы пользуемся типографически обновленной формой БНФ. Нетерминальные символы записываются *наклонным беззасечным шрифтом*. Терминальные символы записываются машинописным шрифтом. Если смену шрифта трудно различить, как в случае), то мы дополнительно подчеркиваем литеру,). Для обозначения «выводится как» мы используем символ →, а для обозначения «также выводится как» – символ |.

Предложение
строка символов, которую можно вывести из правил грамматики.

КСГ *G* – это набор правил, или *продукций*, описывающих, как образуются предложения. Множество предложений, выводимых из *G*, называется *языком, определенным G*, и обозначается *L*(*G*). Множество языков, определенных всеми возможными КСГ, называется множеством контекстно-свободных языков. Разобраться в этом поможет пример. Рассмотрим следующую грамматику, которую назовем *SN*:

```
1  SheepNoise  →  baa SheepNoise
2              |  baa
```

Нетерминальный символ
синтаксическая переменная, используемая в продукциях грамматики.

Терминальный символ
слово, которое может встречаться в предложении.

Слово состоит из лексемы и ее синтаксической категории. Слова представляются в грамматике своими синтаксическими категориями.

Первое правило, или продукция, читается как «из *SheepNoise* выводится слово `baa`, за которым следует еще одно *SheepNoise*». Здесь *SheepNoise* – синтаксическая переменная, представляющая множество строк, выводимых из грамматики. Такую синтаксическую переменную мы называем *нетерминальным символом*. Каждое слово языка, например `baa`, является *терминальным символом* грамматики. Второе правило читается как «из *SheepNoise* также выводится слово `baa`».

Чтобы понять связь между грамматикой *SN* и *L*(*SN*), нам необходимо определить, как применяются правила *SN* для вывода предложений *L*(*SN*). Прежде всего мы должны идентифицировать начальный символ *SN*. Он представляет множество всех строк *L*(*SN*). И следовательно, не может быть словом языка, а должен быть одним из нетерминальных символов, которые служат для включения в язык структуры и уровня абстракции. Поскольку в *SN* имеется только один нетерминальный символ, начальным символом является `SheepNoise`.

Контекстно-свободные грамматики

Формально контекстно-свободной грамматикой $\boldsymbol{G}$ называется четверка $(\boldsymbol{T}, \boldsymbol{NT}, \boldsymbol{S}, \boldsymbol{P})$, где:

- $\boldsymbol{T}$ множество терминальных символов, или слов, языка $\boldsymbol{L}(\boldsymbol{G})$. Терминальные символы соответствуют синтаксическим категориям, которые возвращает сканер;
- $\boldsymbol{NT}$ множество нетерминальных символов. Это синтаксические переменные, имеющие целью определить уровень абстракции и структуру продукций $\boldsymbol{G}$;
- $\boldsymbol{S}$ нетерминальный символ, отмеченный как *начальный символ* грамматики. $\boldsymbol{S}$ представляет множество всех предложений $\boldsymbol{L}(\boldsymbol{G})$.
- $\boldsymbol{P}$ множество продукций, или правил подстановки $\boldsymbol{G}$. Каждое правило в $\boldsymbol{P}$ имеет вид $\boldsymbol{NT} \rightarrow (\boldsymbol{T} \cup \boldsymbol{NT})^{+}$; т. е. оно заменяет один нетерминальный символ строкой, состоящей из одного или более символов грамматики.

Множества ***T*** и ***NT*** можно вывести прямо из продукций. Начальный символ может не вызывать сомнений, как в грамматике *SheepNoise*, но может быть и не очевиден, как в следующей грамматике:

Paren → (*Bracket*)
| ( )

Bracket → [*Paren*]
| [ ]

В этом случае выбор начального символа определяет вид внешних скобок. Если начальным является символ *Paren*, то внешними будут круглые скобки, а если *Bracket*, то квадратные скобки. Чтобы разрешить те и другие, автор компилятора может добавить новый нетерминальный символ S и две продукции $S \rightarrow Paren \mid Bracket$.

Некоторые инструменты работы с грамматиками требуют, чтобы S не встречался в правой части какой-либо продукции, это упрощает выявление S.

Вывод
последовательность шагов подстановки, которая начинается начальным символом грамматики и заканчивается предложением языка.

Для вывода предложения мы начинаем со строки-прототипа, содержащей только начальный символ. Затем повторяется следующий процесс: (1) выбрать нетерминальный символ α в строке-прототипе; (2) выбрать правило грамматики $\alpha \rightarrow \beta$; (3) подставить β вместо α. Когда строка-прототип будет содержать только терминальные символы, вывод останавливается – строка-прототип заменена предложением языка.

Сентенциальная форма
строка символов, встречающаяся в качестве одного шага правильного вывода.

В каждой точке вывода строка-прототип состоит из последовательности терминальных и нетерминальных символов. Когда такая строка встречается в качестве шага правильного вывода, она называется *сентенциальной формой*. Любую сентенциальную форму можно вывести из начального символа за нуль или более шагов. Аналогично из любой сентенциальной формы можно вывести правильное предложение за нуль или более шагов. Таким образом, если мы начнем с символа *SheepNoise* и будем применять последовательные подстановки, используя оба правила, то строка на каждом шаге процесса будет являться сентенциальной формой. Когда мы дойдем до точки, в которой строка содержит только терминальные символы, она будет являться предложением $L(SN)$.

Чтобы вывести предложение в SN, мы начинаем со строки, содержащей один символ, *SheepNoise*. Заменить *SheepNoise* можно по правилу 1 или по правилу 2. Если заменить по правилу 2, то получим строку `baa`, и дальнейших возможностей подстановки не будет. Эта подстановка показывает, что `baa` – правильное предложение $L(SN)$. Другой вариант, замена начальной строки по правилу 1, приводит к строке из двух символов: `baa` *SheepNoise*. В этой строке остался один нетерминальный символ; его замена по правилу 2 приводит к строке `baa baa`, являющейся предложением $L(SN)$. Эти два вывода можно записать в табличной форме:

Правило	Сентенциальная форма
	SheepNoise
2	`baa`

Замена по правилу 2

Правило	Сентенциальная форма
	SheepNoise
1	`baa` *SheepNoise*
2	`baa baa`

Замена по правилу 1, затем 2

Для удобства обозначений будем использовать символ $\rightarrow^+$, означающий «выводится за один или более шагов». Таким образом, *SheepNoise* $\rightarrow^+$ `baa` и *SheepNoise* $\rightarrow^+$ `baa baa`.

Правило 1 удлиняет сентенциальную форму, а правило 2 исключает нетерминальный символ *SheepNoise*. (Сентенциальная форма никогда не содержит более одного экземпляра *SheepNoise*.) Любой вывод в *SN* начинается нулем или более применений правила 1, а заканчивается применением правила 2. Применение правила 1 *k* раз с последующим применением правила 2 порождает строку, содержащую *k* + 1 повторений `baa`.

Заметим, что *L*(*SN*) можно описать РВ `(baa)+`. *L*(*SN*) – *регулярный язык*, т. е. элемент подмножества КСГ, которыми можно описать РВ (см. врезку «Классы контекстно-свободных грамматик» на стр. 154).

3.2.3. Более сложные примеры

Грамматика *SheepNoise* слишком проста для демонстрации всей мощи и сложности КСГ. Давайте вернемся к примеру, который использовали для иллюстрации недостатков РВ: языку выражений со скобками.

1	*Expr*	→	(*Expr*)
2		|	*Expr Op Exp*
3		|	`name`

4	*Op*	→	+
5		|	–
6		|	×
7		|	÷

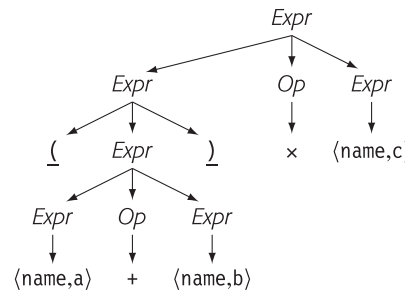
Дерево разбора
граф, представляющий вывод; также называется *синтаксическим деревом*.

Отправляясь от начального символа *Expr*, эта грамматика порождает два вида субтермов: субтермы в скобках по правилу 1 и простые субтермы по правилу 2. Чтобы сгенерировать предложение `(a + b) × c`, мы можем воспользоваться последовательностью подстановок (2, 3, 6, 1, 2, 3, 4, 3), как видим на рис. 3.1(a). А на рис. 3.1(b) показано *дерево разбора*, графически представляющее вывод.

Эта простая КСГ выражений не может сгенерировать предложение с несбалансированными или неправильно вложенными скобками. Только правило 1 может сгенерировать (. И то же правило генерирует парную). Поэтому оно не может сгенерировать такие строки, как `a + (b × c` или `a + b) × c`), и парсер, построенный по этой грамматике, не будет допускать такие строки. (Лучшее РВ в разделе 3.2.1 сопоставлялось с обеими этими строками.) Очевидно, что КСГ способны описывать конструкции, которые РВ описать не могут.

Правило	Сентенциальная форма
	Expr
2	*Expr Op Expr*
3	*Expr Op* name
6	*Expr* × name
1	(*Expr*) × name
2	(*Expr Op Expr*) × name
3	(*Expr Op* name) × name
4	(*Expr* + name) × name
3	(name + name) × name

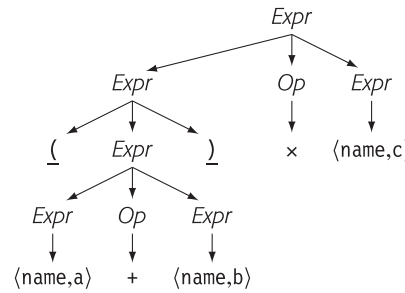
(a) Правосторонний вывод (a + b) × c;



(b) Соответствующее правостороннее дерево разбора

Правило	Сентенциальная форма
	Expr
2	*Expr Op Expr*
1	(*Expr*) *Op Expr*
2	(*Expr Op Expr*) *Op Expr*
3	(name *Op Expr*) *Op Expr*
4	(name + *Expr*) *Op Expr*
3	(name + name) *Op Expr*
6	(name + name) x *Expr*
3	(name + name) x name

(c) Левосторонний вывод (a + b) × c



(d) Соответствующее левостороннее дерево разбора

Рис. 3.1 Выводы (a + b) × c

В выводе (a + b) × c на рис. 3.1(a) на каждом шаге заменяется самый правый из оставшихся нетерминальных символов. Это систематическое поведение выбрано сознательно, но возможны и другие решения. Очевидная альтернатива – заменять на каждом шаге самый левый нетерминальный символ. При этом порождается другая последовательность вывода того же самого предложения. На рис. 3.1(c) и (d) показан левосторонний вывод.

Правосторонний вывод
вывод, при котором на каждом шаге заменяется самый правый нетерминальный символ.

Левосторонний вывод
вывод, при котором на каждом шаге заменяется самый левый нетерминальный символ.

В левостороннем и правостороннем выводах используется один и тот же набор правил, но применяются они в разном порядке. Поскольку синтаксическое дерево представляет только сами применяемые правила, а не порядок их применения, деревья разбора для обоих вариантов вывода одинаковы.

С точки зрения компилятора, важно, чтобы для каждого предложения языка, определенного КСГ, существовал единственный правосторонний (или ле-

Неоднозначность
Грамматика *G* называется *неоднозначной*, если для некоторого предложения *L*(*G*) имеется более одного правостороннего (или левостороннего) вывода.

По-другому, грамматика *G* называется *неоднозначной*, если для некоторого предложения *L*(*G*) имеется более одного дерева разбора.

восторонний) вывод. Если для некоторого вывода существует несколько правосторонних (или левосторонних) выводов, то в какой-то точке вывода несколько разных подстановок самого правого (или самого левого) нетерминального символа приводят к одному и тому же предложению. Грамматика, в которой для предложения существует несколько правосторонних (или левосторонних) выводов, называется *неоднозначной*. Неоднозначная грамматика может порождать несколько выводов и несколько деревьев разбора.

Для неоднозначной грамматики некоторые входы могут порождать несколько выводов и несколько смыслов. Программа должна иметь единственный смысл, в противном случае компилятор не будет знать, какой код генерировать. Следовательно, *неоднозначность – плохое свойство языка программирования*.

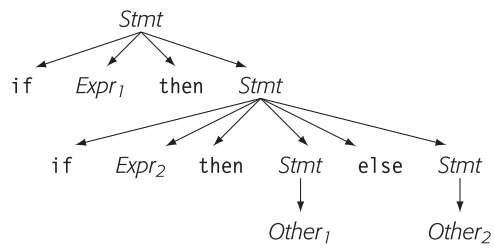
Классический пример неоднозначной грамматики языка программирования – конструкция `if-then-else` в Algol-подобных языках. Прямолинейная грамматика `if-then-else` могла бы выглядеть так:

```
1  Stmt  →  if Expr then Stmt
2           |  if Expr then Stmt else Stmt
3           |  Other
```

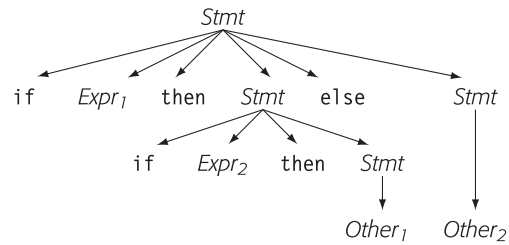
где *Other* представляет все предложения, которые могут встречаться в списке предложений, кроме `if-then` и `if-then-else`. Грамматика показывает, что фраза `else` факультативна. К сожалению, предложение

`if` $Expr_1$ `then if` $Expr_2$ `then` $Other_1$ `else` $Other_2$

имеет два разных правосторонних вывода. Разница между ними проста. В первом выводе $Other_2$ управляется внутренним `if`, поэтому $Other_2$ исполняется, когда $Expr_1$ истинно и $Expr_2$ ложно:



Во втором выводе фраза `else` ассоциируется с первым `if`, поэтому $Other_2$ исполняется, когда $Expr_1$ ложно независимо от значения $Expr_2$:



Эти два вывода порождают две совершенно разные семантики.

Чтобы устранить эту неоднозначность, мы должны модифицировать грамматику, включив в нее правило, которое определяет, какое `if` каким `else` управляет. Классическое решение – переписать грамматику следующим образом:

```
1  Stmt      →  if Expr then Stmt
2            |  if Expr then WithElse else Stmt
3            |  Other
4  WithElse  →  if Expr then WithElse else WithElse
5            |  Other
```

Это решение ограничивает множество предложений, которые могут встречаться в части `then` конструкции `if-then-else`. Оно допускает то же самое множество предложений, что исходная грамматика, но гарантирует, что каждое `else` однозначно сопоставляется с конкретным `if`. Оно включает в грамматику простое правило – связывать каждое `else` с самым внутренним ни с чем не ассоциированным `if`. Тогда пример имеет только один правосторонний вывод, которому соответствует первое дерево разбора.

Правило	Сентенциальная форма
	Stmt
1	`if` *Expr* `then` *Stmt*
2	`if` *Expr* `then if` *Expr* `then` *WithElse* `else` *Stmt*
3	`if` *Expr* `then if` *Expr* `then` *WithElse* `else` *Other*
5	`if` *Expr* `then if` *Expr* `then` *Other* `else` *Other*

В переписанной грамматике неоднозначность устранена.

Неоднозначность `if-then-else` возникает из-за недостатка оригинальной грамматики. Она была разрешена посредством добавления правила, которое программисту легко запомнить. В разделе 3.5.3 мы рассмотрим другие виды неоднозначности и систематические способы их обработки.

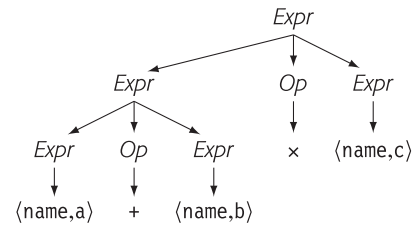
3.2.4. Кодирование семантики в структуре

Неоднозначность `if-then-else` указывает на связь между семантикой и грамматической структурой. Однако неоднозначность – не единственная ситуа-

ция, когда семантика и грамматическая структура взаимодействуют. Рассмотрим дерево разбора, которое было бы построено в случае правостороннего вывода простого выражения a + b × c.

Правило	Сентенциальная форма
	Expr
2	*Expr Op Expr*
3	*Expr Op* name
6	*Expr* × name
2	*Expr Op Expr* × name
3	*Expr Op* name × name
4	*Expr* + name × name
3	name + name × name

Вывод a + b × c



Соответствующее дерево разбора

Естественный способ вычисления выражения – обойти дерево в обратном порядке. При этом сначала было бы вычислено a + b, затем результат был бы умножен на c, и в итоге получилось бы (a + b) × c. Этот порядок вычислений противоречит классическим правилам алгебраического предшествования, согласно которым следует вычислить a + (b × c). Так как конечным итогом разбора выражения является порождение реализующего его кода, грамматика выражений должна обладать тем свойством, что при «естественном» порядке обхода дерева порождается правильный результат.

Истинная проблема – в структуре грамматики в начале раздела 3.2.3. В ней все арифметические операторы трактуются одинаково, приоритеты игнорируются. Правосторонний вывод a + b × c порождает не такое дерево разбора, как левосторонний.

В примере на рис. 3.1 показана строка без скобок. При наличии скобок левосторонний и правосторонний выводы дают одно и то же дерево разбора. Эта дополнительная продукция в грамматике добавила уровень в дерево разбора, что, в свою очередь, выливается в одинаковый порядок вычислений, не зависящий от порядка вывода.

0	*Goal*	→	*Expr*	5		|	*Term* ÷ *Factor*
1	*Expr*	→	*Expr* + *Term*	6		|	*Factor*
2		|	*Expr* – *Term*	7	*Factor*	→	(*Expr*)
3		|	*Term*	8		|	num
4	*Term*	→	*Term* × *Factor*	9		|	name

Рис. 3.2 Классическая грамматика выражений

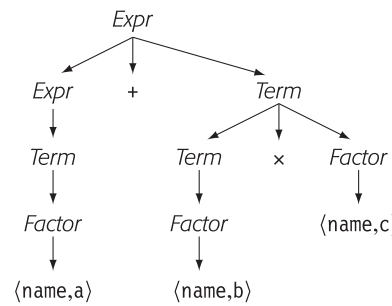
Этот эффект можно использовать для включения уровней приоритетов в грамматику. Сначала нужно решить, сколько уровней необходимо. В простой грамматике выражений необходимо три уровня приоритетов: высший для ( ), средний для × и ÷ и низший для + и –. Далее мы группируем операторы на разных уровнях и используем нетерминальный символ для изоляции

этой части грамматики. На рис. 3.2 показана результирующая грамматика; в нее добавлен начальный символ *Goal* и продукция для терминального символа `num`.

В классической грамматике выражений *Expr* образует уровень для + и -, *Term* – для × и ÷ и *Factor* для ( ). Модифицированная грамматика выводит дерево разбора для `a + b × c`, которое моделирует стандартное алгебраическое предшествование.

Правило	Сентенциальная форма
0	*Expr*
1	*Expr* + *Term*
4	*Expr* + *Term* × *Factor*
9	*Expr* + *Term* × `name`
6	*Expr* + *Factor* × `name`
9	*Expr* + `name` × `name`
3	*Term* + `name` × `name`
6	*Factor* + `name` × `name`
9	`name` + `name` × `name`

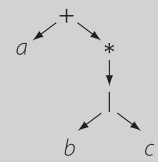
Вывод `a + b × c`



Соответствующее дерево разбора

Представление предшествования операторов

Построение Томпсона должно применить три своих преобразования в порядке, согласованном с предшествованием операторов в регулярном выражении. Чтобы представить этот порядок, реализация построения Томпсона может строить дерево, которое представляет регулярное выражение и его внутреннее предшествование. РВ $a(b|c)^*$ порождает следующее дерево:



где + представляет конкатенацию, | – альтернацию, а * – замыкание. Скобки закодированы в структуре дерева и потому не имеют явного представления.

Построение применяет преобразования в обратном порядке обхода дерева. Поскольку преобразования соответствуют операциям, при обратном порядке обхода строится следующая последовательность НКА: a, b, c, $b|c$, $(b|c)^*$ и, наконец, $a(b|c)^*$. В разделе 5.3 обсуждается механизм построения деревьев выражений

При обратном порядке обхода этого дерева разбора сначала вычисляется `b × c`, а затем к результату прибавляется `a`. Грамматика применяет стандартные правила арифметического предшествования. Эта грамматика однозначна; левосторонний вывод порождает то же самое дерево разбора.

Изменения влияют на длину вывода и размер деревьев разбора. Новые нетерминальные символы, поддерживающие предшествование, добавляют

в вывод шаги, а в дерево – внутренние вершины. В то же время перемещение операторов внутрь позволило исключить одну продукцию и одну вершину на каждый оператор.

Другие операции требуют более высокого приоритета. Например, операция индексирования массива должна применяться до стандартных арифметических операций. Тем самым гарантируется, например, что `a + b[i]` вычисляет значение `b[i]` до его прибавления к `a`, а не рассматривает `i` как индекс некоего массива, адрес которого равен `a + b`. Аналогично операции, изменяющие тип значения, называемые *приведениями типа*, в таких языках, как C или JAVA, имеют более высокий приоритет, чем арифметические операции, но более низкий, чем скобки или индексирование.

Если язык допускает присваивание внутри выражений, то оператор присваивания должен иметь низкий приоритет. Тогда код будет вычислять левую и правую части оператора присваивания до выполнения самого присваивания. Если бы, например, присваивание (←) имело такой же приоритет, как сложение, то вычисление `a ← b + c` слева направо присвоило бы `a` значение `b`, вместо того чтобы вычислить `b + c`, а затем присвоить результат `a`.

3.2.5. Нахождение вывода для заданной строки

Мы видели, как использовать КСГ *G* в качестве подстановочной системы для генерирования предложений языка *L*(*G*). Напротив, парсеру подается на вход строка, которая предположительно принадлежит *L*(*G*), а он должен найти для нее вывод. Процесс построения вывода заданного входного предложения называется (синтаксическим) *разбором*.

Парсер видит программу, которую постепенно отдает сканер: поток слов, аннотированных синтаксическими категориями. Таким образом, парсер увидел бы `a + b × c` как ⟨name,a⟩ + ⟨name,b⟩ × ⟨name,c⟩. На выходе парсер должен породить вывод входной программы или сообщение об ошибке, свидетельствующее о том, что программа неправильна.

Для однозначной грамматики дерево разбора и вывод эквивалентны.

Полезно визуализировать парсер как построение дерева разбора. Корень дерева известен; он представляет начальный символ грамматики. Листья дерева разбора тоже известны; они соответствуют потоку слов, возвращенных сканером. Трудная часть разбора – найти связь между листьями и корнем. Напрашиваются два противоположных подхода к построению дерева.

1. *Нисходящие парсеры* начинают с корня и строят дерево в направлении листьев. На каждом шаге парсер выбирает вершину, соответствующую какому-то нетерминальному символу на нижней кромке частично построенного дерева, и расширяет ее поддеревом, которое представляет правую часть продукции, подставляемой вместо этого символа.
2. *Восходящие парсеры* начинают с листьев и строят дерево по направлению к корню. На каждом шаге парсер находит подстроку на верхней

кромке частично построенного дерева, которая соответствует правой части некоторой продукции; затем он строит вершину, соответствующую левой части правила, и присоединяет ее к дереву.

В любом случае парсер принимает ряд решений о том, какие продукции применить. Интеллектуальная сложность разбора по большей части заключается в механизмах выбора таких решений. В разделе 3.3 рассматривается нисходящий разбор, а в разделе 3.4 – восходящий.

ИТОГИ РАЗДЕЛА

Проектировщики языков и авторы компиляторов используют контекстно-свободные грамматики для выражения синтаксиса языка программирования. Эти грамматики позволяют задавать структуру и форму. Они доказали свою способность выразить все признанные полезными особенности языков программирования.

В этом разделе были определены контекстно-свободные грамматики и показано, как их можно использовать для кодирования полезной структуры, например блоков и предшествования. Мы познакомились с понятием и классическим примером неоднозначности – грамматикой конструкции `if-then-else`. Наконец, были введены двойственные понятия вывода и разбора – предмет оставшейся части этой главы.

ВОПРОСЫ

1. Напишите КСГ для описания формы Бэкуса–Наура.
2. Рассмотрим классическую грамматику выражений. Какие свойства этой грамматики и входной программы определяют длины вывода?

3.3. Нисходящий разбор

Нисходящий парсер начинает работу с корня дерева разбора и систематически расширяет его вниз, пока листья не совпадут со словами, возвращаемыми сканером. В каждой точке процесс рассматривает частично построенное дерево разбора. Он выбирает нетерминальный символ на нижней кромке дерева и расширяет нижнюю кромку сыновними вершинами, соответствующими правой части некоторой продукции для этого символа. Расширить кромку за терминальный символ парсер не может. Этот процесс продолжается до тех пор, пока:

a) кромка дерева разбора не будет содержать только терминальные символы, и при этом входной поток исчерпан;
b) не встретится очевидное несоответствие между кромкой частично построенного дерева разбора и входным потоком.

В первом случае разбор считается успешным. Во втором случае возможны две ситуации. Парсер мог выбрать неверную продукцию на каком-то более раннем шаге процесса, и тогда он может вернуться назад, систематически пересматривая принятые ранее решения. Если входная строка является до-

пустимым предложением языка, то механизм возврата рано или поздно позволит парсеру отыскать правильную последовательность продукций и построить дерево разбора. Если же входная строка не является допустимым предложением, то возврат ничего не даст, и парсер сообщит пользователю об ошибке.

Одно ключевое свойство обеспечивает эффективность нисходящего разбора: *большое подмножество контекстно-свободных грамматик можно разобрать без возврата*. В разделе 3.3.1 показаны преобразования, которые часто превращают произвольную грамматику в пригодную для нисходящего разбора без возвратов. В последующих разделах описываются два вида нисходящих парсеров: рукописные парсеры рекурсивного спуска и сгенерированные LL(1)-парсеры.

Нисходящий парсер строит левосторонний вывод, соответствующий сканированию входа слева направо.

Чтобы сделать обсуждение более конкретным, на рис. 3.3 приведен высокоуровневый алгоритм нисходящего парсера, который строит левосторонний вывод. Он строит дерево разбора с корнем в переменной *root*. Для отслеживания еще не сопоставленной нижней кромки частично построенного дерева служит стек.

```
root ← вершина начального символа S
focus ← root
поместить null в стек

word ← NextWord( )

while (true) do
   if (focus является нетерминальным символом) then
      выбрать следующее правило для расширения focus, скажем A → β1 β2 β3 ... βn
      построить вершины для β1 β2 β3 ... βn в виде дочерних вершин focus
      поместить βn, βn-1, βn-2, ... β2 в стек
      focus ← β1
   else if (word совпадает с focus) then
      word ← NextWord( )
      focus ← извлечь из стека
   else if (word = eof and focus = null) then
      допустить вход и вернуть root
   else возврат
```

Рис. 3.3 Алгоритм левостороннего нисходящего разбора

Тело парсера состоит из цикла while, в котором анализируется самый левый несопоставленный символ на нижней кромке частично построенного дерева разбора – тот, что хранится в *focus*. Если в *focus* находится нетерминальный символ, то парсер расширяет дерево вниз; он выбирает продукцию, строит дерево для ее правой части и присваивает *focus* самый левый символ из этого дерева. Если *focus* – терминальный символ, то он сравнивается со следующим входным словом. При совпадении парсер сдвигает *focus* на следующий символ кромки и получает следующее слово.

Если *focus* – терминальный символ, не совпадающий со входом, то парсер должен вернуться назад. Парсер переставляет *focus* на его родителя в частично построенном дереве разбора, отсоединяет сыновние вершины и отбрасывает их. Если *focus* указывает на левую часть еще не испробованного правила, то парсер расширяет *focus* по этому правилу. Он строит сыновние вершины для каждого символа в правой части, помещает эти символы в стек в порядке справа налево и устанавливает *focus* на первую сыновнюю вершину. Если неиспробованных правил не осталось, то парсер поднимается еще на один уровень вверх и повторяет попытку. По исчерпании всех возможностей он сообщает о синтаксической ошибке и останавливается. Возврат увеличивает асимптотическую стоимость разбора; на практике это дорогой способ обнаружения синтаксических ошибок.

Чтобы упростить нахождение «следующего» правила, парсер может сохранить номер правила в вершине нетерминального символа, когда будет расширять ее.

В процессе возврата парсер должен также отматывать назад входной поток. По счастью, в частичном дереве разбора закодировано достаточно информации, чтобы выполнить это действие эффективно. Парсер должен поместить каждый сопоставленный терминальный символ отброшенной продукции назад во входной поток – это он может сделать, когда отсоединяет символы от дерева разбора во время обхода отброшенных сыновних вершин.

3.3.1. Преобразование грамматики

Эффективность нисходящего парсера сильно зависит от его способности выбирать правильную продукцию при каждом расширении нетерминального символа. Если парсер всегда делает правильный выбор, то разбор будет эффективен. Если же он допускает ошибки при выборе, то стоимость разбора возрастает. В этом разделе рассматриваются структурные проблемы, которые могут сделать КСГ непригодной для нисходящего разбора. Описываются преобразования, которые автор компилятора может применить к грамматике, чтобы избежать этих проблем.

Нисходящий парсер с оракулом

В качестве первого упражнения рассмотрим поведение парсера на рис. 3.3 для классической грамматики выражений на рис. 3.2 применительно к строке `a + b × c`. Предположим ненадолго, что у парсера имеется оракул, который в каждый момент выбирает правильную продукцию. Тогда разбор мог бы происходить, как показано на рис. 3.4. В правом столбце показана входная строка, а маркер ↑ обозначает текущую позицию разбора в строке. Символ → в столбце правила представляет шаг, на котором парсер сопоставляет терминальный символ с входной строкой и продвигает текущую позицию. На каждом шаге сентенциальная форма представляет нижнюю кромку частично построенного дерева разбора. Для строки `a + b × c` парсер применил восемь правил и сопоставил пять слов.

Опережающий символ
слово, с которым парсер пытается произвести сопоставление.

На рис. 3.4 опережающим символом является слово, которое следует непосредственно за ↑.

Заметим, однако, что выбор оракула был непоследовательным. На первых двух *шагах focus* содержала нетерминальный символ *Expr*. На первом шаге парсер применил правило 1, *Expr* → *Expr* + *Term*. На втором шаге в *focus* по-прежнему символ *Expr*, а опережающим символом выступает `name`. На этот раз парсер применил правило 3, *Expr* → *Term*. Столь же непоследователен оракул был при сопоставлении имен – он использовал последовательность (6, 9) для сопоставления с `a` из *Term*, а затем последовательность (4, 6, 9) для сопоставления с `b` из *Term*. Было бы трудно заставить нисходящий парсер хорошо работать с последовательным алгоритмическим выбором для этой версии грамматики выражений.

Правило	Сентенциальная форма	Вход
	Expr	↑ `name + name × name`
1	*Expr* + *Term*	↑ `name + name × name`
3	*Term* + *Term*	↑ `name + name × name`
6	*Factor* + *Term*	↑ `name + name × name`
9	`name` + *Term*	↑ `name + name × name`
→	`name` + *Term*	`name` ↑ `+ name × name`
→	`name` + *Term*	`name +` ↑ `name × name`
4	`name` + *Term* × *Factor*	`name +` ↑ `name × name`
6	`name` + *Factor* × *Factor*	`name +` ↑ `name × name`
9	`name + name` × *Factor*	`name +` ↑ `name × name`
→	`name + name` × *Factor*	`name + name` ↑ `× name`
→	`name + name` × *Factor*	`name + name ×` ↑ `name`
9	`name + name × name`	`name + name ×` ↑ `name`
→	`name + name × name`	`name + name × name` ↑

Рис. 3.4 Левосторонний нисходящий разбор `a + b × c` с оракулом

Устранение левой рекурсии

Левая рекурсия
КСГ называется *леворекурсивной*, если для некоторого $A \in NT$ и $\beta \in (T \cup NT)^*$ имеет место $A \rightarrow^+ A\beta$. То есть за один или несколько шагов грамматика позволяет вывести из A сентенциальную форму, начинающуюся с A.

Прямая левая рекурсия встречается на одном шаге вывода. Косвенная левая рекурсия занимает несколько шагов.

В классической грамматике выражений возникает несколько проблем при использовании левостороннего нисходящего парсера. Быть может, самая трудная связана с тем, что в ней используется *левая рекурсия*. То есть имеются продукции, например *Expr* → *Expr* + *Term*, в которых правая часть начинается нетерминальным символом, находящимся в левой части.

Рассмотрим, как нисходящий парсер повел бы себя на входной строке `a + b × c`, если бы он всегда применял продукции в порядке их появления в грамматике. Первыми его действиями были бы:

Правило	Сентенциальная форма	Вход
	Expr	↑ name + name × name
1	*Expr* + *Term*	↑ name + name × name
1	*Expr* + *Term* + *Term*	↑ name + name × name
1	. . .	↑ name + name × name

Парсер начинает с символа *Expr* и пытается сопоставить с ним a. В результате применения правила 1 на кромке создается сентенциальная форма *Expr* + *Term*. Далее он снова сталкивается с нетерминальным символом *Expr* и входным словом a. Оставаясь последовательным, он применяет правило 1, чтобы заменить *Expr* на *Expr* + *Term*. И конечно, снова видит *Expr* и входное слово a. При такой грамматике стремящийся вести себя последовательно парсер продолжит расширять кромку бесконечно. Проблема в том, что расширение путем подстановки *Expr* + *Term* никогда не порождает терминальный символ в начале.

Эта проблема возникает вследствие использования левой рекурсии в продукциях 1 и 2. Та же проблема имеет место в продукциях 4 и 5. В случае леворекурсивной грамматики нисходящий парсер может расширять кромку бесконечно, не генерируя начального терминального символа, который можно было бы сопоставить или отвергнуть. Чтобы исправить проблему, автор компилятора может преобразовать леворекурсивную грамматику в такую, где используется только *правая рекурсия*.

Правая рекурсия
КСГ называется *праворекурсивной*, если для некоторых $A \in NT$ и $\beta \in (T \cup NT)^*$ имеет место $A \rightarrow^+ \beta A$.

Преобразование левой рекурсии в правую производится механически. Пример ниже иллюстрирует такое преобразование в случае прямой левой рекурсии. В левой части показана простая леворекурсивная грамматика, а в правой части – она же после преобразования: новая грамматика определяет тот же язык, что и оригинальная.

$$\begin{array}{lll} Fee & \rightarrow & Fee\,\alpha \\ & | & \beta \end{array} \qquad \begin{array}{lll} Fee & \rightarrow & \beta\,Fee' \\ Fee' & \rightarrow & \alpha\,Fee' \\ & | & \epsilon \end{array}$$

Преобразование вводит новый нетерминальный символ, *Fee'*. Первая продукция переписывается с целью создать левую рекурсию по *Fee'*, а вторая – чтобы добавить *Fee'* после β. Используется также продукция $Fee' \rightarrow \epsilon$, где ϵ представляет пустую строку, чтобы положить конец рекурсии. Такая продукция называется ϵ-продукцией.

В классической грамматике выражений прямая левая рекурсия встречается в продукциях для *Expr* и *Term*.

Оригинальная грамматика	Преобразованная грамматика
Expr → *Expr* + *Term*	*Expr* → *Term* *Expr'*
| *Expr* – *Term*	*Expr'* → + *Term* *Expr'*
| *Term*	| – *Term* *Expr'*
	| ϵ
Term → *Term* × *Factor*	*Term* → *Factor* *Term'*
| *Term* ÷ *Factor*	| × *Factor* *Term'*
| *Factor*	| ÷ *Factor* *Term'*
	| ϵ

Заметим, что мы не можем просто переписать *Expr* → *Expr* + *Term* в виде *Expr* → *Term* + *Expr*, поскольку это изменило бы ассоциативность сложения (см. раздел 5.7.1).

Подстановка преобразованных последовательностей обратно в оригинальную грамматику дает праворекурсивную грамматику, которая описывает тот же самый язык. Эта праворекурсивная грамматика показана на рис. 3.5.

0	*Goal*	→	*Expr*	6	*Term'*	→	× *Factor Term'*
1	*Expr*	→	*Term Expr'*	7		|	÷ *Factor Term'*
2	*Expr'*	→	+ *Term Expr'*	8	*Factor*	→	ϵ
3		|	– *Term Expr'*	9		|	(*Expr*)
4		|	ϵ	10		|	num
5	*Term*	→	*Factor Term'*	11			name

Рис. 3.5 Праворекурсивный вариант классической грамматики выражений

Преобразование устраняет прямую левую рекурсию. Но левая рекурсия может также быть косвенной, например $\alpha \rightarrow^{+} \alpha\delta$ по цепочке правил, таких как $\alpha \rightarrow \beta$, $\beta \rightarrow \gamma$ и $\gamma \rightarrow \alpha\delta$. Косвенная левая рекурсия не всегда очевидна, ее может маскировать длинная цепочка продукций.

Чтобы устранить все виды левой рекурсии, нам необходим систематический подход. Алгоритм на рис. 3.6 устраняет любую левую рекурсию в грамматике, применяя два метода: прямая подстановка для преобразования косвенной левой рекурсии в прямую и преобразование левой рекурсии в правую, описанное выше. Предполагается, что в оригинальной грамматике нет циклов ($A \rightarrow^{+} A$) и ϵ-продукций.

ввести порядок на множестве нетерминальных символов, A_0, A_1, ..., A_i
for $i \leftarrow 0$ *to* n *do*
  for $s \leftarrow 0$ *to* $i-1$ *do*
    заменить каждую продукцию $A_i \rightarrow A_s\gamma$ *на* $A_i \rightarrow \delta_1\gamma \mid \delta_2\gamma \mid \ldots \mid \delta_k\gamma$,
      где $A_i \rightarrow \delta_1 \mid \delta_2 \mid \ldots \mid \delta_k$ *– текущие продукции для* A_i
  устранить все прямые левые рекурсии по A_i *с помощью преобразования*

Рис. 3.6 Алгоритм устранения косвенной левой рекурсии

Алгоритм устанавливает произвольный порядок на множестве нетерминальных символов грамматики. Во внешнем цикле нетерминальные символы перебираются в этом порядке. Во внутреннем цикле ищутся продукции, которые расширяют A_i в правую часть, начинающуюся с A_s, где $s < i$. Такое расширение может привести к косвенной левой рекурсии. Чтобы избежать этого, алгоритм заменяет вхождение A_s всеми альтернативными правыми частями продукций A_s. В конечном итоге этот процесс преобразует каждую косвенную левую рекурсию в прямую. На последнем шаге внешнего цикла каждая прямая левая рекурсия по A_i преобразуется в правую рекурсию с помощью описанного выше преобразования. Поскольку новые нетерминальные символы

добавляются в конец и подразумевают только правую рекурсию, в цикле их можно игнорировать – их не нужно ни проверять, ни преобразовывать.

Алгоритм станет понятнее, если рассмотреть инвариант внешнего цикла. В начале i-й итерации

$\forall\ k < i$ в правой части ни одной продукции, расширяющей A_k, не встречается A_s для $s < k$.

В конце этого процесса ($i = n$) каждая косвенная левая рекурсия была устранена прямой подстановкой, а каждая прямая левая рекурсия устранена преобразованием в правую рекурсию.

Пример

Рассмотрим простую леворекурсивную грамматику во врезке. Индексы нетерминальных символов обозначают порядок их перебора алгоритмом. Алгоритм работает следующим образом:

$i = 0$ Алгоритм не входит во внутренний цикл. Правило вида $Goal_0 \rightarrow Goal_1\gamma$ найдено.

$i = 1$ Алгоритм ищет правило вида $A_1 \rightarrow Goal_0\gamma$.
$s = 0$ Такого правила не существует.

$i = 2$ Алгоритм ищет правило вида $B_2 \rightarrow Goal_0\gamma$.
$s = 0$ Такого правила не существует.

$i = 2$ Алгоритм ищет правило вида $B_2 \rightarrow A_1\gamma$.
$s = 1$ Он находит правило 3 и переписывает его в виде $B_2 \rightarrow B_2$ a b | a b.

Затем он применяет преобразование прямой левой рекурсии к этим новым правилам. Этот процесс порождает правила 3, 4 и 5 в конечной грамматике. Вводится новый символ C_3.

0	$Goal_0$	→	A_1
1	A_1	→	B_2 a
2		|	a
3	B_2	→	A_1 b

Оригинальная грамматика

0	$Goal_0$	→	A_1
1	A_1	→	B_2 a
2		|	a
3	B_2	→	a b C_3
4	C_3	→	a b C_3
5		|	ϵ

Преобразованная грамматика

Разбор с применением ϵ-продукций

В методе устранения левой рекурсии были введены ϵ-продукции. Нисходящий парсер на рис. 3.3 не понимает ϵ-продукций. К счастью, чтобы решить эту проблему, необходимо лишь мелкое изменение. Когда парсер выбирает продукцию, в правой части которой находится ϵ, он должен просто выполнить присваивание *focus* ← *pop()*. Это простое действие переносит его внимание на следующую вершину, терминальную или нетерминальную, на кромке.

Разбор, свободный от возврата

Основной источник неэффективности нисходящего парсера на рис. 3.3 – потенциальная возможность возврата. В приведенном виде он не имеет никакого механизма выбора правильной продукции для расширения нижней

кромки частично построенного дерева разбора. (Алгоритм просто говорит «*выбрать следующее правило...*».) Если выбор удачный, то разбор будет эффективным. В противном случае обнаружится несоответствие между кромкой и входом, в результате чего придется уничтожать и заново расширять кромку, пока не будет найдено подходящее правило или – после полного перебора – не выяснится, что такого правила не существует.

Чтобы сделать нисходящий разбор эффективным, алгоритму необходим механизм, который за один шаг либо определяет правильное расширение, либо устанавливает тот факт, что правильное расширение отсутствует. В каком-то смысле он должен играть роль всезнающего оракула.

9	*Factor*	→	(	*Expr*)
10		|	num	
11		|	name	

Правила для *Factor*

Рассмотрим правила расширения *Factor* в грамматике выражений, повторенные во врезке. Нисходящий парсер, пытающийся расширить *Factor* для сопоставления с `name`, должен выбрать правило 11. Любое упорядочение правил 9, 10 и 11 приведет к неправильному выбору (и возврату) по крайней мере на двух из трех символов: (, `num` и `name`. Решение кажется очевидным: парсер должен использовать следующий входной символ – опережающий – для обоснования своего выбора. В этом случае комбинирование текущего *focus* с опережающим символом позволит ему выбрать правильную продукцию.

Мы можем модифицировать нисходящий парсер на рис. 3.3, так чтобы он всегда выбирал правильную продукцию в грамматике для *Factor*. Но обобщается ли этот механизм на другие грамматики?

Нисходящий парсер может выбрать правильную продукцию, если на каждом шаге вывода грамматика обладает тем свойством, что опережающий символ однозначно определяет правильный выбор.

Грамматика, свободная от возвратов
КСГ, для которой левосторонний нисходящий парсер всегда может предсказать нужное правило при ограниченном заглядывании вперед.

Иногда грамматики, свободные от возвратов, называют *предиктивными*.

Для такой грамматики нисходящий парсер может устранить неоднозначность выбора с помощью комбинации текущего *focus* и заглядывания вперед на один символ. Мы называем такую грамматику свободной *от возвратов* и отмечаем, что достаточно заглядывания только на один символ вперед. КСГ, свободные от возвратов, образуют большое подмножество.

Чтобы детерминированно производить эффективный выбор, парсер должен знать для каждого возможного расширения нетерминального символа A множество терминальных символов, которые могут встретиться в качестве первого символа строки, выведенной при таком расширении. Если $A \rightarrow \gamma_1 \mid \gamma_2 \mid \ldots \mid \gamma_n$, то парсер должен знать, какие терминальные символы могут встретиться в начале сентенциальной формы, выведенной из каждого γ_i. Если для каждой пары правых частей множества таких символов не пересекаются, то нисходящий парсер всегда может выбрать правильное расширение для A.

Интуитивно это понятно. В грамматике для *Factor* каждая правая часть начинается уникальным терминальным символом, поэтому устраняющие неоднозначность множества содержат по одному терминальному символу. Однако если какая-то правая часть начинается нетерминальным символом, то построение таких множеств осложняется.

Множество FIRST
Для символа грамматики α FIRST(α) – множество терминальных символов, которые могут встречаться в левой стороне сентенциальной формы, выведенной из α.

Для каждого символа грамматики α определим FIRST(α) как множество терминальных символов, которые могут встречаться в качестве первого слова некоторой сентенциальной формы, выведенной из α. Областью определения функции FIRST является $\{NT \cup T \cup \epsilon \cup \texttt{eof}\}$ – множество всех символов грамматики. Областью ее значений является $\{T \cup \epsilon \cup \texttt{eof}\}$ – множество символов грамматики за вычетом нетерминальных. (Отметим, что $\{T \cup \texttt{eof}\}$ – множество слов, возвращаемых сканером.) Если $\alpha \in \{T \cup \epsilon \cup \texttt{eof}\}$, то FIRST($\alpha$) = $\{\alpha\}$. Для нетерминального символа A множество FIRST(A) содержит полное множество терминальных символов, которые могут встречаться в качестве начального символа сентенциальной формы, выведенной из A.

Мы определили множества FIRST для одиночных символов грамматики. Удобно распространить определение FIRST с одиночного символа на строку символов. Для строки символов $s = \beta_1 \beta_2 \beta_3 \ldots \beta_k$ определим FIRST(s) как объединение множеств FIRST для $\beta_1 \beta_2 \beta_3 \ldots \beta_i$, где β_i – первый символ, для которого множество FIRST не содержит ϵ. Далее, $\epsilon \in$ FIRST(s) тогда и только тогда, когда она принадлежит FIRST(β_i) для всех β_i, $1 \leqslant i \leqslant k$.

`eof` неявно завершает каждое предложение. Таким образом, он принадлежит как области определения, так и области значений FIRST.

На рис. 3.7 показан алгоритм вычисления множеств FIRST. Сначала алгоритм заполняет множества FIRST для терминальных символов ϵ и `eof`. Для праворекурсивной грамматики выражений, показанной на рис. 3.5, этот начальный шаг порождает следующие множества FIRST:

	num	name	+	-	×	÷	(	)	eof	ϵ
FIRST	num	name	+	-	×	÷	(	)	eof	ϵ

Далее алгоритм в цикле обходит продукции. Для каждой продукции $A \rightarrow \beta$ он вычисляет FIRST(β), используя распространение с одиночного символа на строку. Алгоритм записывает множество FIRST всей правой части в переменную *rhs*, а затем прибавляет *rhs* к множеству FIRST нетерминального символа в левой части продукции. Этот процесс останавливается, когда достигает неподвижной точки. Для праворекурсивной грамматики выражений множества FIRST нетерминальных символов имеют вид:

	Expr	*Expr'*	*Term*	*Term'*	*Factor*
FIRST	(, name, num	+, -, ϵ	(, name, num	×, ÷, ϵ	(, name, num

```
for each α ∈ (T ∪ eof ∪ ϵ) do
  FIRST(α) ← α

for each A ∈ NT do
  FIRST(A) ← ∅

while (множества FIRST продолжают изменяться) do
  for each p ∈ P вида A → β1β2 ... βk do
    rhs ← FIRST(β1) – {ϵ}
    trailing ← true
    for i ← 1 to k-1 do
      if ϵ ∈ FIRST(βi)
        then rhs ← rhs ∪ (FIRST(βi+1) – {ϵ})
        else
          trailing ← false
          break
  if trailing and ϵ ∈ FIRST(βk) then
    rhs ← rhs ∪ {ϵ}
  FIRST(A) ← FIRST(A) ∪ rhs
```

Рис. 3.7 Вычисление множеств FIRST для символов грамматики

Концептуально множества FIRST упрощают реализацию нисходящего парсера. Например, в подграмматике для *Factor* множества FIRST делают выбор правильной продукции очевидным.

	Продукции	Множество FIRST
9	*Factor* → (*Expr*)	{ (}
10	| num	{ num }
11	| name	{ name }

Однако для подграмматики *Term* выбор не столь очевиден.

	Продукции	Множество FIRST
6	*Term'* → × *Factor Term'*	{×}
7	| ÷ *Factor Term'*	{÷}
8	| ϵ	{ϵ}

Для правил 6 и 7 выбор очевиден. Правило 8 ставит более трудную задачу. FIRST(ϵ) не совпадает ни с каким словом, возвращенным сканером.

Интуитивно кажется, что парсер должен применить продукцию ϵ, когда опережающий символ не является элементом множества FIRST никакой другой продукции – в данном случае когда опережающий символ не совпадает ни с ×, ни с ÷. Однако парсер не должен допускать вообще любое слово; одни слова приводят к выводам, другие – к синтаксическим ошибкам. Чтобы различить эти случаи, множество, используемое для правила 8, должно содержать любое слово, которое может встретиться в качестве начального символа после допустимого применения правила 8, – это множество слов, которые могут следовать за *Term'*.

Множество FOLLOW
Для нетерминального символа α множество FOLLOW(α) содержит слова, которые могут встречаться непосредственно после α в предложении.

Чтобы уловить это знание о структуре грамматики, мы определим FOLLOW(A) как множество, содержащее все слова, которые могут находиться непосредственно справа от строки, выведенной из A. Областью определения FOLLOW является *NT*, а областью значений – $T \cup \{\texttt{eof}\}$. На рис. 3.8 представлен алгоритм, который вычисляет множества FOLLOW для грамматики; в нем используются множества FIRST.

```
for each A ∈ NT do
  FOLLOW(A) ← ∅

FOLLOW(S) ← { eof }

while (множества FOLLOW продолжают изменяться) do
  for each p ∈ P вида A → β1β2β3 ... βk do
    TRAILER ← FOLLOW(A)
    for i ← k down to 1 do
      if βi ∈ NT then
        FOLLOW(βi) ← FOLLOW(βi) ∪ TRAILER
        if ϵ ∈ FIRST(βi) then
          TRAILER ← TRAILER ∪ (FIRST(βi) – ϵ)
        else TRAILER ← FIRST(βi)
      else TRAILER ← {βi}  // βi ∈ T
```

Рис. 3.8 Вычисление множеств FOLLOW для нетерминальных символов

Алгоритм вычисления множества FOLLOW несколько хитрее, чем вычисления множества FIRST. В самом начале он делает все множества FOLLOW пустыми. Затем в качестве множества FOLLOW начального символа берется {eof}. В основной своей части алгоритм обходит все продукции, уточняя множества FOLLOW. И завершается, когда эти множества достигают неподвижной точки.

В построении используется соседство в правой части для обновления множеств FOLLOW нетерминальных символов в этой правой части. Если β_i β_{i+1} встречается в какой-то правой части, то FOLLOW(β_i) должно содержать FIRST(β_{i+1}). Правила с ϵ осложняют ситуацию. Если $\epsilon \in$ FIRST(β_{i+1}), то FOLLOW(β_i) должно также содержать символы, которые могут следовать за β_{i+1}.

Чтобы уловить этот эффект, алгоритм обходит правую часть каждой продукции, $A \rightarrow \beta_1\beta_2 \ldots \beta_k$ от β_k до β_1. Для каждого суффикса

$$\beta_k;\ \beta_{k-1}\beta_k;\ \ldots;\ \beta_1\beta_2 \ldots \beta_k$$

он строит множество FIRST. Переменная *TRAILER* содержит эти множества FIRST; если из некоторого β_i можно вывести ϵ, то *TRAILER* переносит вперед правый контекст. Если из β_i нельзя вывести ϵ, то алгоритм усекает этот правый контекст и полагает *TRAILER* равным FIRST(β_i).

Множества FOLLOW для праворекурсивной грамматики выражений имеют вид:

	Expr	*Expr'*	*Term*	*Term'*	*Factor*
FIRST	eof,)	eof,)	eof, +, -,)	eof, +, -,)	eof, +, -, ×, ÷,)

Мы ввели множества FOLLOW, чтобы парсер мог правильно выбрать продукции ϵ, такие как в подграмматике для *Term'*. Парсер должен выбрать между тремя вариантами правой части *Term'*.

FIRST(× *Factor Term'*) = { × }
FIRST(÷ *Factor Term'*) = { ÷ }

Множества FIRST делают очевидным выбор, когда опережающий символ равен × или ÷. Трудности возникают, когда опережающий символ не равен ни ×, ни ÷. Должен ли парсер расширить продукцию на ϵ или выдать синтаксическую ошибку?

Множества FOLLOW позволяют парсеру принять это решение. Если опережающий символ $l \in$ FOLLOW(*Term'*), то парсер должен расширить на ϵ. Если l не равен ни ×, ни ÷ и $l \notin$ FOLLOW(*Term'*), то вход содержит синтаксическую ошибку.

Комбинация FIRST и FOLLOW позволяет точно определить правильное поведение парсера. Для некоторой продукции $A \rightarrow \beta$:

$$\text{START}(A \rightarrow \beta) = \begin{cases} \text{FIRST}(\beta), \text{ если } \epsilon \notin \text{FIRST}(\beta) \\ (\text{FIRST}(\beta) - \epsilon) \cup \text{FOLLOW}(A) \text{ в противном случае} \end{cases}.$$

На рис. 3.9 показаны множества START для праворекурсивной грамматики выражений. Нисходящий парсер может использовать множества START, чтобы выбрать правильное расширение для любого нетерминального символа *A* путем сравнения опережающего символа с множествами START альтернативных правых частей для *A*.

Продукция			Множество START
Goal	→	*Expr*	{ (, name, num }
Expr	→	*Term Expr'*	{ (, name, num }
Expr'	→	+ *Term Expr'*	{ + }
	|	- *Term Expr'*	{ - }
	|	ϵ	{ eof,) }
Term	→	*Factor Term'*	{ (, name, num }
Term'	→	*Factor Term'*	{ × }
	|	÷ *Factor Term'*	{ ÷ }
	|	ϵ	{ eof, +, -,) }
Factor	→	(*Expr*)	{ (}
	|	num	{ num }
	|	name	{ name }

Рис. 3.9 Множества START для праворекурсивной грамматики выражений

Это обсуждение ведет к простому и эффективному критерию возможности разобрать грамматику слева направо нисходящим парсером без необходимо-

сти возвратов. Грамматика *свободна от возвратов* тогда и только тогда, когда для любого нетерминального символа A с несколькими правыми частями $A \to \beta_1 \mid \beta_2 \mid \ldots \mid \beta_n$ имеет место:

$$\text{START}(A \to \beta_i) \cap \text{START}(A \to \beta_j) = \emptyset, \forall\ 1 \leqslant i, j \leqslant n, i \neq j.$$

Если грамматика обладает этим свойством, то нисходящий парсер всегда может выбрать правильное расширение для нетерминального символа A, зная только один опережающий символ. Это свойство называется условием LL(1), потому что обладающие им грамматики можно разобрать LL(1)-парсером (см. раздел 3.3.3).

Применение левой факторизации для устранения возвратов

Не все грамматики свободны от возвратов. Например, рассмотрим включение в грамматику выражений вызовов функций, обозначаемых (и), и ссылок на элементы массивов, обозначаемых [и]. Мы можем продолжить правило *Factor* → `name` тремя правилами плюс продукции для описания списков аргументов:

Factor → `name` – правило 11 на рис. 3.5.

```
11  Factor     →  name
12             |  name [ ArgList ]
13             |  name ( ArgList )
15  ArgList    →  Expr MoreArgs
16  MoreArgs   →  , Expr MoreArgs
17             |  ϵ
```

Поскольку каждое из правил 11, 12 и 13 начинается с `name`, их множества START одинаковы. Когда парсер пытается продолжить экземпляр *Factor* опережающим символом `name`, у него нет никаких оснований для выбора между 11, 12 и 13. Заглядывания вперед на одно слово недостаточно для разрешения сомнений. В этом случае мы можем переписать грамматику, создав непересекающиеся множества START.

Этот случай можно было бы разрешить заглядыванием вперед на два слова. Однако для любого конечного заглядывания вперед можно придумать грамматику, для которой такого заглядывания будет недостаточно.

```
11  Factor     →  name Arguments
12  Arguments  →  [ ArgList ]
13             |  ( ArgList )
14             |  ϵ
```

Такая перезапись, называемая *левой факторизацией*, разбивает *Factor* на два шага. На первом шаге сопоставляется общий префикс оригинальных правил 11, 12 и 13. На

Левая факторизация
процесс выделения и изоляции общих префиксов в множестве продукций.

втором шаге распознаются три разных суффикса: [*ArgList*], (*ArgList*) и ϵ. Добавляется нетерминальный символ *Arguments*, и альтернативные суффиксы *Factor* перемещаются в расширения *Arguments*.

Если нетерминальный символ имеет несколько правых частей с общим префиксом, то с помощью левой факторизации можно вынести правила распознавания этого префикса в новый нетерминальный символ. В этом преобразовании участвуют нетерминальный символ и его продукции:

$$A \rightarrow \alpha\beta_1 \mid \alpha\beta_2 \mid \ldots \mid \alpha\beta_n \mid \gamma_1 \mid \gamma_2 \mid \ldots \mid \gamma_j,$$

где α – общий префикс, а различные γ_i представляют правые части, не начинающиеся с α. Преобразование вводит новый нетерминальный символ B для представления альтернативных суффиксов α и перезаписывает оригинальную продукцию по схеме

$$A \rightarrow \alpha B \mid \gamma_1 \mid \gamma_2 \mid \ldots \mid \gamma_j$$
$$B \rightarrow \beta_1 \mid \beta_2 \mid \ldots \mid \beta_n$$

Чтобы подвергнуть левой факторизации всю грамматику целиком, мы должны систематически исследовать каждый нетерминальный символ, выявить общие префиксы и применить описанное преобразование. Например, в схеме выше мы должны рассмотреть факторизацию правых частей B, поскольку два или более β_i могли бы иметь общий префикс. Процесс останавливается, когда все общие префиксы будут устранены.

Левая факторизация часто способна устранить необходимость возврата. Однако у некоторых контекстно-свободных языков не существует грамматики, свободной от возвратов. Их следует разбирать с помощью более общих методов, например LR(1)-парсерами, описанными в следующем разделе. Пример языка, который не имеет грамматики, свободной от возвратов, но может быть разобран LR(1)-парсером, приведен в упражнении 3.10.

Если дана произвольная КСГ, то автор компилятора может систематически устранить левую рекурсию и подвергнуть общие префиксы левой факторизации. Эти изменения могут породить грамматику, свободную от возвратов. Но в общем случае невозможно решить, существует ли для произвольного контекстно-свободного языка грамматика, свободная от возвратов.

3.3.2. Нисходящие парсеры рекурсивного спуска

Грамматики, свободные от возвратов, могут быть разобраны с помощью простой и эффективной парадигмы, называемой *рекурсивным спуском*. Парсер рекурсивного спуска оформляется в виде набора взаимно рекурсивных процедур, по одной для каждого нетерминального символа. Процедура, соответствующая нетерминальному символу A, распознает экземпляр A во входном потоке. Чтобы распознать нетерминальный символ B в какой-то правой части A, парсер вызывает процедуру, соответствующую B. Таким образом, сама грамматика направляет реализацию парсера в нужную сторону.

Напомним правила для *Term'* в грамматике выражений:

	Продукции	Множество START
6	*Term'* → × *Factor Term'*	{x}
7	| × *Factor Term'*	{ ÷ }
8	| ϵ	{ eof, +, -,) }

Чтобы распознать экземпляры *Term'*, мы создадим функцию *TermPrime()*. Она устроена просто: выбрать одно из трех правил (или сообщить о синтаксической ошибке) на основе множеств START их правых частей. Для каждой правой части код непосредственно проверяет следующие символы.

3.3.3. Таблично управляемые LL(1)-парсеры

Чтобы проверить наличие нетерминального символа, скажем *A*, код вызывает процедуру, соответствующую *A*. Для проверки наличия терминального символа, скажем `name`, он выполняет прямое сравнение, и если оно успешно, то продвигает входной поток, вызывая сканер, *NextWord()*. В случае обнаружения ϵ-продукции код не вызывает *NextWord()*. На рис. 3.10 показан псевдокод парсера рекурсивного спуска для выражений. Взгляните на реализацию *TermPrime* в правом верхнем углу; она следует этой схеме в том смысле, что

```
Main( )  /* Goal → Expr */
    word ← NextWord( );
    if (Expr( ))
        then if (word = eof )
            then report success;
            else Fail( );

Fail( )
    сообщить о синтаксической ошибке;
    попробовать восстановиться после ошибки
    или выйти;

Expr( ) /* Expr → Term Expr' */
    if ( Term( ) )
        then return ExprPrime( );
        else Fail();

ExprPrime( )
    /* Expr' → + Term Expr'  | - Term Expr' */
    if (word = + or word = - ) then
        word ← NextWord( );
        if ( Term() )
            then return ExprPrime( );
            else Fail();

    /* Expr → ϵ */
    else if (word = ) or word = eof)
        then return true;
        else Fail();

Term( ) /* Term → Factor Term' */
    if ( Factor( ) )
        then return TermPrime( );
        else Fail();
```

```
TermPrime( )
    /* Term' → × Factor Term'
           | ÷ Factor Term' */
    if (word = × or word = ÷ ) then
        word ← NextWord( );
        if ( Factor( ) )
            then return TermPrime( );
            else Fail();

    /* Term → _ */
    else if (word = + or word = - or
             word = ) or word = eof)
        then return true;
        else Fail();

Factor( )
    /* Factor → ( Expr ) */
    if (word = ( ) then
        word ← NextWord( );
        if (not Expr( ) )
            then Fail();
        if (word ≠ ) )
            then Fail();
        word ← NextWord( );
        return true;

    /* Factor → num | name */
    else if (word = num or
             word = name ) then
        word ← NextWord( );
        return true;
        else Fail();
```

Рис. 3.10 Парсер рекурсивного списка для выражений

имеются явные проверки символов в различных множествах START. Здесь объединен код правил 6 и 7.

Стратегия построения полного парсера рекурсивного спуска понятна. Для каждого нетерминального символа мы строим процедуру, которая распознает его альтернативные правые части. Эти процедуры вызывают друг друга для распознавания нетерминальных символов. Терминальные символы распознаются прямым сравнением. Каждая процедура возвращает индикатор успеха или вызывает *Fail*.

Для небольшой грамматики автор компилятора может быстро создать парсер рекурсивного спуска. Такой парсер способен выдавать точные и информативные сообщения об ошибках. Естественно выдавать сообщения, когда парсер не находит ожидаемого терминального символа – в нашем примере это бывает внутри функций *ExprPrime*, *TermPrime* и *Factor*.

Генератор парсеров
инструмент, который строит парсер по спецификациям, обычно в виде БНФ-подобных грамматик.

Зная множества START, мы можем автоматизировать генерирование нисходящих парсеров для грамматик, свободных от возвратов. Для этого создадим инструмент, *генератор парсеров*, который строит множества FIRST, FOLLOW и START, а затем использует множества START для конструирования нисходящего парсера.

Грамматики, свободные от возвратов, часто называют LL(1)-грамматиками.

В одной из схем построения нисходящих парсеров создаются таблично управляемые LL(1)-парсеры. Любая свободная от возвратов грамматика может быть разобрана LL(1)-парсером. Название LL(1) проистекает из того факта, что эти парсеры сканируют вход слева (**L**eft) направо, находят левосторонний (**L**eftmost) вывод и используют заглядывание на **1** символ вперед.

Чтобы построить LL(1)-парсер, автор компилятора создает праворекусивную свободную от возвратов грамматику. Генератор парсеров строит фактический парсер. В большинстве генераторов LL(1)-парсеров применяется таблично управляемая заготовка парсера типа показанной на рис. 3.11. Генератор же строит только таблицу. На рис. 3.12 показана LL(1)-таблица для праворекурсивной грамматики выражений.

В заготовке парсера переменная `focus` содержит следующий символ грамматики, который должен быть сопоставлен на нижней кромке частично построенного дерева разбора, – такую же роль она играла в нисходящем парсере на рис. 3.3. LL(1)-таблица отображает пары, состоящие из нетерминальных символов и опережающих символов (терминальных или `eof`) в продукции. Для заданных нетерминального символа *A* и опережающего символа *w Table[A, w]* определяет правильное расширение.

В примере на рис. 3.13(а) показаны действия LL(1)-парсера выражений для входной строки `a + b × c`. В среднем столбце показано содержимое стека парсера, в котором хранится нижняя кромка частично построенного дерева разбора. Разбор завершается успешно, если он выталкивает из стека символ *Expr'*, оставляя в стеке `eof`, при этом неявным опережающим символом должен быть `eof`.

```
word ← NextWord( )
поместить eof в стек
поместить начальный символ S в стек

while( true ) do
  focus ← вершина стека
  if (focus = eof and word = eof)
    then сообщить об успехе и выйти из цикла
  else if (focus ∈ T or focus = eof) then
    if focus совпадает с word then
      вытолкнуть вершину стека
      word ← NextWord( )
    else сообщить об ошибке поиска символа в focus
  else  // focus является нетерминальным символом
    if Table[focus,word] имеет вид A → β1β2 ... βk then
      вытолкнуть вершину стека
      for i ← k to 1 by -1 do
        if (βi ≠ ϵ) then
          поместить βi в стек
    else сообщить об ошибке при расширении focus
```

Рис. 3.11 Заготовка LL(1)-парсера

	eof	+	-	×	÷	(	)	num	name
Goal						0		0	0
Expr						1		1	1
Expr'	4	2	3				4		
Term						5		5	5
Term'	8	8	8	6	7		8		
Factor						9		10	11

Рис. 3.12 Таблица LL(1)-разбора праворекурсивной грамматики выражений

Как видно по рис. 3.13(a), LL(1)-парсер эффективен. Время его работы пропорционально размеру вывода. При выводе он заталкивает каждый символ грамматики в стек, а позднее выталкивает их из стека. Невозможно ожидать, что парсер построит вывод за меньшее количество шагов.

Теперь рассмотрим действия LL(1)-парсера для недопустимой входной строки `x + ÷ y`, показанные на рис. 3.13(b). Он распознает первое имя `name` и знак +. В этот момент на вершине стека находится *Term*, опережающим символом является ÷. Соответствующий элемент таблицы содержит *error*.

В *Table* имеется строка для каждого нетерминального символа и столбец для каждого терминального символа. Алгоритм заполнения таблицы прямолинеен. В нем предполагается, что для грамматики доступны множества START.

В построении, показанном на рис. 3.14, каждой продукции присваивается порядковый номер, а все элементы таблицы инициализируются значением *error*. Затем для каждой продукции $A \rightarrow \beta$ и каждого символа $w \in \text{START}(A \rightarrow \beta)$

алгоритм записывает в элемент на пересечении строки *A* и столбца *w* номер этой продукции. Для праворекурсивной грамматики выражений получается таблица, показанная на рис. 3.12.

Правило	Стек	Вход
—	eof *Goal*	↑ name + name × name
0	eof *Expr*	↑ name + name × name
1	eof *Expr′ Term*	↑ name + name × name
5	eof *Expr′ Term′ Factor*	↑ name + name × name
11	eof *Expr′ Term′* name	↑ name + name × name
→	eof *Expr′ Term′*	name ↑ + name × name
8	eof *Expr′*	name ↑ + name × name
2	eof *Expr′ Term* +	name ↑ + name × name
→	eof *Expr′ Term*	name + ↑ name × name
5	eof *Expr′ Term′ Factor*	name + ↑ name × name
11	eof *Expr′ Term′* name	name + ↑ name × name
→	eof *Expr′ Term′*	name + name ↑ × name
6	eof *Expr′ Term′ Factor* ×	name + name ↑ × name
→	eof *Expr′ Term′ Factor*	name + name × ↑ name
11	eof *Expr′ Term′* name	name + name × ↑ name
→	eof *Expr′ Term′*	name + name × name ↑
8	eof *Expr′*	name + name × name ↑
4	eof	name + name × name ↑

(a) Действия LL(1)-парсера для строки a + b × c

Правило	Стек	Вход
—	eof *Goal*	↑ name + ÷ name
0	eof *Expr*	↑ name + ÷ name
1	eof *Expr′ Term*	↑ name + ÷ name
5	eof *Expr′ Term′ Factor*	↑ name + ÷ name
11	eof *Expr′ Term′* name	↑ name + ÷ name
→	eof *Expr′ Term′*	name ↑ + ÷ name
8	eof *Expr′*	name ↑ + ÷ name
2	eof *Expr′ Term* +	name ↑ + ÷ name
→	eof *Expr′* *Term*	name + ↑ ÷ name

синтаксическая ошибка в этом месте

(b) Действия LL(1)-парсера для строки x+ ÷y

Рис. 3.13 Примеры LL(1)-разбора

```
for each нетерминального символа A do
  Table[A,eof] ← error
  for each терминального символа w do
    Table[A,w] ← error

  for each продукции p вида A → β do
   for each терминального символа w ∈ START(A → β) do
      Table[A,w] ← p
    if eof ∈ START(A → β)
      then Table[A,eof] ← p
```

Рис. 3.14 Алгоритм построения LL(1)-таблицы

Для грамматики, обладающей свойством LL(1) (см. выше), это построение создает правильную таблицу за время $\mathbf{O}(|P| \times |T|)$, где P – множество продукций, а T – множество терминальных символов.

Если построение записывает несколько номеров продукций в *Table[A, w]*, значит, символ *w* встречается в множествах START нескольких правых частей для *A*, что нарушает условие LL(1). Поскольку элементы *Table* инициализированы значением *error*, построитель таблицы может проверять это перед каждой записью и сообщать об ошибке, если ячейка, в которую производится запись, не содержит *error*.

В качестве альтернативы таблично управляемому парсеру генератор LL(1)-парсеров мог бы просто порождать парсер рекурсивного спуска. Этот процесс похож на тот, который строит таблично управляемый парсер. Сначала генератор парсеров должен был бы построить множества FIRST, FOLLOW и START. Затем он должен был бы обойти нетерминальные символы из левых частей правил грамматики. Для каждого такого символа он выводит небольшую процедуру для распознавания его правых частей. Для принятия решений код должен был бы использовать множества START. Такая система могла бы сочетать скорость и локальность парсера рекурсивного спуска с удобством генератора на основе грамматики.

ИТОГИ РАЗДЕЛА

Нисходящие парсеры просты, компактны и эффективны. Их можно реализовать разными способами, в т. ч. в виде рукописных парсеров, парсеров рекурсивного спуска и сгенерированных LL(1)-парсеров. Зная в каждой точке разбора, какие слова могут быть следующими в допустимой входной строке, они могут выдавать полезные сообщения об ошибках.

Большинство конструкций языков программирования можно выразить грамматикой, свободной от возвратов. Поэтому такие методы получили широкое распространение. Требование, чтобы альтернативные правые части имели непересекающиеся множества START, не слишком сильно ограничивает полезность LL(1)-грамматик. Основной недостаток нисходящих предиктивных парсеров – их неспособность обработать левую рекурсию.

ВОПРОСЫ

1. Чтобы построить эффективный нисходящий парсер, автор компилятора должен выразить исходный язык в несколько ограниченной форме. Объясните, какие ограничения на грамматику исходного языка делают ее пригодной для эффективного нисходящего разбора.

2. Назовите два потенциальных преимущества рукописного парсера рекурсивного спуска перед сгенерированным таблично-управляемым LL(1)-парсером и два преимущества LL(1)-парсера перед реализацией на основе рекурсивного спуска.

3.4. Восходящий разбор

Восходящие парсеры ищут вывод, отправляясь от слов программы и продвигаясь в сторону начального символа, *S*. Мы можем рассматривать этот процесс как построение дерева разбора от листьев к корню. Для вывода

$$Start = \gamma_0 \rightarrow \gamma_1 \rightarrow \gamma_2 \rightarrow \ldots \rightarrow \gamma_{n-1} \rightarrow \gamma_n = \text{предложение}$$

восходящий парсер раз за разом на основе γ_i и грамматики находит продукцию $A \rightarrow \beta$ и место k в γ_i такое, что замена вхождения β, заканчивающегося в позиции k γ_i, на A порождает γ_{i-1}.

Окончание
Окончанием (handle) сентенциальной формы γ_i называется пара $\langle A \rightarrow \beta, k \rangle$ такая, что правый конец вхождения β в сентенциальную форму оказывается в позиции k и замена β на A дает предыдущий шаг вывода, γ_{i-1}.

Пара $\langle A \rightarrow \beta, k \rangle$ описывает этот переход от γ_i к γ_{i-1}. Мы называем эту пару *окончанием* (*handle*). Парсер в цикле находит окончание на кромке и подставляет A вместо β. В правильном выводе из однозначной грамматики у каждого γ_i должно быть только одно окончание, для всех $0 < i \leqslant n$.

Восходящий парсер ищет последовательность окончаний, определяющую допустимый вывод входной программы. На рис. 3.15 показан высокоуровневый алгоритм парсера типа перенос–свертка.

```
поместить INVALID в стек
word ← NextWord()
repeat until (вершина стека = S and word = eof) do
  if на вершине стека находится окончание A → β then
    // свернуть β в A
    вытолкнуть |β| символов из стека
    поместить A в стек
  else if (word ≠ eof) then
    // перенести слово в стек
    поместить слово в стек
    word ← NextWord()
  else // парсер должен перенести, но на входе больше ничего нет
    сообщить о синтаксической ошибке

сообщить об успехе
```

Рис. 3.15 Простой парсер типа перенос–свертка

Парсер повторяет простой процесс. Он в цикле переносит слова – пары ⟨*лексема*, *категория*⟩, полученные от сканера, – в стек, пока не найдет в сте-

ке окончание. Правый конец окончания окажется на вершине стека. Найдя окончание для $A \rightarrow \beta$, парсер выталкивает из стека записи, относящиеся к β, и помещает в стек записи для A – в терминологии восходящего разбора это действие называется *сверткой*.

Парсер продолжает выполнять переносы и свертки до тех пор, пока не случится одно из двух.

1. Парсер нашел начальный символ грамматики, S, на вершине стека. Если это так и опережающим символом является `eof`, то разбор завершился успешно. Если же опережающий символ отличен от `eof`, то нужно обработать дополнительные входные данные.
2. Сканер вернул `eof`, а парсер не выполнил свертку стека в S. В этом случае парсер потребил весь вход, но так и не нашел правильного вывода, а значит, должен сообщить о *синтаксической ошибке* – входная программа содержит одну или более ошибок.

Этот механизм обнаружения ошибок не локализует место ошибки. LR(1)-парсеры обнаруживают ошибки там, где они произошли.

Ключ к практическому восходящему разбору – умение эффективно распознавать окончания. LR(1)-парсер реализует особенно эффективный механизм нахождения окончаний. Мы будем не раз возвращаться к вопросу окончаний и их нахождения на протяжении раздела 3.4. Но сначала закончим с верхнеуровневым описанием нисходящих парсеров.

Рассмотрим расширение парсера типа перенос–свертка, которое строит само дерево разбора. Каждый перенос строит листовый узел, представляющий опережающий символ, а затем продвигает опережающий символ, вызывая сканер. Каждая свертка добавляет узел, представляющий нетерминальный символ, на вершину частично построенного дерева разбора. В этих внутренних узлах нетерминальных символов закодирована грамматическая структура, обнаруженная при выводе входной строки.

На любой стадии разбора частично построенное дерево представляет состояние разбора. Каждый листовый узел представляет слово; чтение листьев слева направо дает префикс входной программы. Внутренние узлы, расположенные выше листьев, кодируют части вывода, которые парсер уже нашел. Парсер работает на верхней кромке этого частично построенного дерева разбора; эта кромка соответствует текущей сентенциальной форме в выводе, который строит парсер.

Чтобы продолжить кромку вверх, парсер ищет окончание $\langle A \rightarrow \beta, k \rangle$. Таким образом, β встречается на кромке, так что ее правый конец находится в позиции k, и парсер знает, что замена β на A создаст на кромке предыдущую сентенциальную форму в выводе. (Если кромка представляет γ_i, то свернутая кромка будет представлять γ_{i-1}.) Этот процесс строит вывод наоборот, т. е. от предложения к начальному символу, а не от начального символа к предложению.

Наш концептуальный парсер строит правосторонний вывод в обратном порядке. Этот порядок согласует сканирование слева направо с правосторон-

ним выводом. При правостороннем выводе самый левый лист считается последним. Обращение этого порядка дает желаемое поведение: самый левый лист является первым, а самый правый – последним.

В каждой точке парсер типа перенос–свертка работает с кромкой частично построенного дерева разбора; содержимое стека, читаемое снизу вверх, является префиксом соответствующей сентенциальной формы в выводе. Поскольку каждая сентенциальная форма встречается в самом правом выводе, еще не исследованный суффикс – часть за пределами вершины стека – состоит только из терминальных символов. Когда парсеру нужен контекст, расположенный правее, он вызывает сканер.

Для однозначной грамматики правосторонний вывод единственный. Для большого класса однозначных грамматик γ_{i-1} можно определить непосредственно по γ_i (верхней кромке дерева разбора) с ограниченным заглядыванием вперед во входном потоке. Иными словами, если дана кромка γ_i и ограниченное число опережающих символов, то парсер может найти, эффективно и детерминированно, окончание, которое переводит γ_i в γ_{i-1}.

LR(1)-парсеры – особенно эффективная реализация этих ищущих окончания парсеров типа перенос–свертка. LR(1)-парсер сканирует вход слева направо с целью построить правосторонний вывод в обратном порядке. На каждом шаге он принимает решения, основываясь на истории разбора и заглядывании не более чем на один символ вперед. Название LR(1) происходит от этих свойств: сканирование слева направо (**L**eft-to-right), правосторонний обратный (**R**everse) вывод и **1** опережающий символ. Инструменты создания LR(1)-парсеров широкодоступны.

Неформально мы будем говорить, что язык обладает свойством LR(1), если его можно разобрать за один проход сканирования слева направо и построить обратный правосторонний вывод с заглядыванием вперед только на один символ. На практике определить, обладает ли грамматика свойством LR(1), проще всего, поручив генератору парсеров построить LR(1)-парсер. Если он выдаст ошибку, значит, грамматика не обладает свойством LR(1).

Оставшуюся часть этого раздела мы посвятим подробному введению в теорию LR(1)-разбора. В разделе 3.4.1 представлены LR(1)-парсеры и их функционирование. В разделе 3.4.2 приведен алгоритм построения таблиц, которые кодируют действия LR(1)-парсера. Наконец, в разделе 3.4.3 показано, что бывает, когда это построение применяется к грамматике, не обладающей свойством LR(1).

3.4.1. Алгоритм LR(1)-разбора

Важнейший шаг восходящего парсера, например таблично управляемого LR(1)-парсера, – поиск следующего окончания. Эффективное обнаружение окончаний – ключ к эффективному восходящему разбору. LR(1)-парсер использует ищущий окончания автомат, закодированный в двух таблицах, которые традиционно называются *Action* и *Goto*. На рис. 3.16 показан простой таблично управляемый LR(1)-парсер.

```
поместить ⟨INVALID, INVALID⟩ в стек
поместить ⟨start symbol, s0⟩ в стек
word ← NextWord( )

while (true) do
  state ← state из пары на вершине стека
  if Action[state,word] = "reduce A → β" then
    вытолкнуть |β| пар из стека
    state ← state из пары на вершине стека
    поместить ⟨A, Goto[state, A]⟩ в стек
  else if Action[state,word] = "shift si" then
    поместить ⟨word, si⟩ в стек
    word ← NextWord( )
  else if Action[state,word] = "accept" and word = eof
    then break
  else сообщить о синтаксической ошибке

сообщить об успехе /* выполнен случай "accept" */
```

Рис. 3.16 Заготовка LR(1)-парсера

Область, следующая за находящимся в стеке префиксом, целиком состоит из терминальных символов. Первым словом в этой области является опережающий символ.

Заготовка LR(1)-парсера интерпретирует таблицы *Action* и *Goto* с целью найти последовательные окончания в обратном правостороннем выводе входа. Обнаружив окончание $\langle A \rightarrow \beta,\ k\rangle$, парсер сворачивает β в позиции k в A в текущей сентенциальной форме – верхней кромке частично построенного дерева разбора. Вместо того чтобы строить явное дерево разбора, заготовка парсера сохраняет префикс верхней кромки дерева в стеке. Каждый элемент стека – это пара $\langle A, s\rangle$, где A – символ грамматики, а s – состояние парсера. Состояния связывают вместе редукции в процессе разбора. В переменной *word* хранится опережающий символ – первое слово за границами содержимого стека.

Использование стека позволяет LR(1)-парсеру сделать позицию k в окончании постоянной и неявной.

Благодаря этому упрощению множество окончаний оказывается конечным и может быть распознано ДКА.

Чтобы найти следующее окончание, LR(1)-парсер переносит символы в стек до тех пор, пока автомат не обнаружит на вершине стека правый конец окончания. Получив окончание, $A \rightarrow \beta$, парсер производит свертку, заменяя β на A. Он выталкивает из стека символы, принадлежащие β, и ассоциированные с ними состояния и помещает в стек A и свое новое состояние. Таблицы *Action* и *Goto* связывают вместе действия (переносы и свертки) и состояния в управляемую грамматикой последовательность, чтобы найти правосторонний вывод, если таковой существует.

Чтобы добавить конкретики, рассмотрим грамматику на рис. 3.17(а), которая описывает язык, состоящий из правильно вложенных скобок. На рисунке

(b) показаны таблицы *Action* и *Goto* для этой грамматики. В них закодирован ищущий окончания автомат для скобочного языка.

1	*Goal*	→	*List*
2	*List*	→	*List Pair*
3		|	*Pair*
4	*Pair*	→	(*List*)
5		|	( )

(a) Скобочная грамматика

	Таблица *Action*			Таблица *Goto*	
Состояние	eof	(	)	*List*	*Pair*
0		s 3		1	2
1	acc	s 3			4
2	r 3	r 3			
3		s 7	s 8	5	6
4	r 2	r 2			
5		s 7	s 10		9
6		r 3	r 3		
7		s 7	s 12	11	6
8	r 5	r 5			
9		r 2	r 2		
10	r 4	r 4			
11		s 7	s 13		9
12		r 5	r 5		
13		r 4	r 4		

(b) Таблицы действий и переходов для скобочной грамматики

Рис. 3.17 Скобочная грамматика

Символ $ в стеке представляет пару ⟨*INVALID*, *INVALID*⟩.

На рис. 3.18 показана последовательность действий, выполняемых парсером для входной строки «( )». В каждой строке представлена одна итерация парсера, а в первой строке – начальное состояние. В последнем столбце показано действие парсера, продиктованное таблицей на этой итерации.

Итерация	Состояние	Слово	Стек	Окончание	Действие
0	–	(	$ ⟨*Goal* 0⟩	*– нет –*	–
1	0	(	$ ⟨*Goal* 0⟩	*– нет –*	*shift 3*
2	3	)	$ ⟨*Goal* 0⟩ ⟨(3⟩	*– нет –*	*shift 8*
3	8	eof	$ ⟨*Goal* 0⟩ ⟨(3⟩ ⟨) 8⟩	( )	*reduce 5*
4	2	eof	$ ⟨*Goal* 0⟩ ⟨*Pair* 2⟩	*Pair*	*reduce 3*
5	1	eof	$ ⟨*Goal* 0⟩ ⟨*List* 1⟩	*List*	*accept*

Рис. 3.18 Состояния LR(1)-парсера на входе ( )

В начале первой итерации в стеке нет окончания. Таблица *Action* говорит парсеру, что нужно перенести (в стек и перейти в состояние 3. В начале второй итерации в стеке по-прежнему нет окончания; таблица *Action* заставляет парсер перенести) в стек и перейти в состояние 8.

На третьей итерации стек содержит окончание *Pair* → ( ); его правый конец находится на вершине стека. Таблица *Action* говорит, что парсер должен

свернуть ( ) в *Pair*. Используя состояние под *Pair* в стеке, 0, и *Pair*, парсер переходит в состояние *Goto[0,Pair] = 2*.

Цепочка окончаний, (5, 3, 1), показывает правосторонний вывод в обратном порядке.

На четвертой итерации *Pair* находится на вершине стека. Парсер находит окончание *List* → *Pair*, производит свертку и переходит в состояние *Goto[0,List] = 1*. Наконец, в состоянии 1 парсер находит окончание *Goal* → *List* и принимает строку.

Действие *accept* – это специальное действие, которое сигнализирует о свертке начального символа с опережающим символом `eof`.

Для входа ( ) парсер использовал два переноса, две свертки и одну приемку. В общем случае время работы LR(1)-парсера пропорционально длине входа (по одному переносу на слово) и длине вывода (по одной свертке на каждый шаг вывода). Таким образом, время разбора ограничено снизу длиной вывода.

На рис. 3.19 показано поведение парсера для входной строки «(( )) ( )». Парсер выполняет шесть переносов, шесть сверток и одну приемку. На рис. 3.20 показано состояние частично построенного дерева в конце каждой итерации, от 1 до 12. Действие приемки могло бы добавить узел для символа *Goal* в случае необходимости. В верхней части каждого рисунка показан номер итерации и серая планка, содержащая верхнюю кромку частично построенного дерева, – символы в стеке LR(1)-парсера.

Итерация	Состояние	Слово	Стек	Окончание	Действие
0	–	(	$ ⟨*Goal* 0⟩	*– нет –*	–
1	0	(	$ ⟨*Goal* 0⟩	*– нет –*	*shift 3*
2	3	(	$ ⟨*Goal* 0⟩ ⟨(3⟩	*– нет –*	*shift 7*
3	7	)	$ ⟨*Goal* 0⟩ ⟨(3⟩ ⟨) 7⟩	*– нет –*	*shift 12*
4	12	)	$ ⟨*Goal* 0⟩ ⟨(3⟩ ⟨) 7⟩ ⟨) 12⟩	( )	*reduce 5*
5	6	)	$ ⟨***Goal*** **0**⟩ ⟨(3⟩ ⟨*Pair* 6⟩	*Pair*	*reduce 3*
6	5	)	$ ⟨***Goal*** **0**⟩ ⟨(3⟩ ⟨*List* 5⟩	*– нет –*	*shift 10*
7	10	(	$ ⟨***Goal*** **0**⟩ ⟨(3⟩ ⟨*List* 5⟩ ⟨) 10⟩	(*List*)	*reduce 4*
8	2	(	$ ⟨*Goal* 0⟩ ⟨*Pair* 2⟩	*Pair*	*reduce 3*
9	1	(	$ ⟨*Goal* 0⟩ ⟨*List* 1⟩	*– нет –*	*shift 3*
10	3	)	$ ⟨*Goal* 0⟩ ⟨*List* 1⟩ ⟨(3⟩	*– нет –*	*shift 8*
11	8	`eof`	$ ⟨*Goal* 0⟩ ⟨*List* 1⟩ ⟨(3⟩ ⟨(8⟩	( )	*reduce 5*
12	4	`eof`	$ ⟨*Goal* 0⟩ ⟨*List* 1⟩ ⟨*Pair* 4⟩	*List Pair*	*reduce 2*
13	1	`eof`	$ ⟨*Goal* 0⟩ ⟨*List* 1⟩	*List*	*accept*

Рис. 3.19 Состояния LR(1)-парсера на входе (( )) ( )

Поиск окончания

Способность LR(1)-парсера находить окончания в стеке – ключ к его работе. Рассмотрим действия парсера для строки «( )», показанной на рис. 3.18. На итерациях 1 и 2 на вершине стека нет окончаний, поэтому выполняется перенос. На каждой из итераций 3, 4 и 5 парсер находит окончание на вершине стека.

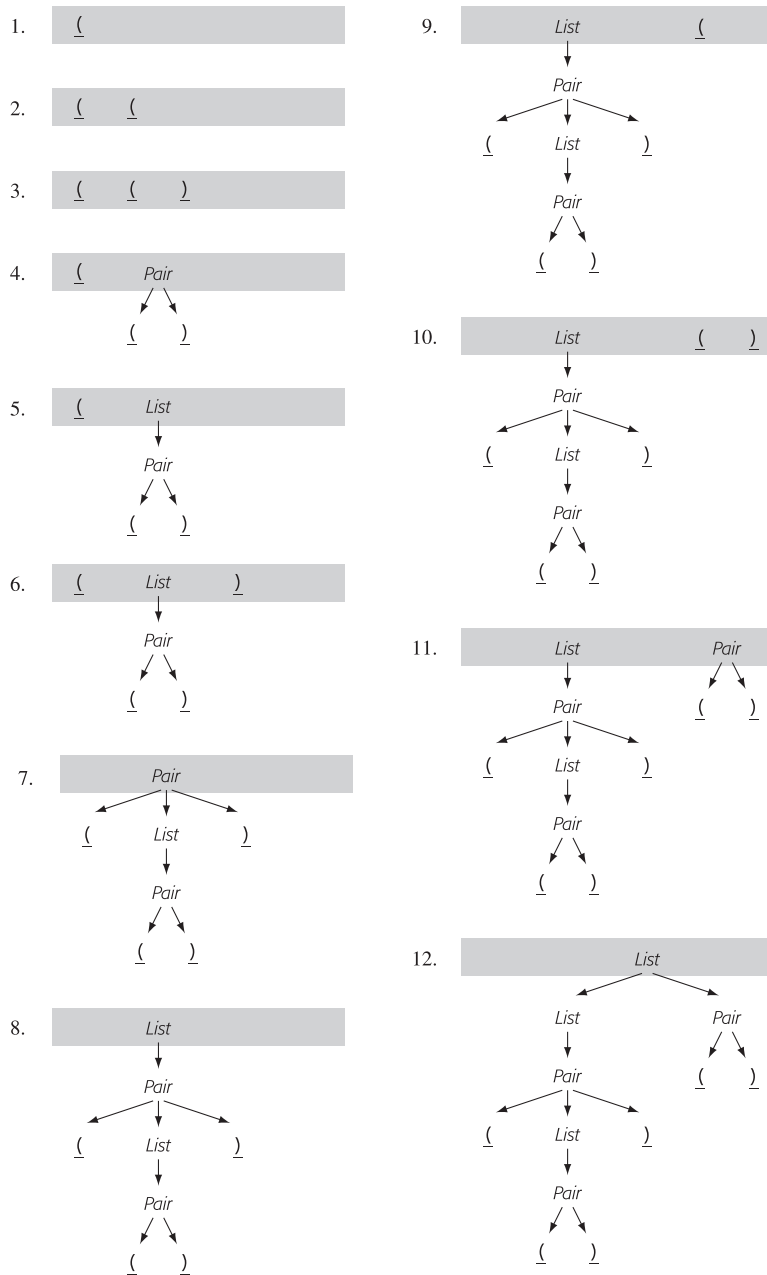


Рис. 3.20 Последовательность частичных деревьев разбора, построенных для строки (( )) ( )

Итерация 3 (состояние 8): ( ) на вершине стека образует окончание для правила 5.

Action[8,`eof`] = *Action*[8,`(`] = *reduce 5*.

Итерация 4 (состояние 2): *Pair* на вершине стека образует окончание для правила 3.

Action[2,`eof`] = *Action*[2,(] = *reduce 3*.

Итерация 5 (состояние 1): *List* на вершине стека образует окончание для правила 1, если опережающим символом является `eof`. *Action*[1,`eof`] = *accept*. При других опережающих символах *List* не является окончанием.

В этих двух примерах парсер распознал ( ) как окончание трижды. Поведение во всех трех случаях различно и зависит от предшествующего левого контекста, закодированного в состоянии под ( ) в стеке. Сравнение этих трех ситуаций показывает, как хранящиеся в стеке состояния кодируют левый контекст и управляют будущим ходом разбора.

В первом примере на рис. 3.18 парсер находился в состоянии 8, когда нашел окончание ( ) с опережающим символом `eof`. Под окончанием было состояние 0 и *Goto*[0,*Pair*] = 2. В состоянии 2 опережающий символ `eof` привел к свертке по правилу 3, за которым последовала приемка.

Если бы вместо `eof` опережающим символом в состоянии 2 был), то парсер сообщил бы об ошибке. Опережающий символ (привел бы к свертке по правилу 3, после чего парсер был бы готов к следующим вложенным скобкам.

Во втором примере на рис. 3.19 парсер находит окончание для ( ) дважды. Первое окончание встречается на итерации 4. Парсер находится в состоянии 12, а опережающий символ равен). Перед этим он перенес в стек (, (и). *Action*[12,)] = *reduce 5*. Под ( ) находится состояние 3, и *Goto*[3,*Pair*] = 6, состоянию, в котором дополнительные) допустимы. Окончание ( ) встречается также на итерации 11. Здесь под ( ) в стеке находится состояние 1, и *Goto*[1,*Pair*] = 4. В состоянии 4 опережающий символ `eof` или (вызывает свертку *List Pair* в *List*, тогда как опережающий символ) приводит к ошибке.

Таблицы *Action* и *Goto* в совокупности со стеком заставляют парсер отслеживать предшествующий левый контекст и позволяют ему выполнять различные действия в зависимости от него. В свою очередь, контекст позволил парсеру правильно обработать все три случая, в которых он обнаружил окончание для ( ). Мы вернемся к этому вопросу, когда будем обсуждать построение таблиц *Action* и *Goto*.

Разбор ошибочной входной строки

Чтобы понять, как LR(1)-парсер обнаруживает синтаксическую ошибку, рассмотрим последовательность действий, выполняемую им для строки «( ))».

Итерация	Состояние	Слово	Стек	Окончание	Действие
0	–	(	$ ⟨*Goal* 0⟩	*– нет –*	–
1	0	(	$ ⟨*Goal* 0⟩	*– нет –*	*shift 3*
2	3	)	$ ⟨*Goal* 0⟩ ⟨(3⟩	*– нет –*	*shift 8*
3	8	)	$ ⟨*Goal* 0⟩ ⟨(3⟩ ⟨) 8⟩	*– нет –*	*error*

Начальное состояние и первые две итерации разбора такие же, как в первом примере, «( )». Парсер переносит (и). На третьей итерации цикла while парсер читает *Action*[8,)] и находит недопустимый элемент, что заставляет его сообщить об ошибке.

Локализация ошибки – особое достоинство LR(1)-парсеров.

LR(1)-парсер обнаруживает синтаксические ошибки с помощью простого механизма: находит недопустимый элемент таблицы. Он обнаруживает ошибку так рано, как только возможно, еще до того, как прочитал слово, следующее за теми, которые необходимы для доказательства ошибочности ввода. Это свойство локализует ошибку в конкретной точке входа. Используя доступный контекст и знание грамматики, LR(1)-парсер может выдавать хорошие диагностические сообщения.

Использование LR-парсеров

Ключ к LR-разбору лежит в построении таблиц *Action* и *Goto*. В таблицах закодированы все допустимые последовательности свертки, которые могут возникнуть при обратном правостороннем выводе для заданной грамматики. Хотя число таких последовательностей огромно, сама грамматика ограничивает возможный порядок сверток.

Автор компилятора может построить таблицы *Action* и *Goto* вручную. Однако алгоритм их построения требует скрупулезного внимания; это прекрасный пример задачи, которую можно автоматизировать и поручить компьютеру. И программы для ее автоматизации широкодоступны. В следующем разделе будет описан один алгоритм, позволяющий построить таблицы LR(1)-разбора.

При наличии генератора LR(1)-парсеров роль автора компилятора сводится к определению грамматики и проверке того, что эта грамматика обладает свойством LR(1). На практике генератор парсеров идентифицирует продукции, которые привносят неоднозначность или нуждаются в заглядывании более чем на один символ вперед, чтобы отличить перенос от свертки. При изучении алгоритма построения таблиц мы увидим, как возникают такие проблемы, как с ними бороться и как понять диагностические сообщения, выдаваемые генераторами LR(1)-парсеров.

Заглядывание вперед на большее число символов

Идеи, лежащие в основе LR(1)-парсеров, на самом деле определяют семейство парсеров, различающихся количеством опережающих символов. LR(k)-парсер использует не более k опережающих символов. Дополнительный опережающий символ позволяет LR(2)-парсеру распознавать более широкое множество грамматик, чем способен LR(1)-парсер. Как это ни парадоксально, добавление опережающего символа не увеличивает множество языков, которые такой парсер способен распознать. LR(1)-парсеры допускают ровно то же множество языков, что и LR(k)-парсеры для $k > 1$. Но LR(1)-грамматика языка может быть более сложной, чем LR(2)- или LR(3)-грамматика.

3.4.2. Построение таблиц LR(1)-разбора

Для построения таблиц *Action* и *Goto* для грамматики генератор LR(1)-парсеров строит модель распознающего окончания автомата и использует ее для заполнения таблиц. Модель, называемая *каноническим набором множеств LR(1)-элементов*, представляет все возможные состояния парсера, а также переходы между этими состояниями. Процесс напоминает построение подмножеств из раздела 2.4.3.

Для иллюстрации алгоритма построения таблиц мы будем использовать два примера. Первый – скобочная грамматика на рис. 3.17(а), повторенная во врезке. Она достаточно мала, чтобы послужить примером, но при этом достаточно велика, чтобы высветить некоторые сложности, возникающие в процессе. Второй пример – в разделе 3.4.3 – абстрагированная версия классической неоднозначности `if-then-else`. Для этой грамматики построение завершается неудачно, потому что она неоднозначна. Пример наглядно демонстрирует ситуации, которые приводят к ошибкам в процессе построения таблиц.

1	*Goal*	→	*List*
2	*List*	→	*List Pair*
3		|	*Pair*
4	*Pair*	→	(*List*)
5		|	( )

Скобочная грамматика

LR(1)-элементы

LR(1)-элемент
$[A \rightarrow \beta \bullet \gamma, \mathtt{a}]$, где $A \rightarrow \beta\gamma$ – продукция грамматики, • представляет позицию вершины стека парсера, а `a` – терминальный символ грамматики.

В LR(1)-парсере таблицы *Action* и *Goto* кодируют информацию об окончаниях и потенциальных окончаниях на каждом шаге разбора. В алгоритме построения таблиц LR(1)-элементы используются для представления окончаний, потенциальных окончаний и ассоциированных опережающих символов. LR(1)-элемент $[A \rightarrow \beta \bullet \gamma, \mathtt{a}]$ состоит из продукции $A \rightarrow \beta\gamma$, маркера •, обозначающего позицию вершины стека в правой части продукции, и конкретного терминального символа `a`, выступающего в роли опережающего.

Алгоритм построения таблиц строит канонический набор множеств LR(1)-элементов, $\mathcal{CC} = \{CC_0, CC_1, CC_2, \ldots, CC_n\}$. Каждое $CC_i \in \mathcal{CC}$ представляет допустимую конфигурацию LR(1)-парсера, или состояние парсера. Оно содержит один или несколько LR(1)-элементов, представляющих окончания или потенциальные окончания, которые соответствуют состоянию парсера. Прежде чем приступать к построению таблиц, необходимо более подробно объяснить смысл LR(1)-элементов.

Для продукции $A \rightarrow \beta\gamma$ и опережающего символа `a` маркер может генерировать три разных элемента, каждый со своей интерпретацией. В каждом случае присутствие элемента в некотором множестве CC_i из канонического набора означает, что вход, который парсер видел, согласован с вхождением в грамматику A, за которым следует `a`. Позиция • в элементе различает три случая.

1. [$A \rightarrow \bullet\beta\gamma$, a] означает, что A мог бы быть допустим и что распознавание β далее стало бы одним шагом в направлении обнаружения A. Мы называем такой элемент *возможностью*; он представляет возможное пополнение уже виденного входа.
2. [$A \rightarrow \beta \bullet \gamma$, a] означает, что парсер вышел из состояния [$A \rightarrow \bullet\beta\gamma$, a], распознав β. Следующим допустимым шагом будет распознавание γ. Мы называем такой элемент *частично полным*.
3. [$A \rightarrow \beta\gamma \bullet$, a] означает, что парсер нашел $\beta\gamma$ в контексте, где символ A, за которым следует a, был бы допустим. Если опережающий символ a, то элемент является окончанием, и парсер может свернуть $\beta\gamma$ в A. Такой элемент мы называем *полным*.

В LR(1)-элементе символы слева от маркера • представляют контекст из уже распознанных частей продукции – левый контекст. (Состояния в стеке кодируют сводку этого левого контекста – по сути дела, историю разбора до текущей точки.) Опережающий символ дает правый контекст – один символ правого контекста. Если парсер обнаруживает, что находится в состоянии, которое включает [$A \rightarrow \beta\gamma \bullet$, a], и при этом опережающий символ равен a, значит, он нашел окончание и должен свернуть $\beta\gamma$ в A.

На рис. 3.21 показан полный набор LR(1)-элементов, генерируемых скобочной грамматикой. Первые два элемента в левом верхнем углу заслуживают особого внимания. Элемент [*Goal* → • *List*, eof] представляет начальное состояние парсера – поиск строки, которая сворачивается в *Goal* и за которой следует eof. Любой разбор начинается в этом состоянии. Второй элемент, [*Goal* → *List* •, eof], представляет желаемое конечное состояние парсера – найдена строка, которая сворачивается в *Goal* и за которой следует eof. Этот элемент представляет любой успешный разбор. Все возможные разборы получаются путем сцепления состояний парсера способом, направляемым грамматикой, начиная с состояния, содержащего [*Goal* → • *List*, eof], и заканчивая состоянием, содержащим [*Goal* → *List* •, eof].

[*Goal* → • *List*, eof]	[*List* → • *List Pair*, eof]	[*Pair* → • (*List*), eof]	[*Pair* → • ( ), eof]
[*Goal* → *List* •, eof]	[*List* → *List* • *Pair*, eof]	[*Pair* → (• *List*), eof]	[*Pair* → (•), eof]
	[*List* → *List Pair* •, eof]	[*Pair* → (*List* •), eof]	[*Pair* → ( ) •, eof]
[*List* → • *Pair*, eof]	[*List* → • *List Pair*, ( ]	[*Pair* → (*List*) •, eof]	[*Pair* → • ( ), ( ]
[*List* → *Pair* •, eof]	[*List* → *List* • *Pair*, ( ]	[*Pair* → • (*List*), ( ]	[*Pair* → (•), ( ]
[*List* → • *Pair*, ( ]	[*List* → *List Pair* •, ( ]	[*Pair* → (• *List*), ( ]	[*Pair* → ( ) •, ( ]
[*List* → *Pair* •, ( ]	[*List* → • *List Pair*,)]	[*Pair* → (*List* •), ( ]	[*Pair* → • ( ),)]
[*List* → • *Pair*,)]	[*List* → *List* • *Pair*,)]	[*Pair* → (*List*) •, ( ]	[*Pair* → (•),)]
[*List* → *Pair* •,)]	[*List* → *List Pair* •,)]	[*Pair* → • (*List*),)]	[*Pair* → ( ) •,)]
		[*Pair* → (• *List*),)]	
		[*Pair* → (*List* •),)]	
		[*Pair* → (*List*) •,)]	

Рис. 3.21 LR(1)-элементы для скобочной грамматики

Построение канонического набора

Чтобы построить канонический набор множеств LR(1)-элементов, $\mathcal{CC}$, генератор парсеров отправляется от начального состояния, [*Goal* → • *List*, `eof`], и строит модель всех потенциально возможных переходов. Алгоритм представляет каждую возможную конфигурацию, или состояние, парсера в виде множества LR(1)-элементов. К этим множествам алгоритм применяет две операции: взятие замыкания и вычисление перехода.

- Функция *closure* пополняет состояние; если дано некоторое базовое множество элементов, то она добавляет в него все вытекающие из него LR(1)-элементы. Например, в любом месте, где продукция *Goal* → *List* допустима, продукции, выводимые из *List*, также допустимы. Таким образом, элемент [*Goal* → • *List*, `eof`] влечет за собой как [*List* → • *List Pair*, `eof`], так и [*List* → • *Pair*, `eof`]. *Closure* находит все такие элементы и добавляет их в состояние.
- Функция *goto* моделирует эффект перехода из некоторого состояния *s* по символу грамматики *x*. Для этого алгоритм исследует множество LR(1)-элементов в *s*. Он находит каждый элемент, где • предшествует *x*, перемещает • после *x* и помещает новый элемент в множество, которое *goto* возвращает в качестве результата. Наконец, он использует замыкание, чтобы пополнить новое состояние.

Чтобы начальный символ было легко найти, мы потребуем, чтобы в грамматике был единственный начальный символ, не встречающийся в правых частях продукций. В скобочной грамматике это символ *Goal*.

Элемент [*Goal* → • *List*, `eof`] представляет начальное состояние скобочной грамматики; любой допустимый разбор распознает *Goal*, за которым следует `eof`. Этот элемент образует базу CC_0, первого состояния в $\mathcal{CC}$.

Если в грамматике имеется несколько продукций для начального символа, то каждая из них добавляет элемент в базу CC_0.

Вычисление замыкания

Чтобы вычислить полное начальное состояние парсера, CC_0, по его базе, алгоритм должен добавить в базу все элементы, вытекающие из элементов в базе. На рис. 3.22(а) показан алгоритм *closure*, применяемый для этого вычисления. Он перебирает все элементы множества *s*. Если маркер • в некотором элементе непосредственно предшествует какому-то нетерминальному символу *C*, то *closure* должен добавить один или несколько элементов для каждой продукции, выводимой из *C*. Замыкание помещает • в начальную позицию каждого элемента, построенного таким способом.

Смысл функции *closure* понятен. Если $[A \rightarrow \beta \bullet C\ \delta, \mathtt{a}] \in s$ и парсер распознает *C*, за которым следует $\delta\mathtt{a}$, то он должен свернуть $\beta C\delta$ в *A*. Замыкание добавляет элемент в *s* для каждой такой возможности.

Здесь β и δ представляют возможные символы до или после *C*. Либо β, либо δ может быть пустым.

```
closure(s)
  while (s продолжает изменяться) do
    for each элемент [A → β • C δ, a] ∈ s do
      lookahead ← δa
      for each продукции C → γ ∈ P do
        for each b ∈ FIRST(lookahead) do
          s ← s ∪ {[C → • γ, b]}
  return s
```

(a) Функция closure

```
goto(s, x)
  t ← ∅
  for each элемент i ∈ s do
    if i является [α → β • x δ, a] then
      t ← t ∪ {[α → β x • δ, a]}
  return closure(t)
```

(b) Функция goto

Рис. 3.22 Вспомогательные функции для построения таблиц LR(1)

По нашему опыту, именно это использование FIRST(δa) является местом в построении таблицы LR(1), где человек чаще всего допускает ошибки.

Для некоторой продукции $C \rightarrow \gamma$ алгоритм *closure* строит элемент для каждого терминального символа в FIRST(δa). Он вставляет маркер • перед γ и добавляет подходящий опережающий символ.

Для скобочной грамматики это построение создает CC_0 путем построения множества, содержащего начальный элемент, { [*Goal* → • *List*, `eof`] }, и вычисления его замыкания:

$$CC_0 = \left\{ \begin{array}{l} [Goal \rightarrow \bullet\ List, \texttt{eof}], [List \rightarrow \bullet\ List\ Pair, \texttt{eof}], \\ [List \rightarrow \bullet\ List\ Pair, \texttt{(}], [List \rightarrow \bullet\ Pair, \texttt{eof}], \\ [List \rightarrow \bullet\ Pair, \texttt{(}], [Pair \rightarrow \bullet\ \texttt{(}\ List\ \texttt{)}, \texttt{eof}], \\ [Pair \rightarrow \bullet\ \texttt{(}\ List\ \texttt{)}, \texttt{(}], [Pair \rightarrow \bullet\ \texttt{(}\ \texttt{)}, \texttt{eof}], \\ [Pair \rightarrow \bullet\ \texttt{(}\ \texttt{)}, \texttt{(}] \end{array} \right\}.$$

Closure – еще один пример вычисления неподвижной точки. В трижды вложенном цикле в множество *s* либо добавляются элементы, либо оно остается неизменным. Удаления элементов из *s* не происходит никогда. Так как множество LR(1)-элементов конечно, этот цикл должен остановиться. Тройная вложенность выглядит дорогостоящей. Однако при ближайшем рассмотрении выясняется, что каждый элемент *s* должен быть обработан только один раз. Опираясь на этот факт, можно написать версию алгоритма с рабочим списком.

Вычисление Goto

Вторая ключевая операция построения – функция *goto*. Она принимает множество, которое моделирует состояние парсера, одно из $CC_i \in \mathcal{CC}$, и символ грамматики *x*. Зная CC_i и *x*, она вычисляет модель состояния парсера, которое имело бы место, если бы в состоянии CC_i был распознан символ *x*.

Функция *goto*, показанная на рис. 3.22(b), принимает множество LR(1)-элементов s и символ грамматики *x* и возвращает новое множество LR(1)-элементов. Она перебирает элементы, принадлежащие *s*. Обнаружив элемент, в котором • непосредственно предшествует *x*, она создает новый элемент, перемещая • вправо за *x*. Этот новый элемент представляет конфигурацию парсера после распознавания *x*. *Goto* помещает эти новые элементы в новое

множество, вычисляет его замыкание, чтобы пополнить состояние парсера, и возвращает это новое состояние.

По множеству CC_0, показанному выше, это построение может вывести состояние, которого парсер достигнет после начальной (путем вычисления *goto*(CC_0, ( ). Внутренний цикл в *goto* находит четыре элемента, содержащих «• (». По каждому из них *goto* создает новый элемент, в котором • перемещен за (. Эти элементы образуют *базу* нового множества – первые четыре его элемента. Затем *goto* вызывает *closure*, чтобы пополнить множество. В построении это множество становится CC_3.

$$CC_3 = \left\{ \begin{array}{l} [Pair \rightarrow (\bullet List), \texttt{eof}] \; [Pair \rightarrow (\bullet List), (] \\ [Pair \rightarrow (\bullet), \texttt{eof}] \; [Pair \rightarrow (\bullet), (] \\ [List \rightarrow \bullet List\ Pair, (] \; [List \rightarrow \bullet List\ Pair,)] \\ [List \rightarrow \bullet Pair, (] \; [List \rightarrow \bullet Pair,)] \; [Pair \rightarrow \bullet (List), (] \\ [Pair \rightarrow \bullet (List),)] \; [Pair \rightarrow \bullet ( ), (] \; [Pair \rightarrow \bullet ( ),)] \end{array} \right\}.$$

Goto создала первые четыре элемента напрямую. *Closure* добавила остальные.

Алгоритм

Для построения канонического набора множеств LR(1)-элементов, $\mathcal{CC}$, алгоритм вычисляет начальное множество CC_0, а затем систематически ищет множества, представляющие все возможные состояния парсера, достижимые из CC_0. Он повторяет этот процесс для каждого нового множества, пока не построит множество всех достижимых состояний парсера. На рис. 3.23 показан этот алгоритм.

```
CC0 ← ∅
for each продукции вида Goal → α do
  CC0 ← CC0 ∪ { [ Goal → • α, eof] }
CC0 ← closure( CC0 )
𝒞𝒞 ← { CC0 }

while (новые множества продолжают добавляться в 𝒞𝒞) do
  for each непомеченного множества CCi ∈ 𝒞𝒞 do
    поместить CCi как обработанное
    for each x, следующего за • в элементе, принадлежащем CCi, do
      temp ← goto(CCi, x)
      if temp ∉ 𝒞𝒞 then
        𝒞𝒞 ← 𝒞𝒞 ∪ {temp}
      зарегистрировать переход из CCi в temp по x
```

Рис. 3.23 Алгоритм построения канонического набора множеств LR(1)-элементов

Чтобы вычислить начальное состояние, алгоритм строит LR(1)-элементы, представляющие каждую продукцию с начальным символом в левой части. Он вычисляет CC_0 как замыкание объединения этих элементов.

Чтобы найти все состояния, достижимые из CC_0, алгоритм ищет все символы *x*, которые следуют за маркером в каком-нибудь элементе, принадлежащем CC_0. Для каждого такого *x* он вычисляет *goto*(CC_0, *x*). Если это множество еще не находится в $\mathcal{CC}$, то оно добавляется туда. Также регистрируется переход по *x* из CC_0 в *goto*(CC_0, *x*).

Версия алгоритма с рабочим списком достигает того же эффекта.

Чтобы построить полный канонический набор, алгоритм повторяет этот процесс для каждого следующего множества CC_i в наборе. Процесс напоминает построение подмножеств (см. раздел 2.6). Чтобы каждое множество CC_i гарантированно обрабатывалось только один раз, алгоритм применяет простую схему пометки. В момент создания множество не помечено, а помечается, когда обрабатывается. Это значительно уменьшает количество вызовов *goto* и *closure*.

Это построение является вычислением неподвижной точки. Канонический набор $\mathcal{CC}$ является подмножеством степенного множества LR(1)-элементов. В цикле while размер $\mathcal{CC}$ монотонно возрастает; новые множества добавляются в $\mathcal{CC}$, но оттуда ничего не удаляется. Если размер множества LR(1)-элементов равен *n*, то размер $\mathcal{CC}$ никогда не станет больше 2^n. Поскольку 2^n конечно, вычисление должно остановиться.

Эта верхняя граница размера канонического набора далеко не точна. Например, в скобочной грамматике имеется 38 LR(1)-элементов, но она порождает набор $\mathcal{CC}$, содержащий всего 14 множеств, хотя верхняя граница 2^{38} гораздо больше. Для более сложных грамматик размер $\mathcal{CC}$ вызывает опасения, прежде всего потому что таблицы *Action* и *Goto* растут вместе с $|\mathcal{CC}|$. Как описано в разделе 3.6, и автор компилятора, и автор генератора парсеров могут принять меры для уменьшения размера этих таблиц.

Построение канонического набора

В качестве первого полного примера рассмотрим задачу о построении $\mathcal{CC}$ для скобочной грамматики. На рис. 3.24 показаны шаги, выполняемые алгоритмом, а на рис. 3.25 – все созданные им множества.

Сначала алгоритм вычисляет *closure*([*Goal* → • *List*, `eof`]), которое становится множеством CC_0. Затем он начинает строить возможные конфигурации парсера. Из итерации 1 на рис. 3.24 мы видим, что *goto*(CC_0, *List*) создает CC_1, *goto*(CC_0, *Pair*) создает CC_2, а *goto*(CC_0, (̲) создает CC_3. Прочие символы грамматики, *Goal* и)̲, порождают пустое множество. В каждом множестве CC_i на рис. 3.25 первая строка содержит базовые элементы, а последующие – элементы, добавленные функцией *closure*.

Итерация	Множество	*Goal*	*List*	*Pair*	(	)
1	CC_0	∅	CC_1	CC_2	CC_3	∅
2	CC_1	∅	∅	CC_4	CC_3	∅
	CC_2	∅	∅	∅	∅	∅
	CC_3	∅	CC_5	CC_6	CC_7	CC_8
3	CC_4	∅	∅	∅	∅	∅
	CC_5	∅	∅	CC_9	CC_7	CC_{10}
	CC_6	∅	CC_{11}	∅	∅	∅
	CC_7	∅	∅	CC_6	CC_7	CC_{12}
	CC_8	∅	∅	∅	∅	∅
4	CC_9	∅	∅	∅	∅	∅
	CC_{10}	∅	∅	∅	∅	∅
	CC_{11}	∅	∅	CC_9	CC_7	CC_{13}
	CC_{12}	∅	∅	∅	∅	∅
5	CC_{13}	∅	∅	∅	∅	∅

Рис. 3.24 Трасса построения LR(1) для скобочной грамматики

$$CC_0 = \left\{ \begin{array}{l} [Goal \to \bullet\ List, \texttt{eof}] \\ [List \to \bullet\ List\ Pair, \texttt{eof}]\ [List \to \bullet\ List\ Pair, \underline{(}]\ [List \to \bullet\ Pair, \texttt{eof}]\ [List \to \bullet\ Pair, \underline{(}] \\ [Pair \to \bullet\ \underline{(}\ List\ \underline{)}, \texttt{eof}]\ [Pair \to \bullet\ \underline{(}\ List\ \underline{)}, \underline{(}]\ [Pair \to \bullet\ \underline{(}\ \underline{)}, \texttt{eof}]\ [Pair \to \bullet\ \underline{(}\ \underline{)}, \underline{(}] \end{array} \right\}$$

$$CC_1 = \left\{ \begin{array}{l} [Goal \to List \bullet, \texttt{eof}]\ [List \to List \bullet\ Pair, \texttt{eof}]\ [List \to List \bullet\ Pair, \underline{(}] \\ [Pair \to \bullet\ \underline{(}\ List\ \underline{)}, \texttt{eof}]\ [Pair \to \bullet\ \underline{(}\ List\ \underline{)}, \underline{(}]\ [Pair \to \bullet\ \underline{(}\ \underline{)}, \texttt{eof}]\ [Pair \to \bullet\ \underline{(}\ \underline{)}, \underline{(}] \end{array} \right\}$$

$$CC_2 = [List \to Pair \bullet, \texttt{eof}]\ [List \to Pair \bullet, \underline{(}]$$

$$CC_3 = \left\{ \begin{array}{l} [Pair \to \underline{(}\ \bullet\ List\ \underline{)}, \texttt{eof}]\ [Pair \to \underline{(}\ \bullet\ List\ \underline{)}, \underline{(}]\ [Pair \to \underline{(}\ \bullet\ \underline{)}, \texttt{eof}]\ [Pair \to \underline{(}\ \bullet\ \underline{)}, \underline{(}] \\ [List \to \bullet\ List\ Pair, \underline{(}]\ [List \to \bullet\ List\ Pair, \underline{)}]\ [List \to \bullet\ Pair, \underline{(}]\ [List \to \bullet\ Pair, \underline{)}] \\ [Pair \to \bullet\ \underline{(}\ List\ \underline{)}, \underline{(}]\ [Pair \to \bullet\ \underline{(}\ List\ \underline{)}, \underline{)}]\ [Pair \to \bullet\ \underline{(}\ \underline{)}, \underline{(}]\ [Pair \to \bullet\ \underline{(}\ \underline{)}, \underline{)}] \end{array} \right\}$$

$$CC_4 = [List \to List\ Pair \bullet, \texttt{eof}]\ [List \to List\ Pair \bullet, \underline{(}]$$

$$CC_5 = \left\{ \begin{array}{l} [List \to List \bullet\ Pair, \underline{(}]\ [List \to List \bullet\ Pair, \underline{)}]\ [Pair \to \underline{(}\ List \bullet\ \underline{)}, \texttt{eof}]\ [Pair \to \underline{(}\ List \bullet\ \underline{)}, \underline{(}] \\ [Pair \to \bullet\ \underline{(}\ List\ \underline{)}, \underline{(}]\ [Pair \to \bullet\ \underline{(}\ List\ \underline{)}, \underline{)}]\ [Pair \to \bullet\ \underline{(}\ \underline{)}, \underline{(}]\ [Pair \to \bullet\ \underline{(}\ \underline{)}, \underline{)}] \end{array} \right\}$$

$$CC_6 = [List \to Pair \bullet, \underline{(}]\ [List \to Pair \bullet, \underline{)}]$$

$$CC_7 = \left\{ \begin{array}{l} [Pair \to \underline{(}\ \bullet\ List\ \underline{)}, \underline{(}]\ [Pair \to \underline{(}\ \bullet\ List\ \underline{)}, \underline{)}]\ [Pair \to \underline{(}\ \bullet\ \underline{)}, \underline{(}]\ [Pair \to \underline{(}\ \bullet\ \underline{)}, \underline{)}] \\ [List \to \bullet\ List\ Pair, \underline{(}]\ [List \to \bullet\ List\ Pair, \underline{)}]\ [List \to \bullet\ Pair, \underline{(}]\ [List \to \bullet\ Pair, \underline{)}] \\ [Pair \to \bullet\ \underline{(}\ List\ \underline{)}, \underline{(}]\ [Pair \to \bullet\ \underline{(}\ List\ \underline{)}, \underline{)}]\ [Pair \to \bullet\ \underline{(}\ \underline{)}, \underline{(}]\ [Pair \to \bullet\ \underline{(}\ \underline{)}, \underline{)}] \end{array} \right\}$$

$$CC_8 = [Pair \to \underline{(}\ \underline{)} \bullet, \texttt{eof}]\ [Pair \to \underline{(}\ \underline{)} \bullet, \underline{(}]$$

$$CC_9 = [List \to List\ Pair \bullet, \underline{(}]\ [List \to List\ Pair \bullet, \underline{)}]$$

$$CC_{10} = [Pair \to \underline{(}\ List\ \underline{)} \bullet, \texttt{eof}]\ [Pair \to \underline{(}\ List\ \underline{)} \bullet, \underline{(}]$$

$$CC_{11} = \left\{ \begin{array}{l} [List \to List \bullet\ Pair, \underline{(}]\ [List \to List \bullet\ Pair, \underline{)}]\ [Pair \to \underline{(}\ List \bullet\ \underline{)}, \underline{(}]\ [Pair \to \underline{(}\ List \bullet\ \underline{)}, \underline{)}] \\ [Pair \to \bullet\ \underline{(}\ List\ \underline{)}, \underline{(}]\ [Pair \to \bullet\ \underline{(}\ List\ \underline{)}, \underline{)}]\ [Pair \to \bullet\ \underline{(}\ \underline{)}, \underline{(}]\ [Pair \to \bullet\ \underline{(}\ \underline{)}, \underline{)}] \end{array} \right\}$$

$$CC_{12} = [Pair \to \underline{(}\ \underline{)} \bullet, \underline{(}]\ [Pair \to \underline{(}\ \underline{)} \bullet, \underline{)}]$$

$$CC_{13} = [Pair \to \underline{(}\ List\ \underline{)} \bullet, \underline{(}]\ [Pair \to \underline{(}\ List\ \underline{)} \bullet, \underline{)}]$$

Рис. 3.25 Канонический набор множеств LR(1)-элементов для скобочной грамматики

Классы контекстно-свободных грамматик



Мы можем ввести на множестве контекстно-свободных грамматик иерархию, основанную на трудности разбора. В этой главе выделено четыре вида грамматик: произвольные КСГ, LR(1)-грамматики, LL(1)-грамматики и регулярные грамматики (РГ). Эти множества вложены друг в друга, как показано на рисунке справа.

Разбор произвольных КСГ занимает больше времени, чем разбор ограниченных грамматик типа LR(1) или LL(1). Например, алгоритм Эрли разбирает произвольные КСГ, но в худшем случае время его работы составляет $O(n^3)$, где n – число слов во входной строке. Исторически авторы компиляторов считали алгоритм Эрли слишком дорогим для практического применения.

Множество LR(1)-грамматик включает большое подмножество однозначных КСГ. LR(1)-грамматику можно разобрать снизу вверх с помощью линейного сканирования слева направо с заглядыванием вперед на один символ. Инструменты для построения LR(1)-парсеров широкодоступны.

Множество LL(1)-грамматик является важным подмножеством LR(1)-грамматик. LL(1)-грамматики можно разобрать сверху вниз с помощью линейного сканирования слева направо с заглядыванием вперед на один символ. Существует много инструментов для построения LL(1)-парсеров; эти грамматики также подходят для написания парсеров рекурсивного спуска вручную. Многие языки программирования можно описать LL(1)-грамматикой.

Регулярные грамматики (РГ) – подмножество КСГ, в котором продукции могут иметь вид либо $A \rightarrow \mathtt{a}$, либо $A \rightarrow \mathtt{a}\ B$, где $A, B \in NT$ и $\mathtt{a} \in T$. Регулярные грамматики кодируют в точности те же языки, что регулярные выражения.

Почти все конструкции языков программирования можно записать в форме LR(1)- или LL(1)-грамматики. Поэтому в большинстве компиляторов используются парсеры, основанные на одном из этих двух классов КСГ. Некоторые конструкции попадают в промежуток между LR(1) и LL(1), но они не особенно полезны. (См. грамматику Уэйта в упражнении 3.10.)

Последующие итерации заполняют остальные множества в $\mathcal{CC}$, как показано на рис. 3.24 и 3.25. Построение останавливается, когда добавить новые множества в $\mathcal{CC}$ невозможно.

Заполнение таблиц

Имея канонический набор $\mathcal{CC}$ для грамматики, генератор парсеров может заполнить таблицы *Action* и *Goto*, обойдя все $CC_i \in \mathcal{CC}$. Каждое CC_i становится состоянием парсера. Его элементы диктуют, где должны находиться переносы и свертки в соответствующей строке таблицы *Action*. Переходы состояний в *Action* и *Goto* происходят из переходов, зарегистрированных при построении канонического набора, как показано на рис. 3.24.

```
for each CC_i ∈ 𝒞𝒞 do
  for each элемент I ∈ CC_i do
    if I является [ A → β • c γ, a] and goto(CC_i,c) = CC_j then
      Action[i, c] ← " shift j "
    else if I является [A → β •, a] then
      Action[i, a] ← " reduce A → β "
    else if I является [Goal → β •, eof] then
      Action[i, eof] ← " accept "
  for each n ∈ NT do
    if goto(CC_i,n) = CC_j then
      Goto[i,n] ← j
```

Рис. 3.26 Алгоритм заполнения таблиц LR(1)

Три вида элементов генерируют записи в таблице *Action*.

1. Элемент вида $[A \rightarrow \beta \bullet \texttt{c}\ \delta, \texttt{a}]$ означает, что поиск терминального символа `c` мог бы быть допустимым следующим шагом поиска нетерминального символа A. Он генерирует элемент *shift* в столбце, соответствующем `c`, в текущем состоянии. При этом находится следующее состояние распознавателя в трассе ниже CC_i и `c`.

   Как и раньше, β, δ или оба могут быть пустыми.

2. Элемент вида $[A \rightarrow \beta \bullet, \texttt{a}]$ означает, что парсер нашел β. Если опережающий символ равен `a`, то элемент является окончанием. Таким образом, генерируется *reduce* в столбце, соответствующем `a`, в текущем состоянии.
3. Элемент вида [*Goal* → β •, `eof`], где *Goal* – начальный символ, означает допускающее состояние. Парсер свернул входной поток в *Goal*, и опережающий символ равен `eof`. Этот элемент генерирует действие *accept* в столбце, соответствующем `eof`, в текущем состоянии.

Алгоритм на рис. 3.26 материализует эту идею. Для LR(1)-грамматики он должен единственным образом определить записи в таблицах *Action* и *Goto*.

Действия по заполнению таблицы могут быть интегрированы в построение $\mathcal{CC}$.

Чтобы заполнить таблицу *Goto*, алгоритму не нужно заново вычислять *goto*(CC_i, n). Вместо этого он может посмотреть переходы, которые запомнил в процессе построения $\mathcal{CC}$. Для нашего примера эта информация зарегистрирована в трассе на рис. 3.24.

Алгоритм заполнения таблицы игнорирует элементы, в которых • предшествует нетерминальному символу. Когда • предшествует терминальному символу, генерируются действия переноса. Действия свертки и приемки генерируются, когда • находится в правом конце продукции. Что, если CC_i содержит элемент $[A \rightarrow \beta \bullet \gamma\ \delta, \texttt{a}]$, где $\gamma \in NT$? Такой элемент не генерирует никаких записей в таблице, но его присутствие в множестве заставляет функцию *closure* включить элементы, которые генерируют записи. Когда *closure* находит •, который непосредственно предшествует нетерминальному символу

γ, она добавляет продукции с γ в левой части и •, предшествующим правой части. Этот процесс порождает FIRST(γ) в CC*i*. Функция *closure* находит все $x \in$ FIRST(γ) и добавляет в CC_i элементы, которые генерируют действия переноса для каждого такого x.

Таблицы, порожденные для скобочной грамматики, показаны на рис. 3.17(b) выше. Эти таблицы плюс заготовка LR(1)-парсера составляют работоспособный парсер скобочного языка.

На практике генератор LR(1)-парсеров должен порождать и другие таблицы, необходимые заготовке LR(1)-парсера. Например, сворачивая $A \rightarrow \beta$, заготовка парсера выталкивает $|\beta|$ пар из стека и помещает в стек A. Генератор таблиц должен создать структуры данных, которые отображают продукцию из записи о свертке в таблице *Action*, скажем $A \rightarrow \beta$, в $|\beta|$ и A. Для отладки и диагностики нужны еще кое-какие таблицы, например для отображения целого числа, представляющего символ грамматики, в его текстовое имя.

Еще раз о нахождении окончаний

Эффективность LR(1)-парсеров зависит от быстроты механизма нахождения окончаний, встроенного в таблицы *Action* и *Goto*. Канонический набор $\mathcal{CC}$ представляет ДКА нахождения окончаний для грамматики. На рис. 3.27 показан ДКА для примера скобочной грамматики.

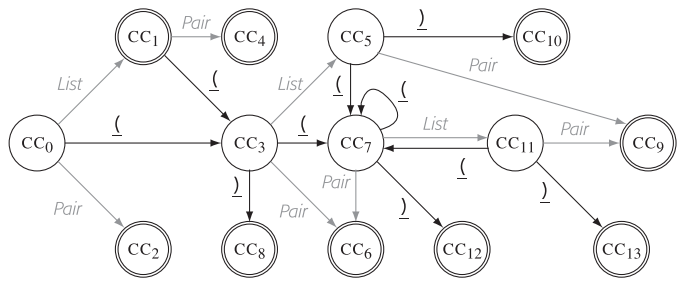


Рис. 3.27 Ищущий окончания ДКА для скобочной грамматики

LR(1)-парсер делает позицию окончания неявной – на вершине стека. Это решение существенно уменьшает число возможных окончаний.

Как LR(1)-парсер использует ДКА для нахождения окончаний, когда нам известно, что скобочный язык не регулярен? LR(1)-парсер полагается на простое наблюдение: *множество окончаний конечно*. Множество окончаний – это в точности множество полных LR(1)-элементов – тех, в которых маркер • находится в правом конце продукции элемента. Любой язык с конечным множеством предложений может быть распознан некоторым ДКА. Поскольку число продукций и число опережающих символов конечно, число полных элементов тоже конечно, и язык окончаний регулярен.

Во время выполнения LR(1)-парсер чередует два вида действий: перенос и свертку. Действия переноса моделируют шаги ДКА, ищущего окончания. Ког-

да парсер переносит каждое слово входного потока в стек разбора, он также изменяет состояние ДКА, как того требует синтаксическая категория слова. Действия свертки происходят, когда ДКА достигает конечного состояния. В этот момент заготовка парсера выталкивает окончание и его состояния ДКА из стека, открывая состояние, в котором ДКА находился до начала поиска текущего окончания. Это состояние находится в стеке под левой частью окончания.

Действие свертки использует нетерминальный символ в левой части окончания, чтобы совершить переход в ДКА. Оно находит этот переход в таблице *Goto*, используя открывшееся состояние и нетерминальный символ, помещает в стек новое состояние, а поверх него нетерминальный символ. По сути дела, заготовка LR(1)-парсера пользуется действиями свертки, чтобы смоделировать вызовы ищущего окончания ДКА для каждой новой подцели разбора.

Переходы по нетерминальным символам показаны серым цветом на рис. 3.27.

LR(1)-парсер чередует фазы поиска окончаний и свертки. Он переносит элементы в стек, пока не дойдет до конечного состояния ДКА. Затем он сворачивает окончание в нетерминальный символ в его левой части. Действия свертки связывают последовательные фазы поиска окончаний. Они используют левый контекст, закодированный в открывшемся состоянии, чтобы перезапустить ищущий окончания ДКА в состоянии, которое отражает только что распознанный нетерминальный символ. При разборе (( )) ( ), показанном на рис. 3.19, парсер помещает в стек для первой (не те же состояния, что для (, следующей за другой (. Эти состояния в стеке позволяют ему найти правильные окончания.

ДКА, ищущий окончания, закодирован непосредственно в таблицах *Action* и *Goto*. Каждое черное ребро соответствует действию переноса; *shift 3* в *Action*[1, (] порождает ребро из CC_1 в CC_3, помеченное (. Каждое серое ребро соответствует действию свертки. Если i – открывшееся состояние и *Goto*[i, x] = j, то ДКА содержит серый переход из CC_i в CC_j, помеченный нетерминальным символом x. Таким образом, ДКА на рис. 3.27 можно прочитать прямо по таблицам *Action* и *Goto* на рис. 3.17(b).

3.4.3. Ошибки при построении таблиц

В качестве второго примера построения таблиц LR(1)-разбора рассмотрим неоднозначную грамматику классической конструкции `if-then-else`, показанную во врезке. Чтобы размер *CC* не оказался слишком велик, мы заменили часть «`if` *Expr* `then`» в правилах 2 и 3 одним терминальным символом `ifthen`. Все остальные предложения абстрагированы одним терминальным символом `assign`. Получившаяся грамматика сохраняет фундаментальную неоднозначность оригинальной, продемонстрированную на стр. 114.

1	*Goal*	→	*Stmt*
2	*Stmt*	→	`ifthen` *Stmt*
3		|	`ifthen` *Stmt* `else` *Stmt*
4	*Pair*	→	`assign`

Упрощенная грамматика if-then-else

Построение канонического набора начинается с вычисления CC_0 как *closure*([*Goal* → • *Stmt*, `eof`]). На рис. 3.28 показана трасса всего построения. На рис. 3.29 представлен результирующий канонический набор множеств LR(1)-элементов. При ближайшем рассмотрении канонического набора для этой грамматики видно, как в алгоритме построения таблиц проявляются ошибки и как алгоритм заполнения таблиц их обнаруживает.

Итерация	Элемент	*Goal*	*Stmt*	`ifthen`	`else`	`assign`
1	CC_0	∅	CC_1	CC_2	∅	CC_3
2	CC_1	∅	∅	∅	∅	∅
	CC_2	∅	CC_4	CC_5	∅	CC_6
	CC_3	∅	∅	∅	∅	CC_8
3	CC_4	∅	∅	∅	CC_7	∅
	CC_5	∅	CC_8	CC_5	∅	CC_6
	CC_6	∅	∅	∅	∅	∅
	CC_8	∅	∅	CC_6	CC_7	CC_{12}
		∅	∅	∅	∅	∅
4	CC_7	∅	CC_9	CC_2	∅	CC_3
	CC_8	∅	∅	∅	CC_{10}	∅
5	CC_9	∅	∅	∅	∅	∅
	CC_{10}	∅	CC_{11}	CC_5	∅	CC_6
6	CC_{11}	∅	∅	∅	∅	∅

Рис. 3.28 Трасса построения LR(1)-разбора для грамматики `if-then-else`

Алгоритм заполнения таблицы работает нормально, пока не дойдет до CC_8:

$$CC_8 = \left\{ \begin{array}{ll} 1. & [\textit{Stmt} \rightarrow \texttt{ifthen}\ \textit{Stmt}\ \bullet, \texttt{eof}] \\ 2. & [\textit{Stmt} \rightarrow \texttt{ifthen}\ \textit{Stmt}\ \bullet, \texttt{else}] \\ 3. & [\textit{Stmt} \rightarrow \texttt{ifthen}\ \textit{Stmt}\ \bullet\ \texttt{else}\ \textit{Stmt}, \texttt{eof}] \\ 4. & [\textit{Stmt} \rightarrow \texttt{ifthen}\ \textit{Stmt}\ \bullet\ \texttt{else}\ \textit{Stmt}, \texttt{else}] \end{array} \right\}.$$

Каждый из этих элементов порождает запись в таблице *Action*. К сожалению, некоторые записи в *Action* определяются несколько раз. Так, в *Action*[8, `else`] записывается «*reduce 2*» элементом 2 и «*shift 10*» элементом 4. Аналогично в *Action*[8, `eof`] записывается «*reduce 2*» элементом 1 и «*shift 10*» элементом 3. Из-за этого построение таблицы завершается неудачно.

Многократная запись в одну и ту же ячейку таблицы – проявление неоднозначной грамматики. В ней существует два разных правосторонних вывода некоторого допустимого предложения, о чем свидетельствует тот факт, что в некоторых допустимых контекстах парсер может совершить либо перенос, либо свертку по одному и тому же опережающему символу.

Ошибка shift-reduce
Ошибка, которая возникает, когда алгоритм построения таблиц LR(1)-разбора пытается записать в одну и ту же ячейку таблицы *Action* и действие переноса, и действие свертки.

Этот конкретный вид конфликта называется *конфликтом shift-reduce*. Он возникает, потому что правило 2 грамматики является префиксом правила 3. Автор компилятора должен устранить неоднозначность грамматики, как показано в разделе 3.2.3.

$$CC_0 = \left\{ \begin{array}{l} [Goal \rightarrow \bullet\ Stmt, \texttt{eof}]\ [Stmt \rightarrow \bullet\ \texttt{ifthen}\ Stmt, \texttt{eof}] \\ [Stmt \rightarrow \bullet\ \texttt{ifthen}\ Stmt\ \texttt{else}\ Stmt, \texttt{eof}]\ [Stmt \rightarrow \bullet\ \texttt{assign}, \texttt{eof}] \end{array} \right\}$$

$$CC_1 = [Goal \rightarrow Stmt\ \bullet, \texttt{eof}]$$

$$CC_2 = \left\{ \begin{array}{l} [Stmt \rightarrow \bullet\ \texttt{ifthen}\ Stmt, \texttt{eof}]\ [Stmt \rightarrow \bullet\ \texttt{ifthen}\ Stmt, \texttt{else}]\ [Stmt \rightarrow \texttt{ifthen}\ \bullet\ Stmt, \texttt{eof}] \\ [Stmt \rightarrow \bullet\ \texttt{ifthen}\ Stmt\ \texttt{else}\ Stmt, \texttt{eof}]\ [Stmt \rightarrow \bullet\ \texttt{ifthen}\ Stmt\ \texttt{else}\ Stmt, \texttt{else}] \\ [Stmt \rightarrow \texttt{ifthen}\ \bullet\ Stmt\ \texttt{else}\ Stmt, \texttt{eof}]\ [Stmt \rightarrow \bullet\ \texttt{assign}, \texttt{eof}]\ [Stmt \rightarrow \bullet\ \texttt{assign}, \texttt{else}] \end{array} \right\}$$

$$CC_3 = [Stmt \rightarrow \texttt{assign}\ \bullet, \texttt{eof}]$$

$$CC_4 = [Stmt \rightarrow \texttt{ifthen}\ Stmt\ \bullet, \texttt{eof}]\ [Stmt \rightarrow \texttt{ifthen}\ Stmt\ \bullet\ \texttt{else}\ Stmt, \texttt{eof}]$$

$$CC_5 = \left\{ \begin{array}{l} [Stmt \rightarrow \bullet\ \texttt{ifthen}\ Stmt, \texttt{eof}]\ [Stmt \rightarrow \bullet\ \texttt{ifthen}\ Stmt, \texttt{else}]\ [Stmt \rightarrow \texttt{ifthen}\ \bullet\ Stmt, \texttt{eof}] \\ [Stmt \rightarrow \texttt{ifthen}\ \bullet\ Stmt, \texttt{else}]\ [Stmt \rightarrow \bullet\ \texttt{ifthen}\ Stmt\ \texttt{else}\ Stmt, \texttt{eof}] \\ [Stmt \rightarrow \bullet\ \texttt{ifthen}\ Stmt\ \texttt{else}\ Stmt, \texttt{else}]\ [Stmt \rightarrow \texttt{ifthen}\ \bullet\ Stmt\ \texttt{else}\ Stmt, \texttt{eof}] \\ [Stmt \rightarrow \texttt{ifthen}\ \bullet\ Stmt\ \texttt{else}\ Stmt, \texttt{else}]\ [Stmt \rightarrow \bullet\ \texttt{assign}, \texttt{eof}]\ [Stmt \rightarrow \bullet\ \texttt{assign}, \texttt{else}] \end{array} \right\}$$

$$CC_6 = [Stmt \rightarrow \texttt{assign}\ \bullet, \texttt{eof}]\ [Stmt \rightarrow \texttt{assign}\ \bullet, \texttt{else}]$$

$$CC_7 = \left\{ \begin{array}{l} [Stmt \rightarrow \bullet\ \texttt{ifthen}\ Stmt, \texttt{eof}]\ [Stmt \rightarrow \bullet\ \texttt{ifthen}\ Stmt\ \texttt{else}\ Stmt, \texttt{eof}] \\ [Stmt \rightarrow \texttt{ifthen}\ Stmt\ \texttt{else}\ \bullet\ Stmt, \texttt{eof}]\ [Stmt \rightarrow \bullet\ \texttt{assign}, \texttt{eof}] \end{array} \right\}$$

$$CC_8 = \left\{ \begin{array}{l} [Stmt \rightarrow \texttt{ifthen}\ Stmt\ \bullet, \texttt{eof}]\ [Stmt \rightarrow \texttt{ifthen}\ Stmt\ \bullet, \texttt{else}] \\ [Stmt \rightarrow \texttt{ifthen}\ Stmt\ \bullet\ \texttt{else}\ Stmt, \texttt{eof}]\ [Stmt \rightarrow \texttt{ifthen}\ Stmt\ \bullet\ \texttt{else}\ Stmt, \texttt{else}] \end{array} \right\}$$

$$CC_9 = [Stmt \rightarrow \texttt{ifthen}\ Stmt\ \texttt{else}\ Stmt\ \bullet, \texttt{eof}]$$

$$CC_{10} = \left\{ \begin{array}{l} [Stmt \rightarrow \bullet\ \texttt{ifthen}\ Stmt, \texttt{eof}]\ [Stmt \rightarrow \bullet\ \texttt{ifthen}\ Stmt, \texttt{else}] \\ [Stmt \rightarrow \bullet\ \texttt{ifthen}\ Stmt\ \texttt{else}\ Stmt, \texttt{eof}]\ [Stmt \rightarrow \bullet\ \texttt{ifthen}\ Stmt\ \texttt{else}\ Stmt, \texttt{else}] \\ [Stmt \rightarrow \texttt{ifthen}\ Stmt\ \texttt{else}\ \bullet\ Stmt, \texttt{eof}]\ [Stmt \rightarrow \texttt{ifthen}\ Stmt\ \texttt{else}\ \bullet\ Stmt, \texttt{else}] \\ [Stmt \rightarrow \bullet\ \texttt{assign}, \texttt{eof}]\ [Stmt \rightarrow \bullet\ \texttt{assign}, \texttt{else}] \end{array} \right\}$$

$$CC_{11} = [Stmt \rightarrow \texttt{ifthen}\ Stmt\ \texttt{else}\ Stmt\ \bullet, \texttt{eof}]\ [Stmt \rightarrow \texttt{ifthen}\ Stmt\ \texttt{else}\ Stmt\ \bullet, \texttt{else}]$$

Рис. 3.29 Канонический набор множеств LR(1)-элементов для грамматики `if-then-else`

Когда генератор парсеров встречает такой конфликт, построение завершается неудачно. Генератор должен сообщить о проблеме пользователю. Трудно написать такое сообщение об ошибке, не ссылаясь на LR(1)-элементы. Поэтому большинство генераторов довольствуются простым сообщением типа «*ошибка shift–reduce*» и предоставляют пользователю (предположительно автору компилятора) узнать, какие конкретно LR(1)-элементы были вовлечены.

В качестве альтернативы генератор LR(1)-парсеров может просто разрешать конфликты shift-reduce в пользу действия переноса. Такой выбор заставляет парсер распознавать более длинную продукцию. В случае грамматики `if-then-else` это решение связывает каждое `else` с самым ближним к нему `ifthen` – именно то правило, которое поддерживает избавленная от неоднозначности грамматика в разделе 3.2.3.

Неоднозначная грамматика также может приводить к *конфликтам reduce-reduce*. Такой конфликт возникает, если грамматика содержит две продукции с одинаковой правой частью, $A \rightarrow \gamma\ \delta$ и $B \rightarrow \gamma\ \delta$. Если некоторое состояние CC_i

Ошибка reduce-reduce
Неоднозначность, из-за которой алгоритм записывает в одну и ту же ячейку таблицы *Action* две разные свертки.

содержит одновременно $[A \rightarrow \gamma\ \delta\ \bullet, \texttt{a}]$ и $[B \rightarrow \gamma\ \delta\ \bullet, \texttt{a}]$, то оно будет генерировать два конфликтующих действия свертки для *Action*[i, a]. И снова конфликт отражает фундаментальную неоднозначность исходной грамматики, которую автор компилятора должен устранить (см. раздел 3.5.3).

Вручную устанавливать, обладает ли грамматика свойством LR(1), трудоемко и чревато ошибками. С другой стороны, генераторы LR(1)-парсеров широкодоступны. Поэтому, чтобы определить, обладает ли заданная грамматика свойством LR(1), проще всего запустить для нее генератор. Если процесс завершится успешно, значит, свойство LR(1) наличествует.

ИТОГИ РАЗДЕЛА

LR(1)-парсеры широко используются в компиляторах, создаваемых как в коммерческих, так и в учебных целях. Эти парсеры допускают большой класс языков. Время их работы пропорционально размеру конструируемого вывода. Инструменты, генерирующие LR(1)-парсеры, существуют для многих языков.

Алгоритм построения таблиц LR(1)-разбора – элегантное применение теории к практике. Он систематически строит модель распознающего окончания ДКА и кодирует ее в виде таблиц *Action* и *Goto*. Построение требует очень внимательного отношения к деталям. Это как раз та задача, которую следует автоматизировать – генераторы парсеров лучше справляются с такими вычислениями, чем люди. Но все равно квалифицированный автор компиляторов должен понимать алгоритмы, потому что они дают представление о том, как работают парсеры, с какими ошибками может столкнуться генератор парсеров, как эти ошибки возникают и как их можно исправить.

ВОПРОСЫ

1. Покажите шаги, которые заготовка LR(1)-парсера, дополненная таблицами для скобочной грамматики, выполнит для входной строки «(( ) ( )) ( )».
2. Постройте канонический набор множеств LR(1)-элементов для грамматики *SheepNoise*, показанной во врезке. Используйте этот канонический набор для построения таблиц *Action* и *Goto*. Покажите действия получившегося парсера для входа «baa baa baa».

SheepNoise → baa *SheepNoise*
| baa

3.5. Практические вопросы

Несмотря на появление высококачественных генераторов парсеров, автор компилятора должен понимать многочисленные практические вопросы, если хочет построить качественный парсер. В этом разделе рассматривается несколько вопросов, которые возникают на практике и оказывают существенное влияние на удобство работы с компилятором.

3.5.1. Восстановление после ошибки

Программисты часто компилируют код, содержащий синтаксические ошибки. На самом деле компиляторы широко применяются как самый быстрый способ обнаружения таких ошибок. При таком использовании компилятор должен найти максимальное число ошибок за одну попытку компиляции кода. Поэтому в задачу автора компилятора входит написание такого парсера, который мог бы переходить из одного ошибочного состояния в другое, а в промежутке продолжать разбор.

Примеры парсеров, приведенные в этой главе, ведут себя при обнаружении синтаксической ошибки одинаково: сообщают о проблеме и останавливаются. Такое поведение позволяет компилятору не тратить лишнее время на неправильную программу. Однако же так компилятор может найти не более одной синтаксической ошибки за прогон, а стало быть, нахождение всех ошибок в коде станет долгим и мучительным процессом.

Парсер должен находить как можно больше ошибок при каждой компиляции. Ему необходим механизм «восстановления» после ошибки, который переводил бы парсер в состояние, где разбор можно продолжить. Типичный способ добиться этой цели – выбрать одно или несколько слов, которые парсер сможет использовать для синхронизации входа со своим внутренним состоянием. Встретив ошибку, парсер отбрасывает входные символы, пока не найдет синхронизирующее слово, а затем переустанавливает свое внутреннее состояние, так чтобы оно было согласовавано с этим словом.

В Algol-подобных языках, где точки с запятой используются в качестве разделителей предложений, синхронизирующим словом часто служит именно точка с запятой. При обнаружении ошибки парсер повторно вызывает сканер, пока не найдет точку с запятой. Затем он изменяет состояние на такое, которое возникло бы не в результате ошибки, а после успешного распознавания полного предложения, скажем нетерминального символа *Stmt*.

Парсер рекурсивного спуска может просто отбрасывать слова, пока не найдет точку с запятой. В этот момент он может вернуть управление в точку, где функция разбора предложений сообщает об успехе. Возможно, при этом придется что-то проделать со стеком процессора или воспользоваться нелокальным переходом, какой в языке C организуют функции `setjmp` и `longjmp`.

В LR(1)-парсере такого рода синхронизация сложнее. Парсер отбрасывает вход, пока не найдет точку с запятой. Затем он просматривает стек разбора назад, пока не найдет такое состояние *s*, что *Goto*[*s*, *Stmt*] содержит правильную, не ошибочную запись. Первое такое состояние в стеке представляет предложение, содержащее ошибку. Затем процедура восстановления после ошибки отбрасывает записи стека вплоть до этого состояния, помещает в стек состояние, хранящееся в *Goto*[*s*, *Stmt*], и возобновляет разбор как ни в чем не бывало.

В случае таблично управляемого парсера, все равно – LL(1) или LR(1), автору компилятора нужен способ сообщить генератору парсеров, где синхронизироваться. Это можно сделать с помощью продукции ошибки, правая часть которой включает зарезервированное слово, обозначающее точку синхронизации после ошибки, и один или несколько синхронизирующих токенов. С помощью такой конструкции генератор парсеров может реализовать желаемое поведение.

Разумеется, компилятор не должен пытаться генерировать и оптимизировать код для синтаксически неправильной программы. Это требует простого протокола взаимодействия между механизмом восстановления после ошибок и драйвером, который вызывает различные проходы компилятора; его цель – остановить компиляцию в случае ошибки разбора.

3.5.2. Унарные операторы

Классическая грамматика выражений содержит только бинарные операторы. Однако в алгебре имеются и унарные операторы, например унарный минус и абсолютная величина. В языках программирования встречаются и другие унарные операторы, как то: инкремент, декремент, взятие адреса, разыменование, булево дополнение и приведения типов. Добавление таких операторов в грамматику выражений требует осторожности.

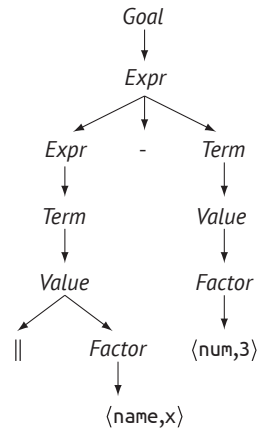
Рассмотрим добавление унарного оператора абсолютной величины, ‖, в классическую грамматику выражений. Абсолютная величина должна иметь больший приоритет, чем × и ÷. Однако ее приоритет должен быть меньше, чем у *Factor*, чтобы вычисление выражений в скобках производилось до применения ‖. Один из способов написать такую грамматику показан на рис. 3.30(a). С такими добавлениями грамматика по-прежнему обладает свойством LR(1). Она позволяет программисту найти абсолютную величину любого множителя *Factor*.

На рис. 3.30(b) изображено дерево разбора строки `‖x - 3`. Правильно показано, что код должен вычислять ‖ x до вычитания. Грамматика не дает программисту написать ‖‖ x, потому что это лишено математического смысла. Однако она позволяет написать ‖(‖ x), что ничуть не более осмысленно, чем ‖‖ x.

Невозможность написать ‖‖ x вряд ли ограничивает выразительную способность языка. Но для других унарных операторов проблема может оказаться более серьезной. Например, программисту на C бывает необходимо написать `**p`, чтобы разыменовать переменную, объявленную как `char **p;`. Мы можем также добавить продукцию разыменования *Value*: *Value* → * *Value*. Результирующая грамматика по-прежнему обладает свойством LR(1), даже если мы заменим оператор × в *Term* → *Term* × *Value* на *, перегрузив оператор «*», как это сделано в C. Такой же подход работает для унарного минуса.

0	*Goal*	→	*Expr*
1	*Expr*	→	*Expr* + *Term*
2		|	*Expr* - *Term*
3		|	*Term*
4	*Term*	→	*Term* × *Value*
5		|	*Term* ÷ *Value*
6		|	*Value*
7	*Value*	→	|| *Factor*
8		|	*Factor*
9	*Factor*	→	(*Expr*)
10		|	num
11		|	name

(a) Грамматика



(b) Дерево разбора || x - 3

Рис. 3.30 Добавление унарного оператора абсолютной величины в классическую грамматику выражений

3.5.3. Обработка контекстно-зависимой неоднозначности

Когда одно слово представляет два разных смысла, может возникнуть синтаксическая неоднозначность. Пример такой проблемы возникал в нескольких ранних языках программирования, включая FORTRAN, PL/I и ADA. В этих языках круглые скобки использовались для обрамления как индексных выражений, ссылающихся на элементы массива, так и списка аргументов подпрограммы или функции. Для текста `fee(i,j)` грамматика допускает два вывода: двумерный массив и вызов процедуры. Чтобы их различить, необходимо знать объявленный тип `fee`, а эта информация синтаксически не очевидна. В любом случае сканер, без сомнения, классифицирует `fee` как `name`.

Ни одна из таких конструкций не встречается в классической грамматике выражений. Мы можем добавить продукции, выводящие их из *Factor*.

9	*Factor*	→	(*Expr*)
10		|	num
11		|	name
12		|	*Subscript*
13		|	*Call*
14	*Subscript*	→	name (*ArgList*)
15	*Call*	→	name (*ArgList*)

В разделе 3.3.1 мы видели, что использование квадратных скобок для *Subscript* и круглых для *Call* также устраняет неоднозначность.

Поскольку правые части правил 14 и 15 одинаковы, эта грамматика неоднозначна. Построитель таблиц LR(1) сообщит о конфликте reduce-reduce между этими продукциями.

Для разрешения неоднозначности необходимы знания, выходящие за пределы синтаксиса. В нисходящем парсере рекурсивного спуска автор компилятора может объединить код для *Subscript* и *Call* и добавить код, необходимый для проверки объявленного типа имени `name`. В таблично управляемом парсере, созданном генератором, решение должно быть включено в каркас, предлагаемый инструментами.

Для нисходящего парсера автор компилятора должен также выполнить левую факторизацию вхождений `name`.

К решению этой проблемы есть два подхода. Автор компилятора может объединить вызов функции и ссылку на элемент массива в одну продукцию. Эта схема откладывает проблему до более поздней стадии трансляции, когда ее можно будет разрешить, зная информацию о типе, полученную из объявлений. Парсер должен построить представление, которое сохраняет информацию, необходимую для разрешения неоднозначности; на каком-то более позднем шаге он перепишет ссылку либо как ссылку на элемент массива, либо как вызов функции.

Определение до использования
Во многих языках программирования требуется, чтобы каждое имя было объявлено раньше, чем оно используется в коде.

Альтернативно сканер может классифицировать идентификаторы на основе их объявленных типов, а не их микросинтаксиса. Эта схема требует протокола взаимодействия между сканером и парсером; его нетрудно определить, коль скоро в языке действует правило определения до использования. Объявление разбирается, прежде чем имя используется. Парсер может предоставить сканеру доступ к своей внутренней таблице символов, чтобы тот мог относить идентификаторы к разным категориям, например `variable-name` или `function-name`. Затем эти категории могут употребляться в грамматике как отдельные нетерминальные символы. Соответствующие продукции принимают вид:

```
14  Subscript  →  variable-name ( ArgList )
15  Call       →  function-name ( ArgList )
```

После такого переписывания грамматика становится однозначной. Сканер возвращает разные синтаксические категории в каждом случае, так что парсер может их различить. Разумеется, проектировщик языка может избежать такой неоднозначности. В таких языках, как C и BCPL, синтаксис вызова функций и ссылки на элемент массива различаются, поэтому неоднозначность не возникает.

ИТОГИ РАЗДЕЛА

Локализация ошибок, сообщения об ошибках и восстановление после ошибки – все это очень важно для удобства работы с компилятором. Автор компилятора должен позаботиться о том, чтобы компилятор находил как можно больше ошибок за один прогон и сообщал об их причинах максимально понятно.

Унарные операторы и неоднозначные конструкции осложняют грамматики языков программирования. Автор компилятора может бороться с этими проблемами путем модификации грамматики, изменения протокола взаимодействия сканера и парсера и того или иного сочетания обоих подходов.

ВОПРОСЫ

1. В языке программирования C для обозначения индекса массива используются квадратные скобки, а для обозначения вызова функции – круглые. Как это упрощает построение парсера для C?
2. В грамматике унарной абсолютной величины введен новый терминальный символ для унарного оператора. Рассмотрите добавление унарного минуса в классическую грамматику выражений. Осложняет ли дело тот факт, что один и тот же терминальный символ выступает в двух разных ролях? Обоснуйте свой ответ.

3.6. Дополнительные темы

Чтобы построить удовлетворительный парсер, автор компилятора должен понимать основы разработки грамматики и парсеров. После того как парсер заработал, зачастую его производительность можно повысить. В этом разделе рассматриваются два конкретных вопроса конструирования парсеров. Во-первых, мы поговорим о преобразованиях грамматики, которые уменьшают длину вывода и тем самым увеличивают скорость разбора. Эти идеи применимы как к нисходящим, так и к восходящим парсерам. Во-вторых, мы обсудим преобразования грамматики и таблиц *Action* и *Goto*, которые уменьшают размер таблиц. Эти приемы применимы только к LR-парсерам.

3.6.1. Оптимизация грамматики

Хотя синтаксический анализ больше не считается основным пожирателем времени компиляции, тем не менее компилятор не должен тратить на разбор несообразно много времени. Форма грамматики напрямую влияет на время, необходимое для ее разбора. Нисходящий парсер производит расширение каждой продукции, участвующей в выводе. Восходящий парсер производит свертку каждой продукции. Грамматика, порождающая более короткие выводы, требует меньше времени для разбора.

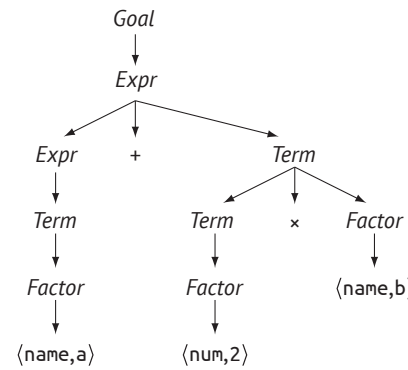
Автор компилятора часто может переписать грамматику, уменьшив высоту дерева разбора. При этом сокращается число расширений в нисходящем парсере и число сверток в восходящем. Оптимизация грамматики не может изменить асимптотическое поведение парсера; в конце концов, в дереве

разбора должен быть листовый узел для каждого символа во входном потоке. Но даже уменьшение константы в наиболее активно используемых частях грамматики, в частности грамматики выражений, может дать достаточный эффект, чтобы оправдать потраченные усилия.

Снова рассмотрим классическую леворекурсивную грамматику выражений, показанную на рис. 3.31(a). Ее таблицы LR(1)-разбора приведены на рис. 3.33. Эта грамматика порождает довольно большие деревья разбора даже для простых выражений. Например, дерево разбора выражения a + 2 × b насчитывает 14 узлов, как показано на рис. 3.31(b). Пять из этих узлов – листья, их мы исключить не можем; изменение грамматики не может укоротить входную программу. Однако количество и расположение внутренних узлов целиком зависит от вывода.

0	*Goal*	→	*Expr*
1	*Expr*	→	*Expr* + *Term*
2		|	*Expr* - *Term*
3		|	*Term*
4	*Term*	→	*Term* × *Factor*
5		|	*Term* ÷ *Factor*
6		|	*Factor*
7	*Factor*	→	(*Expr*)
8		|	num
9		|	name

(a) Классическая грамматика выражений



(b) Дерево разбора a + 2 × b

Рис. 3.31 И снова классическая грамматика выражений

Любой внутренний узел имеет только одного потомка, являющегося кандидатом на оптимизацию. В последовательности *Expr* → *Term* → *Factor* → ⟨name, a⟩ четыре узла используются для одного слова из входного потока. Мы можем устранить по крайней мере один уровень – узлов *Factor*, – воспользовавшись альтернативным расширением *Term* вообще без *Factor*, показанным на рис. 3.32(a). В этом случае число альтернатив для *Term* утраивается, зато в дереве разбора будет на один уровень меньше, как показано на рис. 3.32(b).

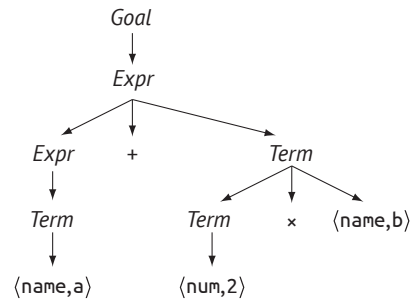
В LR(1)-парсере это изменение устраняет три из девяти действий свертки, а все пять переносов остаются без изменения. В нисходящем парсере рекурсивного спуска для эквивалентной предиктивной грамматики было бы устранено 3 из 14 вызовов процедур.

В общем случае любую продукцию, правая часть которой содержит один символ, можно устранить. Мы называем такие продукции *бесполезными*. Иногда у бесполезных продукций есть определенная цель – сделать грамматику компактнее и, быть может, удобнее для восприятия либо же навязать

выводу определенную форму. (Напомним, что простейшая из наших грамматик выражений допускает строку a + 2 × b, но не кодирует в дереве разбора никакой идеи предшествований.)

4	*Term* →	*Term* × (*Expr*)
5	|	*Term* × name
6	|	*Term* × num
7	|	*Term* ÷ (*Expr*)
8	|	*Term* ÷ name
9	|	*Term* ÷ num
10	|	(*Expr*)
11	|	num
12	|	name

(a) Другие продукции для *Term*



(b) Дерево разбора a + 2 × b

Рис. 3.32 Другие продукции для *Term*

В разделе 5.3 мы обсудим методы связывания вычислений с конкретными точками разбора. В этом контексте продукция, бесполезная с точки зрения синтаксиса, может играть важную роль в трансляции, потому что создает в выводе точку, где можно выполнить определенное действие.

У исключения бесполезных продукций есть цена. В LR(1)-парсере это может привести к увеличению размера таблиц. В нашем примере исключение *Factor* удаляет один столбец из таблицы *Goto*, но дополнительные продукции для *Term* увеличивают размер CC с 32 до 46 множеств. Таким образом, число столбцов в таблицах уменьшилось на единицу, зато число строк увеличилось на 14. В результирующем парсере будет меньше сверток (и работать он будет быстрее), но таблицы станут больше.

В написанном вручную парсере рекурсивного спуска более длинная грамматика может увеличить число альтернатив, которые нужно проверять перед расширением какой-то левой части. Иногда автор компилятора может компенсировать повышенные затраты путем объединения нескольких ветвей. Например, в нашем парсере рекурсивного спуска на рис. 3.10 код обработки слов + и - в функции `ExprPrime` был объединен, потому что в обоих случаях парсер выполняет одни и те же действия.

3.6.2. Уменьшение размера таблиц LR(1)-разбора

К сожалению, таблицы LR(1)-разбора, сгенерированные даже для небольших грамматик, могут быть велики. На рис. 3.33 ниже показаны таблицы *Action* и *Goto* для классической грамматики выражений. Есть много способов уменьшить размер таблиц LR(1). В этом разделе описываются три таких подхода.

	Таблица *Action*									Таблица *Goto*		
Состояние	**eof**	**+**	**–**	**×**	**÷**	**(**	**)**	**num**	**name**	***Expr***	***Term***	***Factor***
0						s 4		s 5	s 6	1	2	3
1	acc	s 7	s 8									
2	r 4	r 4	r 4	s 9	s 10							
3	r 7	r 7	r 7	r 7	r 7							
4						s 14		s 15	s 16	11	12	13
5	r 9	r 9	r 9	r 9	r 9							
6	r 10	r 10	r 10	r 10	r 10							
7						s 4		s 5	s 6		17	3
8						s 4		s 5	s 6		18	3
9						s 4		s 5	s 6			19
10						s 4		s 5	s 6			20
11		s 21	s 22				s 23					
12		r 4	r 4	s 24	s 25		r 4					
13		r 7	r 7	r 7	r 7		r 7					
14						s 14		s 15	s 16	26	12	13
15		r 9	r 9	r 9	r 9		r 9					
16		r 10	r 10	r 10	r 10		r 10					
17	r 2	r 2	r 2	s 9	s 10							
18	r 3	r 3	r 3	s 9	s 10							
19	r 5	r 5	r 5	r 5	r 5							
20	r 6	r 6	r 6	r 6	r 6							
21						s 14		s 15	s 16		27	13
22						s 14		s 15	s 16		28	13
23	r 8	r 8	r 8	r 8	r 8							
24						s 14		s 15	s 16			29
25						s 14		s 15	s 16			30
26		s 21	s 22				s 31					
27		r 2	r 2	s 24	s 25		r 2					
28		r 3	r 3	s 24	s 25		r 3					
29		r 5	r 5	r 5	r 5		r 5					
30		r 6	r 6	r 6	r 6		r 6					
31		r 8	r 8	r 8	r 8		r 8					

Рис. 3.33 Таблицы *Action* и *Goto* для классической грамматики выражений

Уменьшение грамматики

Часто автор компилятора может переписать грамматику, уменьшив число продукций. Обычно чем меньше продукций, тем меньше таблицы. Например, в классической грамматике выражений различие между числом и идентификатором несущественно в продукциях для *Goal*, *Expr*, *Term* и *Factor*. Замена двух продукций *Factor* → `num` и *Factor* → `name` одной продукцией *Factor* → `val` уменьшает грамматику на одну продукцию. В таблице *Action* каждому терминальному символу соответствует один столбец. Сворачивание `num` и `name` в один символ `val` удаляет один столбец. Конечно, сканер тогда должен возвращать одинаковую синтаксическую категорию, или слово для `num` и `name`.

Аналогичные аргументы можно привести в пользу объединения × и ÷ в терминальный символ `muldiv`, а + и - – в терминальный символ `addsub`. Все такие замены устраняют один терминальный символ и продукцию. Грамматика выражений, уменьшившаяся после этих трех изменений, показана на рис. 3.34(a). Эта грамматика порождает меньший набор *CC* вследствие удаления строк из таблицы. А поскольку число терминальных символов в грамматике уменьшилось, то и столбцов в таблице будет меньше.

У объединения + и - в одну категорию такой же эффект, как у объединения ветвей в нисходящем парсере рекурсивного спуска, которое мы обсуждали в разделе 3.6.1.

1	*Factor*	→	*Expr*
2	*Expr*	→	*Expr* `addsub` *Term*
3		|	*Term*
4	*Term*	→	*Term* `muldiv` *Factor*
5		|	*Factor*
6	*Factor*	→	(*Expr*)
7		|	`val`

(a) Уменьшенная грамматика выражений

	Таблица *Action*						Таблица *Goto*		
Состояние	**eof**	**addsub**	**muldiv**	**(**	**)**	**val**	***Expr***	***Term***	***Factor***
0				s 4		s 5	1	2	3
1	acc	s 6							
2	r 3	r 3	s 7						
3	r 5	r 5	r 5						
4				s 11		s 12	8	9	10
5	r 7	r 7	r 7						
6				s 4		s 5		13	3
7				s 4		s 5			14
8		s 15			s 16				
9		r 3	s 17		r 3				
10		r 5	r 5		r 5				
11				s 11		s 12	18	9	10
12		r 7	r 7		r 7				
13	r 2	r 2	s 7						
14	r 4	r 4	r 4						
15				s 11		s 12		19	10
16	r 6	r 6	r 6						
17				s 11		s 12			20
18		s 15			s 21				
19		r 2	s 17		r 2				
20		r 4	r 4		r 4				
21		r 6	r 6		r 6				

(b) Таблицы *Action* и *Goto* для уменьшенной грамматики выражений

Рис. 3.34 Уменьшенная грамматика выражений и ее таблицы

Результирующие таблицы *Action* и *Goto* показаны на рис. 3.34(b). Таблица *Action* содержит 132 записи, а таблица *Goto* – 66 записей, т. е. всего 198 записей. Неплохой результат по сравнению с таблицами для оригинальной грамматики, насчитывавшими 384 записи. Благодаря изменению грамматики удалось сократить размер таблиц на 48 %.

Могут существовать другие соображения, ограничивающие возможности автора компилятора по части объединения продукций. В частности, две похожие продукции могут нуждаться в разных синтаксически управляемых схемах трансляции, так что их объединение не принесет практической пользы (см. раздел 5.3).

Объединение строк и столбцов

Уменьшение грамматики может привести к значительному сокращению размеров таблиц *Action* и *Goto*. Но и результирующие таблицы, возможно, могут быть улучшены. Например, рассмотрим таблицы на рис. 3.34(b). Строки таблицы *Action* для состояний 0, 6 и 7 одинаковы, как и строки для состояний 4, 11, 15 и 17. Генератор таблицы может объединять одинаковые строки или столбцы и соответственно перенумеровывать состояния. Объединение одинаковых строк в двух множествах выше дополнительно сократит таблицу *Action* на рис. 3.34(b) со 132 до 102 записей – еще на 23 %.

В сочетании описанные методы уменьшили размер таблиц с 288 до 102 записей, или на 65 %.

Чтобы использовать получившуюся таблицу, заготовке парсера необходимо отображение состояния парсера на индекс строки в таблице *Action*. Генератор таблиц может аналогично объединить одинаковые столбцы. Отдельное исследование таблицы *Goto* даст другой набор комбинаций состояний – в частности, все строки, содержащие только нули, следует сжать в одну строку.

В некоторых случаях генератор таблиц может доказать, что две строки или два столбца различаются только в позициях, где одна (или один) содержит «ошибочную» запись (обозначенную на наших рисунках пробелом). На рис. 3.33 столбцы `eof` и `num` отличаются только там, где один из них содержит пробел. Объединение таких столбцов дает одно и то же поведение на правильных входах. Но поведение парсера на ошибочных входах изменяется и может негативно отразиться на способности парсера выдавать точные и полезные сообщения об ошибках.

Механизм объединения строк и столбцов похож на метод сжатия таблицы переходов ДКА, который мы обсуждали в разделе 2.5.4. Размер таблицы эти методы уменьшают. Но если уменьшение занятого объема памяти добавляет лишний уровень косвенности при каждом доступе к таблице, то необходимо хорошенько подумать, стоит ли такая экономия памяти дополнительных затрат на операции с ней. Генератор таблиц мог бы использовать и другие приемы для представления разреженных матриц – но все равно автор реализации должен рассматривать компромисс между уменьшением размера занятой памяти и увеличением стоимости доступа.

Непосредственное кодирование таблицы

И наконец, генератор парсеров мог бы вообще отказаться от таблично управляемой заготовки, отдав предпочтение непосредственному кодированию. Каждое состояние становится небольшой ветвью `case` или набором предложений `if-then-else`, который проверяет тип следующего символа и в соответствии с ним производит перенос, свертку, приемку или выдачу сообщения об ошибке. Таблицы *Action* и *Goto* заменяются кодом. (Подобное преобразование для сканеров мы видели в разделе 2.5.2.)

В результирующем парсере все состояния «не имеет значения» в таблицах *Action* и *Goto*, обозначенные на рисунках пробелами, не представлены. Эта

экономия памяти может быть съедена увеличившимся размером кода, поскольку теперь для каждого состояния нужен код. Однако в новом парсере нет таблиц, он не производит поиск в таблицах и обходится без внешнего цикла, который имеется в заготовке парсера. Хотя его структура почти непостижима для человека, выполняться он должен быстрее соответствующего таблично управляемого парсера. При правильной технике организации кода такой парсер может похвастаться отличной локальностью в кеше и в виртуальной памяти. Например, автору компилятора следует поместить функции для грамматики выражений на одну страницу виртуальной памяти, чтобы они не конфликтовали друг с другом в коде кеша.

Использование других алгоритмов построения

Существуют и другие алгоритмы построения парсеров типа LR. Из них упомянем построение SLR(1) – simple LR(1) (простой LR(1)) – и LALR(1) – lookahead LR(1) (LR(1) с заглядыванием вперед). Оба построения порождают меньшие таблицы, чем канонический алгоритм LR(1).

Алгоритм SLR(1) принимает меньший класс грамматик, чем каноническое построение LR(1). SLR(1)-грамматики ограничены, так чтобы алгоритм заполнения таблиц мог различать записи переноса и свертки с помощью множеств FOLLOW. Это ограничение устраняет необходимость опережающих символов в элементах. Результирующий канонический набор множеств элементов имеет меньше состояний, а значит, в таблице меньше строк. Этот метод допускает многие грамматики, представляющие практический интерес.

Алгоритм LALR(1) основан на том наблюдении, что базовые элементы в множестве, представляющем состояние, критичны и что остальные элементы можно добавить в множество, вычислив его *замыкание*. В построении таблиц LALR(1) базовые элементы используются для вычисления канонического набора, а после достижения неподвижной точки вычисляется их замыкание. Алгоритм порождает канонический набор, близкий по размеру к набору, порождаемому алгоритмом SLR(1). Детали различаются, но размеры таблиц похожи.

Напомним, *базовые* элементы множества – это те, что порождаются функциями *goto*. Элементы, добавленные функцией *closure*, базовыми не являются.

Каноническое построение таблиц LR(1)-разбора, представленное выше в этой главе, является самым общим из всех этих алгоритмов построения таблиц. Оно порождает самые большие таблицы, но зато допускает самый широкий класс грамматик. Применение подходящих методов уменьшения таблиц LR(1) может приблизить их размер к размеру, достигаемому более ограниченными методами.

Построение SLR(1) допускает меньший класс грамматик, чем LALR(1). В свою очередь, построение LALR(1) допускает меньший класс грамматик, чем каноническое построение LR(1). Однако можно доказать несколько противоречащий интуиции результат: любой язык, имеющий LR(1)-грамматику, имеет также SLR(1)- и LALR(1)-грамматику. Грамматики для этих более огра-

ничительных алгоритмов построения должны быть организованы так, чтобы алгоритмы могли различить действия переноса и действия свертки.

3.7. Итоги и перспективы

Почти в любом компиляторе имеется парсер. На протяжении многих лет синтаксический разбор был предметом активных исследований. Это привело к разработке многочисленных методов построения эффективных парсеров. Семейство LR(1)-грамматик включает все КСГ, которые можно разобрать детерминированно. Инструментальные средства порождают эффективные парсеры с доказуемо сильными свойствами обнаружения ошибок. Это сочетание возможностей вкупе с широкой доступностью генераторов парсеров для грамматик классов LR(1), SLR(1) и LALR(1) снизило интерес к другим методам восходящего разбора, таким как операторное предшествование.

У нисходящих парсеров рекурсивного спуска есть свои преимущества. Пожалуй, из всех парсеров такие проще всего писать вручную. Они предлагают великолепные возможности для обнаружения и исправления синтаксических ошибок. Они эффективны; на самом деле хорошо написанный нисходящий парсер рекурсивного спуска может оказаться быстрее таблично управляемого LR(1)-парсера. (Схема с непосредственным кодированием для LR(1) может превзойти их по быстроте.) Применение нисходящего парсера рекурсивного спуска позволяет автору компилятора проще разрешать неоднозначности в исходном языке, которые могут поставить в тупик LR(1)-парсер – например, при разборе языка, в котором ключевые слова могут также быть идентификаторами. Автору компилятора, который по той или иной причине хочет написать парсер вручную, мы настоятельно рекомендуем использовать нисходящий метод рекурсивного спуска.

Пример языка, имеющего LR(1)-грамматику, но не имеющего эквивалентной LL(1)-грамматики, см. в упражнении 3.10.

Выбор между LR(1)- и LL(1)-грамматиками часто зависит от доступности инструментов. На практике мало найдется конструкций языков программирования (если таковые вообще имеются), которые попадают в зазор между LR(1)- и LL(1)-грамматиками. Поэтому начинать с имеющегося генератора парсеров всегда лучше, чем реализовывать генератор парсеров с нуля.

Существуют более общие алгоритмы синтаксического разбора. Но на практике ограничения, которые свойства LR(1) и LL(1) налагают на КСГ, похоже, позволяют проектировщикам языков выразить большую часть возможностей, встречающихся в реальных языках программирования.

Замечания к этой главе

В самых первых компиляторах парсеры были написаны вручную [28, 238, 324]. Богатый синтаксис языка ALGOL-60 стал серьезным вызовом для ав-

торов ранних компиляторов. Они испробовали разные схемы разбора этого языка; Ранделл и Расселл составили обзор методов, нашедших применение в компиляторах ALGOL-60 [303].

Айронс был одним из первых, кто отделил понятие синтаксиса от трансляции [214]. Кажется, Лукас ввел понятие разбора методом рекурсивного спуска [263]. Конвей применил аналогичные идеи к эффективному однопроходному компилятору языка COBOL [103].

Идеи, положенные в основу LL- и LR-разборов, были развиты в 1960-х годах. Льюис и Стирнс ввели понятие LL(k)-грамматик [253], а Розенкранц и Стирнс более глубоко описали свойства LL(k)-грамматик [313].

Фостер разработал алгоритм преобразования некоторых грамматик в форму LL(1) [161]. Вуд формализовал понятие левой факторизации грамматики и исследовал теоретические проблемы, возникающие при преобразовании грамматики в форму LL(1) [365–367].

Кнут разработал теорию, лежащую в основе LR(1)-разбора [239]. Деремер и др. разработали алгоритмы построения таблиц SLR(1) и LALR(1), благодаря которым использование генераторов LR-парсеров стало практически возможным на тогдашних компьютерах с ограниченной памятью [132, 133]. Уэйт и Гоос описали технику автоматического исключения бесполезных продукций в алгоритме построения таблиц LR(1) [352]. Пенелло предложил непосредственное кодирование таблиц в исполняемый код [291]. Книга Ахо и Ульмана [9] – авторитетный учебник по LL- и LR-разборам. Билл Уэйт привел пример грамматики, описанной в упражнении 3.10.

Несколько алгоритмов разбора произвольных КСГ появилось в 1960–1970-х годах. В работах Кока и Шварца [98], Ирли [146], Касами [223] и Янгера [370] были разработаны алгоритм схожей вычислительной сложности. Алгоритм Эрли заслуживает упоминания в силу своей схожести с алгоритмом построения таблиц LR(1). Его алгоритм выводит множество возможных состояний разбора на этапе синтаксического разбора, а не во время выполнения, тогда как методы LR(1) заранее вычисляют их в генераторе парсеров. С общей точки зрения, LR(1)-алгоритмы можно рассматривать как естественную оптимизацию алгоритма Эрли.

Упражнения

К разделу 3.2

1. Напишите контекстно-свободную грамматику для синтаксиса регулярных выражений.
2. В ответ на просьбу определить однозначную контекстно-свободную грамматику два студента на экзамене дали разные ответы. Первый определил ее как «грамматику, в которой каждое предложение имеет единственное дерево разбора при левостороннем выводе». Второй – как «грамматику, в которой каждое предложение имеет единственное дерево разбора при любом выводе». Кто из них прав?

3. Рассмотрим роботизированную руку, принимающую две команды: ▽ кладет яблоко в сумку, а △ вынимает яблоко из сумки. Предположим, что в начале работы сумка пуста.

Правильная последовательность команд для руки не должна содержать никакого префикса, в котором команд △ больше, чем команд ▽. Таким образом, последовательности ▽▽△△ и ▽△▽ правильны, а ▽△△▽ и ▽△▽△△ – нет.

Напишите грамматику, которая описывает все правильные команды для роботизированной руки.

К разделу 3.3

4. Следующая грамматика непригодна для нисходящего предиктивного парсера. Найдите, в чем проблема, и исправьте ее, переписав грамматику. Покажите, что новая грамматика удовлетворяет условию LL(1).

```
1  L → R a
2    | Q ba
3  R → aba
4    | caba
5    | R bc
6  Q → bbc
7    | bc
```

5. Грамматики, которые можно разобрать сверху вниз, путем линейного сканирования слева направо с заглядыванием вперед на *k* слов, называются LL(*k*)-грамматиками. В тексте условие LL(1) описывается в терминах множеств FIRST. Как бы вы определили множества FIRST, необходимые для описания условия LL(*k*)?

6. Предположим, что лифт управляется двумя командами: ↑ – подняться на один этаж и ↓ – опуститься на один этаж. Предположим, что здание может быть сколь угодно высоким и что в начале работы лифт находится на этаже *x*.

Напишите LL(1)-грамматику, генерирующую произвольные последовательности команд, которые (1) никогда не заставляют лифт опуститься ниже этажа *x* и (2) всегда возвращают лифт на этаж *x* в конце последовательности. Например, ↑↑↓↓ и ↑↓↑↓ – допустимые последовательности команд, а ↑↓↓↑ и ↑↓↓ – нет. Пустая последовательность считается допустимой. Докажите, что ваша грамматика удовлетворяет условию LL(1).

К разделу 3.4

7. Нисходящие и восходящие парсеры строят деревья разбора в разном порядке. Напишите две программы, `TopDown` и `BottomUp`, которые принимают дерево разбора и печатают его узлы в порядке построения. `TopDown` должна отображать порядок для нисходящего парсера, а `BottomUp` – для восходящего.

8. Язык *ClockNoise* (*CN*) (тиканье часов) представлен следующей грамматикой:

```
1  Goal       →  ClockNoise
2  ClockNoise →  ClockNoise tick tock
3             |  tick tock
```

a. Каковы LR(1)-элементы в *CN*?
b. Каковы множества FIRST в *CN*?
c. Постройте канонический набор множеств LR(1)-элементов для *CN*.
d. Выведите таблицы *Action* и *Goal*.

9. Рассмотрим следующую грамматику:

```
1  Start →  S
2  S     →  A a
3  A     →  B C
4        |  B C f
5  B     →  b
6  C     →  c
```

a. Постройте канонический набор множеств LR(1)-элементов для этой грамматики.
b. Выведите таблицы *Action* и *Goal*.
c. Удовлетворяет ли эта грамматика условию LR(1)?

10. Для следующей грамматики неизвестна эквивалентная ей LL(1)-грамматика:

```
1  Start →  A
2        |  B
3  A     →  ( A )
4        |  a
5  B     →  ( B ]
6        |  b
```

Покажите, что эта грамматика удовлетворяет условию LR(1).

К разделу 3.6

11. Напишите грамматику выражений, которая включает бинарные операторы (+ и ×), унарный минус (–), инкремент (++) и декремент (--) с традиционными приоритетами. Считайте, что повторяющиеся унарные минусы запрещены, но повторяющиеся операторы инкремента и декремента разрешены.

К разделу 3.7

12. Рассмотрите задачу построения парсера для языка программирования SCHEME. Сравните сложность построения нисходящего парсера рекурсивного спуска и таблично управляемого LR(1)-парсера. (Считайте, что генератор таблиц LR(1)-разбора уже имеется.)

Глава 4

Промежуточные представления

Аннотация

Центральной структурой данных в компиляторе является представление компилируемой программы. Большинство проходов читают и изменяют это промежуточное представление, или ПП. Поэтому решения о том, что и как представлять, играют важнейшую роль с точки зрения стоимости и эффективности компиляции. В этой главе приведен обзор ПП, используемых в компиляторах, включая графовые ПП, линейные ПП и их гибриды, а также вспомогательных структур данных, которые поддерживает компилятор.

Ключевые слова

Промежуточное представление, графовые ПП, линейные ПП, SSA-форма, таблица символов, модель памяти, размещение в памяти

4.1. Введение

Как правило, компиляторы организованы в виде набора проходов. По мере того как компилятор получает знания о транслируемом коде, он должен как-то сохранить их и передать последующим проходам. Поэтому компилятору необходимо представлять все собранные им факты, касающиеся программы. Это собрание структур данных мы называем промежуточным представлением (ПП). У компилятора может быть одно или несколько ПП, которыми он пользуется при трансляции исходного кода на целевой. Для представления программы компилятор целиком полагается на ПП, на исходный текст он не ссылается. От свойств ПП напрямую зависит, что компилятор может делать с кодом, а чего не может.

Использование ПП позволяет компилятору совершать несколько проходов по коду. Компилятор способен сгенерировать более эффективный код входной программы, если может собрать информацию на одном проходе и использовать ее на другом. Однако это налагает одно требование: ПП должно

иметь возможность представить выведенную информацию. Поэтому компиляторы строят разнообразные вспомогательные структуры данных для ее представления и обеспечения к ней эффективного доступа. Эти структуры также составляют часть ПП.

Почти все фазы компиляторы манипулируют промежуточным представлением программы. Поэтому такие свойства ПП, как методы чтения и записи отдельных полей, нахождения определенных фактов и навигации по программе оказывают непосредственное влияние на простоту написания проходов и стоимость их выполнения.

Концептуальная дорожная карта

Эта глава посвящена вопросам, связанным с проектированием и использованием ПП при компиляции. Некоторые компиляторы пользуются для представления компилируемой программы деревьями и графами. Например, с помощью деревьев разбора легко представить результаты вывода парсера, а S-выражения в Lisp сами являются простыми графами. Поскольку большинство процессоров работают с линейным языком ассемблера, в компиляторах часто встречаются линейные ПП, напоминающие ассемблерный код. Такое линейное ПП может раскрывать свойства платформенного кода целевой машины, предоставляя тем самым дополнительные возможности компилятору.

Строя ПП программы, компилятор вычленяет информацию, которую нелегко разместить в дереве, графе или линейном ПП. Он должен понять пространство имен программы и построить вспомогательные структуры данных для хранения выведенной информации. Он должен создать план размещения в памяти, чтобы откомпилированный код мог сохранять в ней значения и выбирать их по мере необходимости. Наконец, ему необходим эффективный доступ по имени ко всей выведенной информации. Чтобы справиться со всем этим, компиляторы строят наряду с древовидным, графовым или линейным ПП вспомогательные структуры, которые являются важной частью базы знаний компилятора о программе.

Краткий обзор

Современные многопроходные компиляторы используют какую-то форму ПП для моделирования кода, подвергаемого анализу, трансляции и оптимизации. Большинство проходов потребляют ПП; поток классифицированных слов, порождаемый сканером, можно рассматривать как ПП, предназначенное для взаимодействия между сканером и парсером. Большинство проходов компилятора порождают ПП; проходы кодогенератора можно считать исключением. Многие современные компиляторы пользуются несколькими ПП на протяжении одной компиляции. В многопроходном компиляторе ПП служит основным представлением кода.

ПП должно быть достаточно выразительным, чтобы сохранить все полезные факты, которые могут понадобиться компилятору для передачи от одного прохода другому. Исходный код для этой цели неэффективен; компилятор

выводит много фактов, не имеющих представления в исходном коде. К примерам можно отнести адреса переменных или номер регистра, в котором передается параметр. Чтобы зафиксировать все детали, которые компилятор должен кодировать, многие авторы компиляторов дополняют ПП таблицами и множествами для хранения дополнительной информации. Эти структуры являются неотъемлемой частью ПП.

Выбор подходящего ПП для проекта компилятора требует понимания исходного языка, целевой машины, целей компилятора и свойств приложений, которые компилятор будет транслировать. Например, компилятор с одного исходного языка на другой мог бы использовать дерево разбора, близкое к исходному коду, тогда как компилятор, генерирующий ассемблерный код для микроконтроллера, мог бы добиться лучших результатов с помощью низкоуровневого ПП, похожего на язык ассемблера. Аналогично компилятору C могут понадобиться аннотации значений указателей, которые компилятору LISP ни к чему. Компиляторы JAVA или C++ запоминают факты об иерархии классов, которые не имеют аналогов в компиляторах C.

Часто выполняемые операции должны быть недорогими. Стоимость выполнения редких операций должна быть разумной.

Например, единственная цель операции ILOC ⇒ символ – сделать код удобочитаемым.

Реализация ПП заставляет автора компилятора сосредоточиться на практических вопросах. ПП – центральная структура данных компилятора. Компилятору нужны недорогие способы выполнения частых операций. Ему необходимы краткие способы выразить весь спектр конструкций, возникающих в процессе компиляции. Наконец, автору компилятора не обойтись без механизмов, дающих человеку возможность легко и непосредственно изучать ПП программы; залогом того, что автор уделит достаточно внимания этому последнему пункту, является личная заинтересованность.

Далее в этой главе мы будем изучать вопросы, возникающие при проектировании и использовании ПП. В разделе 4.2 приводится таксономия ПП и их свойства. В разделе 4.3 описывается несколько ПП, основанных на деревьях и графах, а в разделе 4.4 – несколько распространенных линейных форм ПП. В разделе 4.5 приводится общий обзор таблиц и способов их использования; в приложении B.4 некоторые низкоуровневые детали реализации с помощью хеш-таблиц рассматриваются более подробно. И в последних двух разделах, 4.6 и 4.7, изучаются проблемы, проистекающие из способа, каким компилятор именует значения, и правила, применяемые компилятором для размещения значений в памяти.

Несколько слов о времени

Промежуточные представления – конструкция, которая почти целиком существует на этапе компиляции. Поэтому автор компилятора полностью контролирует выбор ПП во время проектирования. Само ПП создается, используется и уничтожается во время компиляции.

Некоторая вспомогательная информация, сгенерированная как часть ПП, например таблицы символов и карты памяти, сохраняется в интересах последующих инструментов, таких как отладчики. Но это не влияет на проектирование и реализацию ПП, потому что такую информацию все равно необходимо преобразовать в стандартную форму, диктуемую инструментами.

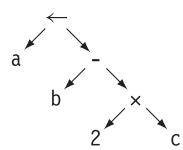
4.2. Таксономия

В компиляторах находили применение разные виды ПП. Мы при обсуждении будем рассматривать три характеристики: структурная организация, уровень абстракции и режим использования. Вообще говоря, эти три атрибута независимы; в компиляторах использовалось большинство возможных комбинаций организации, абстракции и именования.

Структурная организация

Можно выделить три больших класса ПП.

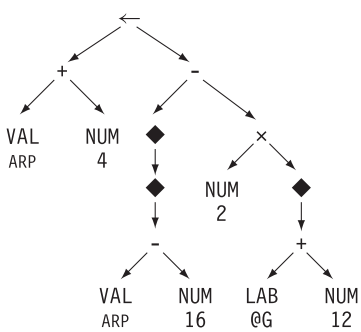
Графовые ПП кодируют знания компилятора в виде графа. Затем алгоритмы оперируют его вершинами и ребрами. Деревья разбора, использованные для изображения вывода в главе 3, – примеры графовых ПП, как и деревья, показанные на рис. 4.1(a) и (c).



(a) АСД для a ← b - 2 x c

Op	Arg_1	Arg_2	Result
×	2	c	t
-	b	t	a

(b) Четверки для a ← b - 2 x c



(c) Низкоуровневое АСД

$t_0 \leftarrow r_{arp}$ - 16
$t_1 \leftarrow$ ◆ t_0
$t_2 \leftarrow$ ◆ t_1
$t_3 \leftarrow$ @G
$t_4 \leftarrow t_3$ + 12
$t_5 \leftarrow$ ◆ t_4
$t_6 \leftarrow t_5 \times 2$
$t_7 \leftarrow t_2 - t_6$
$t_8 \leftarrow r_{arp}$ + 4
◆ $t_8 \leftarrow t_7$

(d) Низкоуровневый линейный код

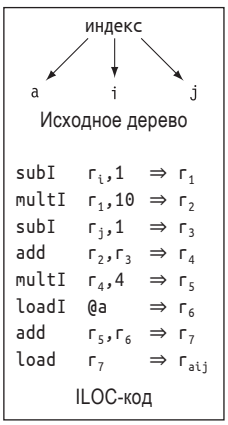
Рис. 4.1 Различные представления a ← b - 2 x c

Линейные ПП напоминают псевдокод некоей абстрактной машины. Алгоритм обходят простые линейные последовательности операций. ILOC-код, используемый в этой книге, – форма линейного ПП, как и представления, показанные на рис. 4.1(b) и (d).

Гибридные ПП сочетают элементы графовых и линейных ПП, пользуясь их сильными сторонами и отбрасывая слабые. В типичном графе потока управления (ГПУ) линейное ПП используется для представления блоков кода, а графовое – для представления потока управления между этими блоками.

Структурная организация ПП сильно влияет на то, как автор компилятора размышляет об анализе, оптимизации и генерировании кода. Например, для древовидных ПП проходы естественно организуются как та или иная форма обхода дерева. Аналогично линейные ПП естественно подводят к организации проходов как обхода последовательности операций по порядку.

Уровень абстракции



Автор компилятора должен также выбирать уровень детализации, раскрываемый ПП: его уровень абстракции. ПП могут варьироваться от формы, близкой к исходному коду, в которой доступ к массиву или вызов процедуры представлен парой вершин, до низкоуровневой формы, когда для формирования одной операции целевой машины нужно скомбинировать несколько операций ПП. Для иллюстрации возможностей во врезке показана ссылка на `a[i,j]`, представленная в виде исходного дерева, а ниже та же ссылка в виде ILOC-кода. В обоих случаях `a` – массив 4-байтовых элементов размера 10×10.

В исходном дереве компилятор легко может распознать вычисление ссылки на элемент массива, тогда как ILOC-код довольно умело скрывает этот факт. Если компилятор пытается определить, могут ли две ссылки относиться к одной и той же ячейке памяти, то с помощью исходного дерева он легко найдет и сравнит ссылки. Напротив, ILOC-код затрудняет решение таких задач. С другой стороны, если цель состоит в том, чтобы оптимизировать окончательный код, сгенерированный для доступа к массиву, то ILOC-код позволяет компилятору оптимизировать детали, остающиеся неявными в исходном дереве. Так что этой цели низкоуровневое ПП может отвечать лучше.

Уровень абстракции не зависит от структуры. На рис. 4.1 показаны четыре разных представления для предложения `a←b - 2 x c`. На рисунках (a) и (c) изображены абстрактные синтаксические деревья (АСД, *англ.* AST) на уровне абстракции, близком к исходному языку и близком к машинному коду. На рисунках (b) и (d) изображены соответствующие линейные представления.

В низкоуровневом АСД на рисунке (c) узлы представляют понятия уровня языка ассемблера. Узел `VAL` представляет значение, уже находящееся в регистре. Узел `NUM` представляет известную константу, которая может поместиться

в поле непосредственного операнда. Узел LAB представляет метку уровня ассемблера. Оператор разыменования ◆ рассматривает значение как адрес и представляет ссылку на память. С этим конкретным АСД мы встретимся еще раз в главе 11.

Трансляция ◆, встречающегося в низкоуровневом линейном коде, зависит от контекста. Слева от оператора ← он представляет команду `store`, а справа от него – команду `load`.

Уровень абстракции важен, потому что в общем случае компилятор может оптимизировать только те детали, которые раскрывает ПП. Факты, присутствующие в ПП неявно, трудно изменить, потому что компилятор трактует неявные знания единообразно, что мешает зависящей от контекста оптимизации. Например, чтобы оптимизировать код ссылки на элемент массива, компилятор должен переписать ПП ссылки. Если детали ссылки явно не представлены, то изменить их компилятор не сможет.

Режим использования

Третья характеристика относится к использованию ПП компилятором.

Определяющее ПП. *Определяющее* ПП – это основное представление компилируемого кода. Компилятор не обращается к исходному коду, а вместо этого анализирует, преобразует и транслирует одну или несколько (последовательных) версий ПП. Это определяющие ПП.

Производное ПП. *Производное* ПП компилятор строит для конкретной временной цели. Оно может дополнять определяющее ПП, как в случае графа зависимостей для планирования выполнения команд (см. главу 12). Компилятор может транслировать код в производное ПП, а его в машинный код, чтобы включить конкретную оптимизацию.

В общем случае, если ПП передается от одного прохода другому, то его следует считать определяющим. Если ПП строится внутри одного прохода для какой-то определенной цели, а затем отбрасывается, то оно производное.

Именование

Автор компилятора также должен выбрать пространство имен для ПП. Это решение определит, какие значения в программе видны для оптимизации. В процессе трансляции исходного кода компилятор должен выбирать имена и места хранения для множества разных значений.

На рис. 4.1 это показано конкретно. В АСД имена неявные; компилятор может ссылаться на любое поддерево АСД по его корневому узлу. Так, в дереве на рисунке (c) поименованы многие значения, которые нельзя поименовать на рисунке (a); это стало возможно благодаря более низкому уровню абстракции. Код на рисунке (b) создает всего два значения, которые могут использовать другие операции, тогда как код на рисунке (d) создает девять таких значений.

Схема именования сильно влияет на то, как оптимизация может улучшить код. На рисунке (d) t_0 – адрес `b` во время выполнения, t_4 – адрес `c` во время

выполнения, а t_8 – адрес a во время выполнения. Если близко расположенный код ссылается на любой из этих адресов, то оптимизация должна распознать одинаковые ссылки и повторно использовать уже вычисленные значения (см. раздел 8.4.1). Если бы компилятор поместил по адресу t_0 какое-то другое значение, то вычисленный адрес b был бы потерян, потому что его невозможно поименовать.

Представление строк

Сканер относит слова входной программы к небольшому числу категорий. С функциональной точки зрения каждое слово во входном потоке становится парой ⟨*лексема*, *категория*⟩, где *лексема* – текст слова, а *категория* – его синтаксическая категория.

Для некоторых категорий хранить и лексему, и категорию нет смысла. В категориях +, × и for имеется всего по одной лексеме. Для других категорий, например идентификаторов, чисел и строк литер, разным словам соответствуют разные лексемы. В таких случаях компилятору необходимо представлять и сравнивать лексемы.

Строки литер – один из наименее эффективных способов представления имени в компиляторе. Занятая строкой память и время сравнения двух строк в общем случае пропорциональны их длине. У компилятора есть и более эффективные решения.

Компилятор мог бы вместо этого отображать имена, встречающиеся в исходном коде, на компактное множество целых чисел. При таком подходе экономится память; целые упаковываются плотнее, чем строки литер. Этот подход экономит также время – сравнения на равенство занимают время **O**(1).

Чтобы реализовать такое отображение, автор компилятора может поручить сканеру создание хеш-таблицы (см. приложение B.4), в которой будет хранить различные строки, встречающиеся во входной программе. Затем компилятор может использовать либо индекс строки в этой «таблице строк», либо указатель на относящуюся к ней запись в таблице строк в качестве заместителя строки. Информацию о строке, например длину строковой константы или значение и тип числовой константы, можно вычислить один раз, а затем быстро ссылаться на нее с помощью таблицы.

Если использовать слишком мало имен, то пострадает оптимизация, а если слишком много, то некоторые структуры данных этапа компиляции разбухнут, и время компиляции увеличится, не принеся никакой выгоды. Эти проблемы подробно разбираются в разделе 4.6.

Практические соображения

На практике стоимость генерирования ПП и манипулирования им должны волновать автора компилятора, потому что напрямую отражаются на скорости его работы. Требования различных ПП к памяти существенно различаются. Поскольку компилятор обычно обращается ко всей памяти, которую выделил, размер памяти, выделенной для данных, имеет непосредственное отношение ко времени работы.

И последнее, но от того отнюдь не менее важное, – автор компилятора должен думать о выразительности ПП, т. е. его способности включить в себя все факты, которые компилятор должен хранить. ПП процедуры должно вклю-

чать ее код, результаты статического анализа, данные о профиле предыдущих выполнений и карты памяти, которые позволят отладчику понять код и его данные. Все эти факты должны быть выражены способом, проясняющим их связь с конкретными точками ПП.

4.3. Графовые ПП

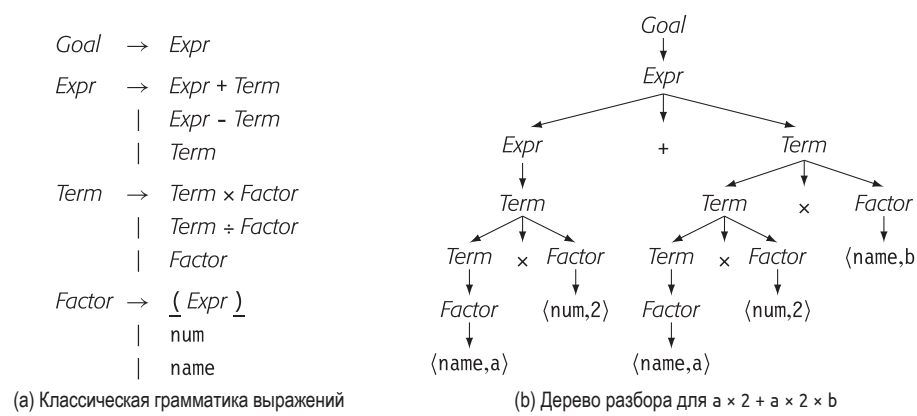
Во многих компиляторах используются ПП, представляющие код в виде графа. И хотя все графовые ПП состоят из вершин и ребер, они различаются уровнем абстракции, связью между графом и кодом и структурой графа.

4.3.1. Деревья, связанные с синтаксисом

Деревья разбора, АСД и ориентированные ациклические графы (ОАГ, *англ.* DAG) – все это графы, используемые для представления кода. Структура этих древовидных ПП соответствует синтаксису исходного кода.

Деревья разбора

В разделе 3.2 мы видели, что *дерево разбора* – это представление вывода, или разбора входной программы в виде графа. На рис. 4.2 показана классическая грамматика выражений и дерево разбора выражения `a × 2 + a × 2 × b`. Дерево разбора велико по сравнению с исходным текстом, потому что представляет полный вывод, в котором каждому символу грамматики соответствует узел дерева. Поскольку компилятор должен выделять память для каждого узла и каждого ребра, а во время компиляции обходит эти узлы и ребра, стоит подумать о том, как сжать это дерево разбора.



(a) Классическая грамматика выражений (b) Дерево разбора для `a × 2 + a × 2 × b`

Рис. 4.2 Дерево разбора для `a × 2 + a × 2 × b`

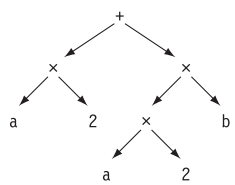
Незначительные преобразования грамматики, описанные в разделе 3.6.1, могут устранить некоторые шаги вывода и соответствующие им узлы дерева.

Но более эффективный способ сжать дерево разбора состоит в том, чтобы абстрагировать узлы, ненужные на последующих этапах работы компилятора. Этот подход приводит к упрощенной версии дерева разбора, которая называется *абстрактным синтаксическим деревом* и обсуждается ниже.

Режим использования. Деревья разбора используются в основном при обсуждении синтаксического разбора и в системах с атрибутными грамматиками, где они являются определяющими ПП. В большинстве других приложений, которым необходимо исходное дерево, авторы компиляторов стремятся использовать какую-нибудь более компактную альтернативу, например АСД или ОАГ.

Абстрактные синтаксические деревья

Абстрактное синтаксическое дерево результат сжатия дерева разбора, из которого удалены узлы большинства нетерминальных символов.



АСД для a × 2 + a × 2 × b

Абстрактное синтаксическое дерево (АСД) сохраняет структуру и семантику дерева разбора, но устраняет лишние узлы. Устраняются узлы нетерминальных символов, в которых закодированы детали вывода. АСД для выражения a × 2 + a × 2 × b показано во врезке.

Режим использования. АСД использовались в качестве определяющих ПП во многих практических компиляторах. Уровень абстракции, необходимый таким системам, меняется в широких пределах.

- Транспиляторные системы, включая синтаксически ориентированные редакторы, инструменты для рефакторинга кода и системы автоматического распараллеливания, часто применяют АСД с абстракциями, близкими к исходному коду. Структура такого АСД отражает структуру входной программы.
- Компиляторы, генерирующие ассемблерный код, могут использовать АСД. Такие системы обычно начинают с АСД, близкого к исходному коду, и систематически понижают уровень абстракции, пока он не сравняется или не окажется ниже уровня абстракции ISA целевой машины. Структура этого конечного низкоуровневого АСД отражает поток значений между операциями.

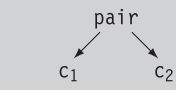
Системы на основе АСД обычно обходят ПП, применяя один из способов обхода дерева. Многие используемые в компиляторах алгоритмы естественно формулируются в терминах обхода дерева (см. раздел 11.4) или поиска в глубину (см. раздел 8.5.1).

Некоторые компиляторы строят АСД как производные ПП, потому что преобразование в АСД и из него производится быстро и может упростить другие алгоритмы. В частности, оптимизации, которые переупорядочивают выражения, выигрывают от построения АСД в качестве ПП, потому что АСД устраняет все явные имена промежуточных результатов. Другие алгоритмы,

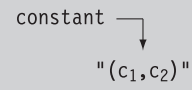
например балансировки деревьев по высоте (раздел 8.4.2) или сопоставления с древовидным образцом (раздел 11.4), «естественно» выражаются в терминах обхода дерева.

Выбор правильной абстракции

Даже в случае исходного дерева выбор представления влияет на удобство пользования. Например, в среде программирования $\mathcal{R}^n$ использовалось дерево, изображенное на рисунке (а) ниже, для представления комплексного числа в языке FORTRAN, записанного в виде (c_1,c_2). Этот выбор хорошо работал в синтаксически ориентированном редакторе, где программист мог независимо изменять c_1 и c_2; узел `pair` соответствовал скобкам и запятой.



(а) АСД, предназначенное для редактирования



(б) АСД, предназначенное для компиляции

Однако формат пары оказался проблематичным для компилятора. Во всех частях компилятора, имевших дело с константами, приходилось заводить особый случай для комплексных констант.

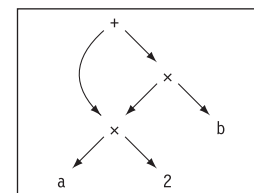
Все прочие константы представлялись одним узлом, содержащим указатель на лексему константы, как показано на рисунке (b). Использование аналогичного формата для комплексных констант усложнило бы редактор, но упростило компилятор. В масштабе всей системы преимущества, вероятно, перевесили бы трудности.

Ориентированные ациклические графы

Ориентированный ациклический граф
ОАГ – это АСД, в котором каждое уникальное поддерево представлено ровно один раз. ОАГ часто называют АСД с разделением.

Хотя АСД компактнее дерева разбора, оно точно сохраняет структуру оригинального исходного кода. Например, АСД для `a × 2 + a × 2 × b` содержит две копии выражения `a × 2`. *Ориентированный ациклический граф* (ОАГ) – это результат сжатия АСД, в котором такое дублирование устранено. В ОАГ вершины могут иметь нескольких родителей, а одинаковые поддеревья используются повторно. Такое разделение делает ОАГ более компактным, чем соответствующее дерево разбора.

Для выражений, в которых нет присваивания или вызовов функций, текстуально идентичные выражения должны порождать одинаковые значения. ОАГ для выражения `a × 2 + a × 2 × b`, показанного во врезке, отражает этот факт, разделяя единственную копию `a × 2`. Если значение `a` не может измениться между двумя случаями использования `a`, то компилятор должен сгенерировать код вычисления `a × 2` один раз, а результат использовать дважды. Эта стратегия способна уменьшить стоимость вычислений.



ОАГ для `a × 2 + a × 2 × b`

В ОАГ явно закодирована избыточность подвыражений. Если компилятор представляет такие факты в ПП, то сможет избежать затрат на их повторное установление.

При построении ОАГ для этого выражения компилятор должен доказать, что значение a не может измениться между использованиями. Если выражение не содержит ни присваиваний, ни вызовов процедур, то доказать это легко. Поскольку присваивание или вызов процедуры может изменять значение, ассоциированное с именем, алгоритм построения ОАГ обязан делать недействительными поддеревья, когда значения их операндов могут измениться.

Эффективное использование памяти и графовые представления

Во многих практических системах абстрактные синтаксические деревья использовались в качестве определяющих ПП. Выяснилось, что во многих таких системах АСД было велико по сравнению с размером входного текста. В среде программирования $\mathcal{R}^n$, созданной в университете Райса в 1980-х годах, АСД поставило две проблемы: рабочие станции середины 1980-х годов имели ограниченную память и ввод-вывод дерева замедлял работу всех инструментов $\mathcal{R}^n$.

В $\mathcal{R}^n$ каждый узел АСД занимал 92 байта. В среднем ПП содержало 11 узлов на каждое предложение исходного кода. Следовательно, АСД занимало примерно 1000 байт на предложение. На рабочей станции с 4 МБ памяти это налагало практическое ограничение – 1000 строк кода – в большинстве инструментов среды.

Не было какого-то одного решения, приведшего к этой проблеме. Чтобы упростить выделение памяти, в $\mathcal{R}^n$ АСД имели узлы только одного вида. Поэтому каждый узел включал все поля, необходимые хотя бы одному узлу. (Примерно половина узлов была листовыми, и указатели на сыновние узлы им были не нужны.) По мере развития системы росли и узлы. Новым инструментам нужны были новые поля.

Внимательный анализ может помочь в борьбе с такого рода разбуханием узлов. В $\mathcal{R}^n$ мы написали программы для анализа содержимого и способа использования АСД. Мы объединили одни поля и исключили другие. (В некоторых случаях было дешевле заново вычислить информацию, чем записывать и читать ее.) Мы использовали хеш-связывание, чтобы переместить редко используемые поля из АСД во вспомогательную таблицу. (Один бит в поле типа узла означал присутствие вспомогательных фактов, относящихся к этому узлу.) Для ввода-вывода на диск мы преобразовывали АСД в линейную форму в прямом порядке обхода, в результате чего указатели становились неявными.

В $\mathcal{R}^n$ эти изменения уменьшили размер АСД в памяти примерно на 75 %. Файлы на диске занимали примерно в два раза меньше, чем их представление в памяти. Благодаря этим изменениям $\mathcal{R}^n$ получила возможность обрабатывать программы большего размера, а инструменты стали откликаться заметно быстрее.

Режим использования. ОАГ используются в реальных системах по двум основным причинам. Если недостаток памяти ограничивает размер программ, которые компилятор способен обработать, то использование ОАГ в качестве определяющего ПП может уменьшить объем занимаемой ПП памяти. В других системах ОАГ применяются для выявления избыточности. В этом случае

преимуществом является более качественно откомпилированный код. В таких системах ОАГ используется как производная ПП – построить ОАГ, преобразовать определяющее ПП, чтобы выявить места, где есть избыточность, и отбросить ОАГ.

4.3.2. Графы

Деревья предлагают естественное представление грамматической структуры, обнаруженной в исходном коде в процессе синтаксического разбора, но из-за своей негибкой структуры они не так полезны для представления других свойств программы. Для моделирования этих аспектов поведения программы компиляторы часто используют более общие графы в качестве ПП. ОАГ, с которыми мы познакомились в предыдущем разделе, – пример такого графа.

Граф потока управления

Простой блок
последовательность операций без ветвления максимальной длины.

Самая простая единица потока управления – *простой блок* – максимальная прямолинейная, т. е. не содержащая переходов, последовательность операций. Операции блока всегда выполняются вместе, если только какая-нибудь из них не возбудит исключение. Блок начинается помеченной операцией и заканчивается условным или безусловным переходом либо предикатной операцией. Управление попадает на первую операцию блока. Операции выполняются в порядке сверху вниз. Управление выходит из блока через его последнюю операцию.

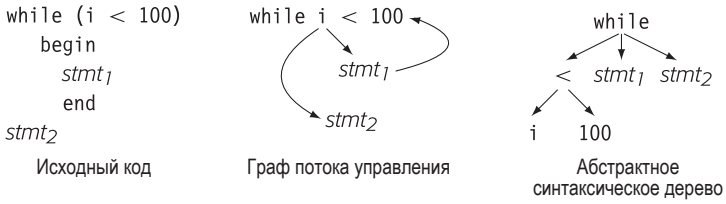
Граф потока управления
ГПУ имеет вершину для каждого простого блока и ребро для каждой возможной передачи управления.

Граф потока управления (ГПУ) моделирует поток управления между простыми блоками в процедуре. ГПУ – это ориентированный граф, $G = (N, E)$. Каждая вершина $n \in N$ соответствует простому блоку. Каждое ребро $e = (n_i, n_j) \in E$ соответствует возможной передаче управления из блока n_i в блок n_j.

Если компилятор добавляет искусственные входную и выходную вершины, то им могут не соответствовать реальные простые блоки.

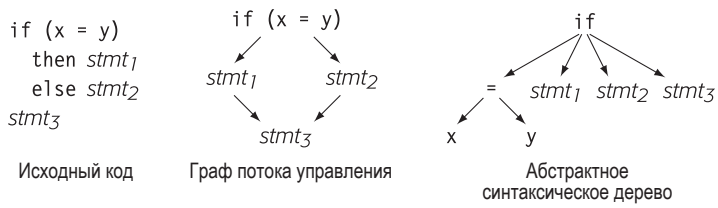
Чтобы упростить обсуждение анализа программ в главах 8 и 9, мы будем предполагать, что каждый ГПУ имеет единственную входную вершину, n_0, и единственную выходную вершину, n_f. Если имеется несколько входов в процедуру, то компилятор может создать одну вершину n_0 и добавить ребра, ведущие из n_0 в каждую настоящую точку входа. Аналогично n_f соответствует выходу из процедуры. Несколько выходов встречается чаще, чем несколько входов, но компилятор легко может добавить одну n_f и соединить ее с каждым настоящим выходом.

ГПУ дает графовое представление возможных потоков управления на этапе выполнения. Он отличается от синтаксически ориентированных ПП, таких как АСД, которые показывают грамматическую структуру. Рассмотрим показанный ниже цикл while. Его ГПУ представлен на среднем рисунке, а его АСД – на правом.



ГПУ улавливает сам смысл цикла: поток управления в нем. Обратное ребро идет из *stmt1* назад в начало цикла. Напротив, АСД улавливает синтаксическую структуру; оно ациклическое, но содержит все фрагменты, из которых можно восстановить исходный код цикла.

Для конструкции `if-then-else` и ГПУ, и АСД ациклические, как показано ниже.



И снова ГПУ моделирует поток управления; выполняется только одно из предложений $stmt_1$ или $stmt_2$, но не оба. АСД опять-таки улавливает синтаксис, но мало что дает для понимания того, как код выполняется на деле. Любая связь такого рода неявна.

Режим использования. Компиляторы обычно используют ГПУ в сочетании с каким-то другим ПП, так что ГПУ оказывается производным ПП. ГПУ представляет связи между блоками, тогда как операции внутри блока представлены другим ПП, например АСД уровня выражений, ОАГ или каким-то линейным ПП. Компилятор мог бы рассматривать такое гибридное ПП как определяющее, но из-за трудностей, связанных с поддержанием нескольких форм в согласованном состоянии, этот подход редко встречается на практике.

Построение ГПУ обсуждается в разделе 4.4.4.

Многие части компилятора зависят от ГПУ, явно или неявно. Анализ программы с целью оптимизации обычно начинается с анализа потока управления и построения ГПУ (см. главу 9). Планировщикам команд ГПУ нужен для того, чтобы понять, как будет совместно исполняться запланированный код отдельных блоков (см. главу 12). Механизм распределения регистров использует ГПУ, чтобы понять, как часто может выполняться каждая

операция и где следует вставлять команды загрузки и сохранения значений (см. главу 13).

Длина блока

Однооператорные блоки
схема, в которой каждый блок соответствует одному предложению исходного кода.

Некоторые авторы рекомендуют строить ГПУ на основе блоков, более коротких, чем простой блок. Самая распространенная альтернатива – *однооператорный блок*. Такие блоки могут упрощать алгоритмы для анализа и оптимизации.

Компромисс между ГПУ, построенными на основе однооператорных блоков и блоков максимальной длины, затрагивает время и память. ГПУ на основе однооператорных блоков имеет больше вершин и ребер, чем ГПУ на основе блоков максимальной длины. Поэтому однооператорный ГПУ будет потреблять больше памяти при прочих равных условиях. При большем числе вершин и ребер обход графа будет занимать больше времени. Но важнее то, что однооператорный ГПУ имеет гораздо больше поставленных компилятором аннотаций, чем ГПУ на основе простого блока. Время и память, необходимые для построения и использования этих аннотаций, несомненно, превосходят стоимость построения ГПУ.

С другой стороны, для некоторых оптимизаций однооператорные блоки предпочтительнее. Например, метод отложенного перемещения кода (см. раздел 10.3.1) вставляет код только на границах блока. Поэтому однооператорные блоки позволяют оптимизировать такое перемещение более точно, чем было бы возможно в случае блоков максимальной длины.

Граф зависимостей

Граф зависимостей по данным
граф, моделирующий поток значений от определения до использования в каком-то фрагменте года.

Определение
Об операции, которая создает значение, говорят, что она *определяет* это значение.

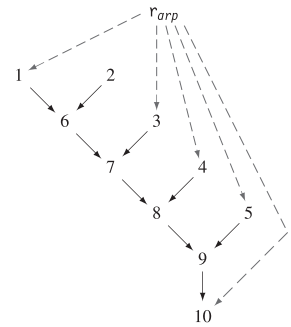
Использование
Об операции, которая ссылается на значение, говорят, что она использует это значение.

Компиляторы также используют графы для кодирования потока значения из точки, где значение создано, т. е. от *определения*, до любой точки, где оно читается, т. е. *использует*ся. *Граф зависимостей по данным* описывает эту связь. Вершины этого графа представляют операции. Большинство операций содержат и определение, и использование. Ребро графа соединяет две вершины, одна из которых представляет определение, а другая – использование. Мы изображаем графы зависимостей так, что ребра направлены от определения к использованию; некоторые авторы рисуют противоположно направленные ребра – от использования к определению.

На рис. 4.3 показан ILOC-код вычисления `a ← a × 2 × b × c × d` (повторение рис. 1.4). На рисунке (a) приведен сам ILOC-код, а на рисунке (b) – соответствующий граф зависимостей по данным.

```
1   loadAI  r_arp, @a ⇒ r_a
2   loadI   2         ⇒ r_2
3   loadAI  r_arp, @b ⇒ r_b
4   loadAI  r_arp, @c ⇒ r_c
5   loadAI  r_arp, @d ⇒ r_d
6   mult    r_a, r_2  ⇒ r_a
7   mult    r_a, r_b  ⇒ r_a
8   mult    r_a, r_c  ⇒ r_a
9   mult    r_a, r_d  ⇒ r_a
10  storeAI r_a       ⇒ r_arp, @a
```

(a) Пример кода из главы 1



(b) Граф зависимостей для этого примера

Рис. 4.3 Простой блок на языке ILOC и его граф зависимостей

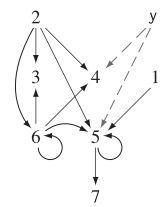
В графе зависимостей по данным имеется по одной вершине для каждой операции в блоке. Каждое ребро показывает поток одного значения. Например, ребро из вершины 3 в вершину 7 отражает тот факт, что r_b определен в предложении 3, а затем используется в предложении 7. Виртуальный регистр r_{arp} содержит адрес, отстоящий на фиксированное расстояние от начала области локальных данных. Использования r_{arp} соотносятся с его неявным определением в начале процедуры; они показаны штриховыми линиями.

Ребра графа представляют реальные ограничения на порядок операций – значение нельзя использовать, пока оно не определено. Ребра графа зависимостей вводят частичный порядок на множестве возможных способов выполнения кода. Например, граф требует, чтобы 1 и 2 предшествовали 6. Но требования, чтобы 1 или 2 предшествовали 3, не существует. Многие последовательности выполнения соблюдают зависимости, показанные в графе, в т. ч. ⟨1, 2, 3, 4, 5, 6, 7, 8, 9, 10⟩ и ⟨2, 1, 6, 3, 7, 4, 8, 5, 9, 10⟩. Планировщик команд пользуется свободой, предоставленной частичным порядком, когда планирует выполнение «не по порядку».

На более абстрактном уровне рассмотрим фрагмент кода, показанный на рис. 4.4(a), который включает несколько простых блоков, а также цикл `while` и конструкцию `if-then`. Компилятор может построить единственный граф зависимостей для всего фрагмента, он показан на рисунке (b).

```
1   x ← 0
2   i ← 1
3   while (i < 100)
4     if (y[i] > 0)
5       then x ← x + y[i]
6     i ← i + 1
7   print x
```

(a) Код



(b) Его граф зависимостей

Рис. 4.4 Взаимодействие между потоком и его графом зависимостей

Ссылки на `y[i]` выводят значения из одной вершины, представляющей все предшествующие определения `y`. Без изощренного анализа индексных

выражений компилятор не может провести различие между ссылками на отдельные элементы массива.

Этот граф зависимостей сложнее предыдущего. Вершины 5 и 6 зависят от самих себя; в них используются значения, которые могли быть определены на предыдущей итерации. Например, вершина 6 может получать значение `i` либо от 2 (на первой итерации), либо от себя самой (на всех последующих итерациях). Вершины 4 и 5 также имеют два источника значения `i`: вершины 2 и 6.

Режим использования. Графы зависимостей по данным обычно строятся для конкретной задачи, а затем отбрасываются, так что их можно назвать производным ПП. Они играют главную роль в планировании выполнения команд (глава 12), а также находят применение в разнообразных оптимизациях, особенно в преобразованиях, которые изменяют порядок выполнения команд в цикле, чтобы распараллелить его и повысить эффективность использования памяти. При более изощренном применении графа зависимостей по данным компилятор может выполнить подробный анализ значений индексов массива, чтобы определить, когда ссылки на один и тот же массив могут перекрываться.

Граф вызовов

Для оптимизации кода, пересекающего границы процедур, некоторые компиляторы выполняют *межпроцедурный* анализ и оптимизацию. Для представления вызовов процедур компилятор строит граф вызовов. В нем для каждой процедуры имеется вершина, а для каждого места вызова процедуры – ребро. Таким образом, если код вызывает q из трех текстуально различных мест в p, то граф вызовов будет иметь три ребра (p, q), по одному для каждого места вызова.

Межпроцедурная
Любая техника исследования взаимодействий более чем в одной процедуре называется *межпроцедурной*.

Внутрипроцедурная
Любая техника, применение которой ограничено одной процедурой, называется *внутрипроцедурной*.

Граф вызовов
граф, который представляет связи между процедурами программы, выражающиеся в виде вызовов.

В графе вызовов имеется по одной вершине для каждой процедуры и по одному ребру для каждого места вызова.

Построение графа вызовов осложняется как практикой программной инженерии, так и особенностями языка.

- Раздельная компиляция ограничивает возможности компилятора в части построения графа вызовов, потому что сужает множество видимых компилятору процедур. Некоторые компиляторы строят частичные графы вызовов для процедур, входящих в единицу трансляции, и оптимизируют это подмножество.
- Параметры, значениями которых являются процедуры, – как фактические параметры, так и возвращаемые значения, – создают неоднозначные вызовы, что затрудняет построение графа вызовов. Компилятор может выполнить межпроцедурный анализ, чтобы ограничить включаемое в граф множество ребер, индуцируемых такими вызовами, так что построение графа вызовов превращается в процесс итеративного уточ-

нения. (Эта проблема аналогична проблеме неоднозначных переходов при построении ГПУ, обсуждавшейся в разделе 4.4.4.)

Анализ иерархии классов
вид статического анализа, который строит модель иерархии наследования в программе.

- В объектно-ориентированных программах наследование может быть причиной неоднозначных вызовов процедур, которые разрешаются только при наличии дополнительной информации о типе. В некоторых языках анализ иерархии классов способен устранить неоднозначность многих таких вызовов, в других информация может быть неизвестна до этапа выполнения. Разрешение неоднозначных вызовов на этапе выполнения представляет серьезную проблему для построения графа вызовов; кроме того, разрешение вызовов во время выполнения влечет значительные дополнительные затраты.

В разделе 9.4 обсуждаются некоторые проблемы построения графа вызовов.

Режим использования. Графы вызовов почти всегда выступают в роли производного ПП; они строятся, чтобы поддержать межпроцедурный анализ и оптимизацию, а затем отбрасываются. Самое известное межпроцедурное преобразование, встраиваемая подстановка (см. раздел 8.7.1), изменяет граф вызовов в процессе работы, в результате чего старый граф вызовов перестает отражать действительность.

ИТОГИ РАЗДЕЛА

Графовые ПП являются абстрактным представлением компилируемого кода. Уровень абстракции графового ПП, например АСД, может варьироваться от уровня исходного кода до уровня ниже машинного кода. Графовые ПП иногда бывают определяющими, а иногда строятся в качестве производных ПП для специальных целей.

Поскольку такие ПП являются графами, в них можно закодировать связи, которые трудно представить линейными ПП. Обход графа – эффективный способ перемещения между логически связанными точками программы; большинство линейных ПП лишены такого рода связности операций.

ВОПРОСЫ

1. Дана исходная программа. Сравните ожидаемый размер ПП как функции от числа возвращенных сканером токенов для (a) дерева разбора, (b) АСД, (c) ОАГ. Можете предполагать, что вершины во всех трех формах ПП имеют одинаковый и фиксированный размер.
2. Как число ребер графа зависимостей для простого блока возрастает в зависимости от числа операций в блоке?

4.4. Линейные ПП

Линейные ПП представляют программу в виде упорядоченной последовательности операций. Они являются альтернативой графовым ПП, описанным в предыдущем разделе. Программа на языке ассемблера – форма линейного

кода. Она состоит из упорядоченной последовательности команд. Команда может содержать более одной операции; в таком случае эти операции выполняются параллельно. Линейные ПП, используемые в компиляторах, часто похожи на ассемблерный код абстрактной машины.

Логика, стоящая за линейной формой, проста. Исходный код, подаваемый на вход компилятора, записан в линейной форме, как и код целевой машины, генерируемый компилятором. В нескольких ранних компиляторах использовались линейные ПП; такая нотация казалась естественной их авторам, т. к. раньше они программировали на ассемблере.

Линейные ПП определяют полный порядок на последовательности операций. Сравните ILOC-код на рис. 4.3 с графом зависимостей по данным. В ILOC-коде имеется неявный полный порядок; граф зависимостей определяет частичный порядок, допускающий несколько порядков выполнения.

Если линейное ПП используется в качестве определяющего, то оно должно включать механизм кодирования передачи управления из одной точки программы в другую. Поток управления в линейном ПП обычно моделирует реализацию потока управления в целевой машине. Таким образом, линейный код обычно включает условные и безусловные переходы. Поток управления размечает простые блоки в линейном ПП; блок кончается в точке условного или безусловного перехода либо просто перед помеченной операцией.

Ветвления в ILOC отличаются от имеющихся в типичном языке ассемблера. В них явно задаются метки для выбранной ветви и для неявной ветви. Это устраняет все неявные передачи управления и упрощает нахождение простых блоков, их переупорядочение и построение ГПУ.

Выбранная ветвь
В большинстве ISA в условных ветвях используется только одна метка. Поток управления переходит либо на эту метку по *выбранной ветви* (taken branch), либо на следующую операцию *по неявной ветви* (fall-through branch).

Переход по неявной ветви часто быстрее, чем по выбранной.

В компиляторах использовалось много разных линейных ПП.

- Одноадресный код моделирует поведение аккумуляторных и стековых машин. В таком коде отражается использование машиной неявных имен, что позволяет компилятору подстроить код под нее. Получающееся ПП может быть очень компактным.
- Двухадресный код моделирует машину с деструктивными операциями. Такой код вышел из употребления, потому что деструктивные операции и ограничения на размер ПП утратили актуальность; трехадресный код способен моделировать деструктивные операции явно.

Деструктивная операция
операция, в которой один из операндов замещается результатом.

Такие операции, вероятно, были придуманы, чтобы сэкономить место в формате команды для 8- или 16-разрядных компьютеров.

- Трехадресный код моделирует машину, в которой большинство операций принимают два операнда и порождают результат. Развитие RISC-архитектур в 1980–1990-х годах снова сделало такой код популярным.

Далее в этом разделе описываются два широко применяемых линейных ПП: код для стековой машины и трехадресный код. Код для стековой машины предлагает компактное представление, эффективно использующее память. В тех приложениях, где размер ПП имеет значение, например в JAVA-аплетах, передаваемых по сети перед исполнением, применение кода для стековой машины оправдано. Трехадресный код моделирует формат команды современной RISC-машины; в нем имеются различные имена для двух операндов и результата. Вы уже знакомы с трехадресным кодом, это язык ILOC, используемый в книге.

4.4.1. Код для стековой машины

Код для стековой машины, вариант одноадресного кода, предполагает наличие стека операндов. Он прост для генерирования и понимания. Большинство операций читают операнды из стека и помещают результаты обратно в стек. Например, оператор вычитания выталкивает из стека два верхних элемента, а помещает в стек их разность.

Порядок работы со стеком создает потребность в новых операциях: `push` копирует значение из памяти в стек, `pop` удаляет верхний элемент стека и записывает его в память, а `swap` меняет местами два верхних элемента. Были сконструированы стековые процессоры; это ПП, похоже, появилось как модель таких ISA. Во врезке приведен код для стековой машины, вычисляющий выражение `a - 2 × b`.

```
push b
push 2
multiply
push a
subtract
```

Код для стековой машины, вычисляющий a - 2 × b

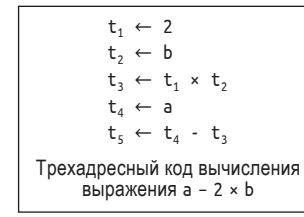
Код для стековой машины компактен. Стек создает неявное пространство имен и исключает многие имена в ПП, что уменьшает размер ПП. Однако использование стека означает, что все результаты и аргументы существуют недолго, если только код явно не переместит их в память.

Байт-код
ПП, спроектированное специально ради компактной формы; обычно это код для абстрактной стековой машины.

Название напоминает об ограниченном размере; многие операции, например умножение, требуют всего одного байта.

Режим использования. Код для стековых машин обычно используется в качестве определяющей ПП – зачастую как ПП для передачи между системами и средами. В SMALLTALK-80 и JAVA в качестве внешней интерпретируемой формы кода используется *байт-код*, ISA стековой виртуальной машины. Байт-код либо исполняется интерпретатором, например виртуальной машиной JAVA, либо транслируется в платформенный код целевой машины перед выполнением. При таком дизайне получается система с компактной формой программы для распространения и простой схемой переноса языка на новую машину: нужно только реализовать интерпретатор для виртуальной машины.

4.4.2. Трехадресный код



Трехадресный код вычисления выражения a – 2 × b

В трехадресном коде большинство операций имеют вид `i ← j op k`, с оператором (`op`), двумя операндами (`j` и `k`) и одним результатом (`i`). Некоторые операторы, например непосредственной загрузки и безусловного перехода, требуют меньше аргументов. Иногда у операции больше трех адресов, например у операции «умножить и сложить» с плавающей точкой. Во врезке приведен трехадресный код вычисления выражения `a - 2 × b`. Язык ILOC – трехадресный код.

Трехадресный код привлекателен по нескольким причинам. Во-первых, он относительно компактен. Большинство операций состоят из четырех элементов: *кода операции* и трех имен. Код операции и имена выбираются из ограниченных множеств. Коды операций обычно занимают один или два байта. Имена, как правило, представляются целыми числами или табличными индексами. Во-вторых, отдельные имена для операндов и результата оставляют компилятору свободу управлять повторным использованием имен и значений; в трехадресном коде нет деструктивных операций. Трехадресный код вводит новое множество сгенерированных компилятором имен – имена, содержащие результаты различных операций. Тщательно продуманное пространство имен может открыть новые возможности улучшения кода. Наконец, поскольку многие современные процессоры реализуют трехадресные операции, трехадресный код хорошо моделирует их свойства.

Уровень абстракции. Для трехадресных кодов множество поддерживаемых операторов и уровень абстракции могут изменяться в широких пределах. Часто трехадресное ПП содержит преимущественно низкоуровневые операции – условные и безусловные переходы, загрузку и сохранение, – а также более сложные операции, инкапсулирующие некоторый поток управления, например `max`. Непосредственное представление таких составных операций упрощает анализ и оптимизацию.

Например, рассмотрим операцию, которая копирует строку литер из источника в приемник. Эта операция включена в виде библиотечной функции `bcopy` в дистрибутив 4.2 BSD UNIX и в виде команды `mvcl` (**m**ove **ch**aracter **l**ong) в ISA IBM 370. На машине, где нет аппаратной команды типа `mvcl`, для выполнения такого копирования может потребоваться много операций.

Добавление `mvcl` в трехадресный код позволяет компилятору компактно представить эту сложную операцию. Компилятор может анализировать, оптимизировать и перемещать операцию, не задумываясь о ее внутреннем механизме работы. Если оборудование поддерживает операции типа `mvcl`, то кодогенератор отобразит конструкцию ПП прямо на аппаратную реализацию. В противном случае компилятор может транслировать

Эта стратегия использовалась в компиляторе IBM PL.8, первом компиляторе для RISC-машины.

mvcl в последовательность низкоуровневых операций ПП или вызвать функцию типа bcopy перед окончательной оптимизацией и генерированием кода.

Режим использования. Компиляторы, в которых используется трехадресный код, обычно принимают его в качестве определяющего ПП. Трехадресный код со своим явным пространством имен и моделью загрузки и сохранения в памяти особенно хорошо подходит для оптимизации на регистровых машинах с командами загрузки и сохранения.

4.4.3. Представление линейных кодов

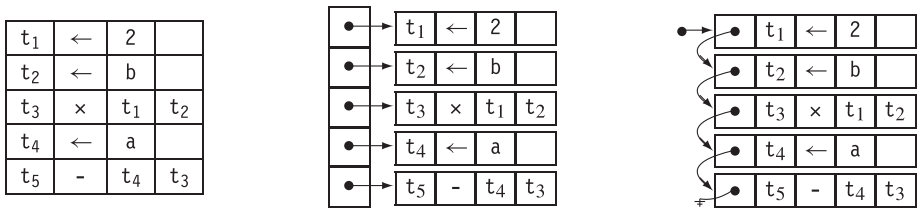
Для реализации линейных ПП применялось много структур данных. От выбора автора компилятора зависит стоимость различных операций в коде ПП. Поскольку компилятор тратит большую часть времени на манипулирование кодом в форме ПП, эти затраты заслуживают внимания. И хотя это обсуждение относится к трехадресным кодам, многие моменты в равной степени применимы к коду для стековых машин (и другим линейным формам).

Трехадресный код часто реализуется как множество четверок. Каждая четверка представлена четырьмя полями: оператор, два операнда (или источника) и результат. Для формирования блоков компилятору нужен механизм соединения отдельных четверок. Компиляторы реализуют четверки разными способами.

$t_1 \leftarrow 2$
$t_2 \leftarrow b$
$t_3 \leftarrow t_1 \times t_2$
$t_4 \leftarrow a$
$t_5 \leftarrow t_4 - t_3$

Трехадресный код для вычисления выражения a - 2 × b

На рис. 4.5 показаны три схемы представления трехадресного кода для вычисления выражения a - 2 × b (приведен во врезке). В первой схеме (рисунок (a)) используется массив структур. Компилятор мог бы построить такой массив для каждой вершины ГПУ и поместить в него код соответствующего блока. На рисунке (b) четверки блока хранятся в векторе указателей, а на рисунке (c) – в связанном списке.



(a) Простой массив (b) Массив указателей (c) Связанный список

Рис. 4.5 Реализации трехадресного кода для вычисления выражения a - 2 × b

Рассмотрим затраты на переупорядочение кода в этом блоке. Первая операция загружает константу в регистр; в большинстве машин это транслируется в операции непосредственной загрузки. Вторая и четвертая операции загружают значения из памяти, что в большинстве машин может повлечь за собой многотактовую задержку, если значения еще не находятся в кеше

первого уровня. Чтобы компенсировать часть задержки, планировщик команд мог бы переместить команды загрузки `b` и `a` перед командой непосредственной загрузки 2.

В случае массива структур перенос загрузки `b` до непосредственной загрузки требует сохранения первой операции во временном месте, сдвига второй операции вверх и переноса команды непосредственной загрузки во вторую позицию. В случае вектора указателей необходим такой же трехшаговый подход – с тем отличием, что изменяются только значения указателей. Компилятор может сохранить указатель на команду непосредственной загрузки, скопировать указатель на команду загрузки `b` в первый элемент вектора и перезаписать второй элемент вектора сохраненным указателем. В случае связанного списка операции похожи, но компилятор должен сохранить состояние, достаточное для прохода по списку.

Теперь рассмотрим, что произойдет в фазе анализа, когда она генерирует начальное ПП. В случае массива структур и вектора указателей компилятор должен выбрать размер массива, т. е. сколько четверок он ожидает в блоке. В процессе генерирования четверок он заполняет структуру данных. Если компилятор выделил слишком много памяти, то она расходуется напрасно. А если слишком мало, то придется выделять память для большего массива или вектора, копировать старое содержимое в новое место и освобождать ранее выделенную память. В случае связанного списка таких проблем не возникает. Для расширения списка нужно лишь выделить память для новой четверки и настроить соответствующий указатель в списке.

Многопроходный компилятор может использовать различные реализации ПП в разных точках процесса компиляции. В фазе анализа, где основное внимание уделяется генерированию ПП, связанный список мог бы упростить реализацию и снизить общую стоимость. В планировщике команд, где важно переупорядочение операций, массив указателей мог бы иметь больше смысла. Различие в деталях реализации можно скрыть за общим интерфейсом.

Промежуточные представления на практике

На практике компиляторы используют самые разные ПП. В легендарных компиляторах FORTRAN минувших дней, например в компиляторе FORTRAN H от IBM, использовалась комбинация четверок и графов потока управления для представления кода перед оптимизацией. Поскольку FORTRAN H был написан на FORTRAN, ПП хранилось в массиве.

На протяжении многих лет GCC полагался на ПП очень низкого уровня, называемое языком межрегистровых передач (register transfer language – RTL). Но затем GCC перешел на использование нескольких ПП. Первоначально парсеры порождают языково-зависимое дерево, близкое к исходному коду. Затем компилятор понижает уровень дерева до второго ПП, GIMPLE, которое включает языково-независимую древовидную структуру для управляющих конструкций и трехадресный код для выражений и присваиваний. В большей части оптимизатора GCC используется GIMPLE; например, GCC строит поверх GIMPLE форму с единственным статическим присваиванием (static single-assignment – SSA). В конечном итоге GCC транслирует GIMPLE в RTL для окончательной оптимизации и генерирования кода.

В компиляторе LLVM используется одно низкоуровневое ПП; на самом деле само название LLVM расшифровывается как «low-level virtual machine» (низкоуровневая виртуальная машина). ПП в LLVM – это линейный трехадресный код. ПП полностью типизировано и располагает явной поддержкой адресации массивов и структур. Оно поддерживает векторные (SIMD) данные и операции. Скалярные значения хранятся в форме SSA до тех пор, пока код не дойдет до фазы синтеза. В тех средах LLVM, где используются фазы анализа из GCC, ПП порождается проходом, который выполняет трансляцию GIMPLE в LLVM.

В компиляторе с открытым исходным кодом Open64 для архитектуры IA-64 использовалось семейство из пяти взаимосвязанных ПП, называемое WHIRL. Парсер порождал WHIRL, близкое к уровню исходного кода. Последующие фазы компилятора привносили в WHIRL-код больше деталей, понижая уровень абстракции до фактического машинного кода. Эта схема позволяла компилятору подстраивать уровень абстракции к различным оптимизациям, применявшимся в ПП.

Заметим, что на рис. 4.5 отсутствует кое-какая информация. Например, не показаны метки, потому что метки – свойство блока, а не отдельной четверки. Хранение списка меток в вершине ГПУ, ассоциированной с блоком, экономит место в каждой четверке; также явно видно, что метки могут встречаться только в начале блока. Коль скоро метка привязана к блоку, компилятор может игнорировать их, когда занимается переупорядочением операции внутри блока, – одним осложнением меньше.

4.4.4. Построение ГПУ из линейного кода

Компиляторы нередко должны преобразовывать одно ИП в другое, зачастую меняя стиль ПП. Одно из таких стандартных преобразований – построение ГПУ из линейного ПП, например ILOC-кода. Существенные особенности ГПУ – то, что он идентифицирует начало и конец каждого простого блока и соединяет получившиеся блоки ребрами, которые описывают возможные передачи управления между блоками. Часто компилятор должен строить ГПУ из простого линейного ПП, представляющего процедуру.

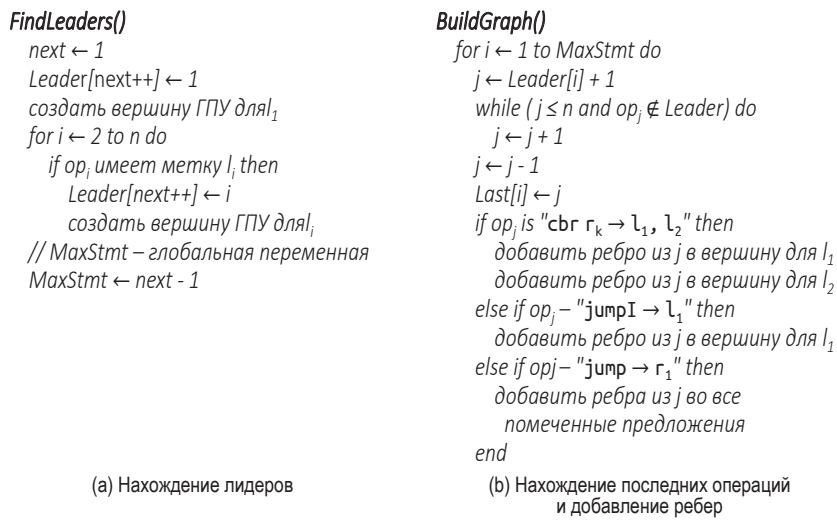
На первом шаге компилятор должен найти начало и конец каждого простого блока в линейном ПП. Будем называть начальную операцию блока *лидером*. Операция является лидером, если это первая операция в процедуре или если она имеет метку, т. е. потенциально является целью некоторого перехода. Компилятор может идентифицировать лидеров на первом проходе по ПП, показанному на рис. 4.6(а). Функция *FindLeaders* обходит операции по порядку, находит помеченные предложения и запоминает их в качестве лидеров.

Неопределенный переход
условный или безусловный переход, цель которого неизвестна на этапе компиляции (например, переход по адресу в регистре).

В ILOC переход `jump` неопределенный, а `jumpI` – нет.

Если линейное ПП содержит метки, которые не являются целями команд перехода, то рассмотрение меток как лидеров может без необходимости расщеплять блоки. Алгоритм мог бы отслеживать, какие метки

являются целями переходов. Однако неопределенные переходы могут заставить его трактовать все помеченные предложения как лидеров.



(a) Нахождение лидеров

(b) Нахождение последних операций и добавление ребер

Рис. 4.6 Построение графа потока управления

На втором проходе, показанном на рисунке (b), находятся все операции, завершающие блок. Предполагается модель ILOC, в которой каждый блок, кроме последнего, завершается условным или безусловным переходом, а в командах условного перехода задаются метки выбранной и неявной ветви. Это предположение упрощает обработку блоков и позволяет оптимизатору или фазе синтеза выбирать, какая ветвь будет неявной. (Пока будем предполагать, что в командах условного перехода нет слотов отсрочки.)

> Построение ГРУ с неявными ветвями оставляем читателю в качестве упражнения (см. упражнение 4).

Чтобы найти конец каждого блока, алгоритм обходит блоки в порядке их появления в массиве *Leader*. Он проходит вперед по ПП, пока не найдет лидера следующего блока. Операция, предшествующая лидеру, завершает текущий блок. Алгоритм запоминает индекс этой операции в *Last[i]*, так что пара ⟨*Leader[i], Last[i]*⟩ описывает блок *i*. При необходимости в ГПУ добавляются ребра.

По разным причинам, ГПУ должен иметь единственную входную вершину n_0 и единственную выходную вершину n_f. Если форма исходного кода не такая, то простой дополнительный проход по графу может создать n_0 и n_f.

Затруднения при построении ГПУ

Особенности ПП, целевой машины и исходного языка могут осложнить построение ГПУ.

Неопределенные переходы могут вынудить компилятор добавлять ребра, ненужные на этапе выполнения. Автор компилятора может исправить эту ситуацию, запоминая потенциальные цели неопределенных переходов в ПП.

Псевдооперация
операция, которая манипулирует внутренним состоянием ассемблера или компилятора, но не транслируется в исполняемую операцию.

ILOC включает псевдооперацию `tbl`, которая задает возможные цели неопределенного перехода (см. приложение А). Всякий раз, как компилятор генерирует команду `jump`, он должен сгенерировать вслед за ней одну или несколько операций `tbl`, в которых запоминаются возможные цели. Такие указания уменьшают количество лишних ребер в процессе построения ГПУ.

Относительный переход
команда передачи управления, в которой задается смещение, положительное или отрицательное, относительно ее собственного адреса в памяти.

Если инструмент строит ГПУ по машинному коду, то особенности целевой ISA могут осложнить процесс. Алгоритм на рис. 4.6 предполагает, что все лидеры, кроме первого, помечены. Если в целевой машине имеются неявные ветви, то алгоритм необходимо дополнить, так чтобы он распознавал непомеченные предложения, получающие управление на неявном пути. Относительные переходы вызывают похожие осложнения.

Слоты отсрочки переходов также привносят сложности. Компилятор должен сгруппировать все операции в слоте отсрочки в блок, который предшествует команде условного или безусловного перехода. Если такая операция имеет метку, то она является членом нескольких блоков. Чтобы устранить неоднозначность, компилятор может поместить непомеченную копию операции в слот отсрочки, а помеченную операцию использовать, чтобы начать новый блок.

Если команда условного или безусловного перехода может встретиться в слоте отсрочки, то построитель ГПУ должен пройти вперед от лидера и найти переход, завершающий блок, – первый переход, который встретится. Переходы в слоте отсрочки операции условного перехода, завершающей блок, могут быть отложены при входе в целевой блок. По существу, они могут разбивать целевой блок на несколько блоков и создавать как новые блоки, так и новые ребра. Эта особенность серьезно осложняет построение ГПУ.

Некоторые языки допускают безусловные переходы на метки, расположенные вне текущей процедуры. В процедуре, содержащей такой переход, цель перехода можно смоделировать с помощью нового блока. Однако в процедуре, содержащей цель, помеченный блок может создать проблему. Компилятор должен знать, что метка является целью нелокального перехода; в противном случае аналитические проходы могут дать вводящие в заблуждение результаты. По этой причине в таких языках, как PASCAL или ALGOL, нелокальные переходы ограничены метками, видимыми во внешних лексических областях видимости. Язык C требует использовать функции `setjmp` и `longjmp`, чтобы сделать такие передачи управления явными.

ИТОГИ РАЗДЕЛА

Линейные ПП представляют компилируемый код в виде упорядоченной последовательности операций. Линейные ПП различаются по уровню абстракции; исходный код программы в текстовом файле записан в линейной форме, как и ассемблерный код той же программы. Линейные ПП позволяют создавать компактные, понятные человеку представления.

Два широко употребительных линейных ПП – байт-код, обычно реализуемый как одноадресный код с неявными именами во многих операциях, и трехадресный код, похожий на ILOC.

ВОПРОСЫ

1. Рассмотрите выражение a × 2 + a × 2 × b. Оттранслируйте его в код для стековой машины и в трехадресный код. Сравните общее число операций и операндов в обеих формах. Как они соотносятся с деревом на рис. 4.2(b)?
2. Обрисуйте изменения, которые необходимо внести в алгоритм на рис. 4.6 для учета неопределенных условных и безусловных переходов. Если все условные и безусловные переходы пометить конструкцией, аналогичной `tbl` в ILOC, упростит ли это алгоритм?

4.5. Таблицы символов

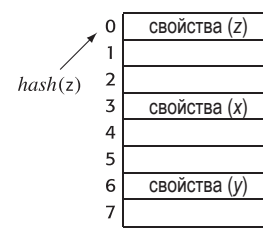
Таблица символов
совокупность структур данных, в которых хранится информация об именах и значениях.

Большинство компиляторов хранят таблицы символов в постоянных вспомогательных структурах данных, которые используются в сочетании с ПП, представляющим исполняемый код.

В процессе синтаксического разбора компилятор выявляет имена и свойства многих сущностей, в т. ч. именованных значений, таких как переменные, массивы, записи, структуры, строки и объекты, а также определения классов, константы, метки в коде и сгенерированные компилятором временные значения (см. врезку на стр. 210).

Для каждого имени, используемого в программе, компилятору нужно знать разнообразную информацию, чтобы сгенерировать код манипулирования соответствующей сущностью. Состав информации зависит от вида сущности. Для простой скалярной переменной компилятору, возможно, достаточно знать тип данных, размер и место размещения в памяти. Для функции нужен тип и размер каждого параметра, тип и размер возвращаемого значения и перемещаемая метка точки входа в функцию в коде на языке ассемблера.

Таким образом, компилятор обычно создает один или несколько репозиториев для хранения описательной информации и предоставляет эффективный способ нахождения информации, связанной с данным именем. Эффективность особенно важна, потому что компилятору приходится обращаться к этим репозиториям многократно.



Обычно компилятор хранит набор таблиц, часто называемых *таблицами символов*. Концептуально таблица символов состоит из двух главных компонентов: отображение текстового имени на индекс в репозитории и сам репозиторий, связывающий с индексом свойства имени. Во врезке показано абстрактное представление такой таблицы.

Компилятор может использовать несколько таблиц для представления различных видов информации о различных видах значений. Для имен ему нужна таблица символов, которая отображает каждое имя на его свойства, объявленные или неявные. Для агрегатов, таких как записи, массивы и объекты, компилятору необходима таблица структур, в которой хранится информация об организации сущности: ее членах или полях, их свойствах, а также относительном местоположении в структуре. Что касается литеральных констант, таких как числа, литеры и строки, то компилятор должен отвести область данных для хранения их значений; эту область часто называют пулом констант. Также он должен сопоставить каждой литеральной константе ее тип и смещение от начала пула.

Пул констант
статически инициализированная область данных, отведенная для хранения постоянных значений.

4.5.1. Разрешение имен

Основная цель таблицы символов – разрешение имен. Если компилятор находит ссылку на имя *n* в какой-то точке программы *p*, то ему необходим механизм, который отображает *n* на его объявление в том пространстве имен, которое действует в точке *p*. Отображение имени на его объявление и свойства должно быть корректно определено, в противном случае смысл программы может оказаться неоднозначным. Поэтому языки программирования вводят правила, определяющие, где данное объявление имени видимо и допустимо.

Область видимости
область программы, в которой к данному имени можно обратиться.

В общем случае область видимости представляет собой непрерывную последовательность предложений, в которой имя объявлено, видимо и доступно. Границы области видимости отмечаются специальными символами языка. Как правило, новая процедура определяет новую область видимости, которая охватывает все ее определение. В языках C и C++ блоки ограничены фигурными скобками. Каждый блок определяет новую область видимости.

В большинстве языков области видимости могут быть вложенными. Объявление *x* во внутренней области видимости маскирует все определения *x* в объемлющих областях. Вложенность создает иерархию пространств имен. Такие иерархии играют очень важную роль в программной инженерии, они позволяют программисту выбирать локальные имена, не думая о том, что имя может встречаться еще где-то в программе.

Представление ссылок в ПП

При реализации ПП автор компилятора должен решить, как представлять ссылки на имена исходного языка. Компилятор мог бы просто запоминать лексему, но это решение потребовало бы поиска в таблице символов при каждом использовании ссылки.

Самый лучший вариант – хранить описатель ссылки на запись таблицы символов. Описатель мог бы быть абсолютным указателем на запись или указателем на таблицу плюс смещение от ее начала. Такой описатель позволяет напрямую обращаться к информации в таблице символов; он также должен допускать дешевое сравнение на равенство.

Две самые известные иерархии пространств имен создаются правилами лексических областей видимости и правилами наследования. Компилятор должен строить таблицы для моделирования обеих этих иерархий.

Лексические области видимости

Лексическими областями видимости являются вложенные области кода. Имя *n*, объявленное в области видимости *s*, видимо внутри *s*. Оно также видимо во всех областях, вложенных в *s*, с той оговоркой, что новое объявление *n* маскирует объявления *n* в любой объемлющей области.

В точке кода *p* ссылка на *n* отображается на первое объявление *n*, найденное путем прохода по областям видимости, начиная с той, что содержит ссылку, и далее вверх по иерархии до *глобальной области видимости*. Языки с лексическими областями видимости различаются по максимально допустимой глубине вложенности и множеству предоставляемых областей видимости. (В разделах 5.4.1, 6.3.1 и 6.4.3 лексические области видимости обсуждаются более подробно.)

Глобальная область видимости
самая внешняя область видимости для имен, видимых в любой точке программы.

Иерархии наследования

В объектно-ориентированных языках (ООЯ) вводится еще одно множество областей видимости: иерархия наследования. Создается схема именования объектов, ориентированная на данные; объекты содержат члены данных и кода, доступ к которым осуществляется относительно объекта, а не относительно текущей процедуры.

В ООЯ явно объявленные отношения между подклассом и суперклассом определяют иерархию наследования – режим именования, аналогичный лексической иерар-

Суперкласс и подкласс
В языке с наследованием если класс *x* наследует члена и свойства от класса *y*, то мы говорим, что *x* является подклассом *y*, а *y* является суперклассом *x*.

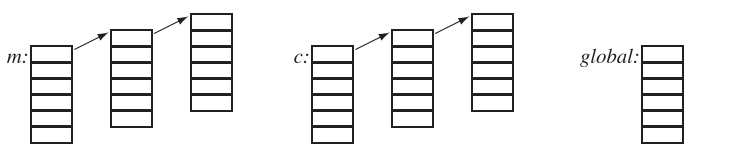
Терминология, применяемая для задания наследования, в разных языках различна. В JAVA говорят, что подкласс *расширяет* свой суперкласс, а в C++ – что подкласс является *производным* от суперкласса.

хии и ортогональный ей. Концептуально подклассы вложены в суперклассы, так же как внутренние области видимости вложены во внешние в лексической иерархии. Компилятор создает таблицы для моделирования отношений между подклассами и суперклассами.

Иерархические таблицы

Компилятор может связать таблицы, построенные для разных областей видимости, и таким образом представить иерархии именования в любой входной программе. Типичная программа на Algol-подобном языке могла бы иметь один набор связанных таблиц для представления лексически вложенных областей видимости. В ООЯ эта лексическая иерархия была бы дополнена другим набором связанных таблиц для представления иерархии наследования.

Когда компилятор встречает ссылку в коде, он первым делом решает, относится ли она к переменной (глобальной или локальной в каком-то методе) или к члену объекта. Обычно это чисто синтаксическое решение: языки проводят различие между ссылками на переменные и на объекты. Поиск ссылки на переменную начинается в таблице для текущей области видимости и продолжается вверх по иерархии, пока ссылка не будет найдена. Что до члена объекта, компилятор определяет класс объекта и начинает поиск в иерархии наследования. В методе *m*, объявленном в классе *c*, поисковое окружение может выглядеть, как показано ниже. Такую схему мы называем «связкой таблиц».



Поиск начинается в таблице для подходящей области видимости и продолжается вверх по цепочке таблиц, пока имя не будет найдено или цепочка не закончится. Такое связывание таблиц в цепочку дает гибкий и мощный инструмент для моделирования сложных правил видимости, характерных для современного языка программирования.

Автор компилятора может моделировать сложные иерархии областей видимости – как лексических, так и наследования – с помощью нескольких таблиц, связанных между собой способом, отражающим порядок поиска, принятый в языке. Например, вложенные классы в JAVA образуют лексическую иерархию внутри иерархии наследования. Порядок связывания может зависеть от языка, но базовая техника не изменяется.

В современных средах компилятор, скорее всего, сохранит все эти таблицы для последующего использования – либо в самом компиляторе, либо в сопутствующих инструментах, таких как мониторы производительности или

отладчики. Подход на основе связки таблицы позволяет создавать компактные отделимые таблицы, пригодные для сохранения.

Другие области видимости

Вложенные области видимости создаются и для поддержки других возможностей языка. Например, записи, структуры и объекты ведут себя как независимые области видимости. Каждая из этих сущностей создает новое пространство имен и определяет набор правил видимости. Компилятор может реализовать эти области видимости с помощью полноценной связки в хеш-таблице, но может этого и не делать; так или иначе, каждая из них является отдельной областью видимости, и ее можно смоделировать с помощью новой связки в подходящей таблице. Пул констант также можно рассматривать как отдельную область видимости.

4.5.2. Реализация таблиц

Как мы видели, компилятор содержит несколько таблиц: от таблиц символов и таблиц наследования до таблиц размещения структур в памяти и таблиц констант. Для каждой таблицы автор компилятора должен выбрать подходящую стратегию реализации: функцию отображения и репозиторий. Хотя решения по большей части независимы, автор компилятора, возможно, захочет использовать одни и те же базовые стратегии для нескольких таблиц, чтобы реализацией можно было воспользоваться повторно.

Реализация отображения

Отображение текстового имени на индекс можно реализовать тысячей способов, и у каждого будут свои плюсы и минусы.

Линейный список

Просто строить, расширять и осуществлять в нем поиск. Основной его недостаток – время поиска, составляющее в среднем $O(n)$, где n – размер списка. Тем не менее для небольшой процедуры использование линейного списка может быть оправдано.

Дерево

Древовидная структура обладает всеми преимуществами списка, включая простое и эффективное расширение. Дополнительно оно может заметно уменьшить время поиска. В предположении, что дерево приблизительно сбалансировано, т. е. размер поддеревьев на каждом уровне примерно одинаков, – ожидаемое время поиска одного элемента будет близко к $O(\log_2 n)$, где n – число элементов в дереве.

Для поддержания сбалансированности требуются более сложные протоколы вставки и удаления элементов. В литературе описывается много эффективных приемов построения сбалансированных деревьев.

Для несбалансированных деревьев протоколы вставки и удаления проще, но они не дают никаких гарантий относительного размера поддеревьев на одном уровне. Поиск в несбалансированном дереве может выродиться в линейный, если противник подал на вход специально подобранные данные.

Хеш-таблица

Коллизия хеша
Когда две строки отображаются в один и тот же индекс, говорят, что имеет место *коллизия*. Для хеш-функции $h(x)$ и размера таблицы s если

$$h(x) \bmod s = h(y) \bmod s,$$

то между x и y имеется коллизия.

Компилятор может выполнить вычисление, называемое хешированием, сопоставляющее строке целое число. Хорошо спроектированная функция хеширования, или хеш-функция, h распределяет эти целые числа так, что лишь немногие строки хешируются в одно и то же число. Для построения хеш-таблицы программист берет в качестве индекса хеш-значение строки по модулю размера таблицы.

Обработка коллизий – ключевой момент проектирования хеш-таблицы, он обсуждается в приложении В.4. Если множество ключей не порождает коллизий, то вставка в хеш-таблицу и поиск в ней должны занимать время O(1). Если все ключи отображаются в один и тот же индекс, то вставка и поиск могут деградировать и занимать время O(|*keys*|), как в случае линейного списка. Чтобы избежать такой проблемы, автор компилятора должен использовать хорошо спроектированную хеш-функцию; их можно найти в любой хорошей книге по алгоритмам.

Статическое отображение

В качестве альтернативы хешированию компилятор может заранее вычислить статическое отображение ключей в индексы, не имеющее коллизий. Эту проблему решает разделение мультимножества (см. врезку на стр. 208).

Для небольших множеств ключей подход, при котором ключи рассматриваются как множество ациклических регулярных выражений и инкрементно строится ДКА для распознавания этого множества, может привести к быстрой реализации (см. раздел 2.6.2). Но после того как размер таблицы переходов оказывается больше размера кеша первого уровня, скорость работы значительно падает.

Реализация репозитория

Реализация репозитория для хранения информации, ассоциированной с данным именем, может быть ортогональна механизму поиска. Для разных таблиц могут понадобиться разные и раздельные структуры, специально приспособленные для хранения той информации, которая нужна компилятору. Тем не менее у всех таких репозиториев должны быть некоторые общие свойства (см. приложение В.4).

- Память для записей должна быть либо непрерывной, либо блочно-непрерывной, чтобы улучшить локальность, уменьшить стоимость выде-

ления и упростить чтение таблиц и их запись на внешний носитель.

- Репозиторий должен содержать достаточно информации, чтобы заново построить поисковую структуру; это необходимо для постепенного расширения таблицы и для восстановления структур с внешнего носителя.
- Репозиторий должен поддерживать изменение пути поиска. Например, по мере того как парсер входит в различные области видимости и выходит из них, путь поиска должен изменяться, отражая текущую ситуацию.

Блочно-непрерывная память
В таких распределителях используются два протокола: крупное выделение получает память для нескольких записей, а мелкое возвращает одну запись. Мелкие выделения производятся быстро; в итоге стоимость крупного выделения раскладывается на много записей.

С точки зрения управления, репозиторий должен быть расширяемым, чтобы можно было эффективно работать с большими программами и в то же время не расходовать место впустую при работе с малыми. Схема его индексирования не должна зависеть от схемы отображения, так чтобы отображение могло расширяться независимо от репозитория; в идеале отображение должно быть разреженным, а репозиторий – плотным.

ИТОГИ РАЗДЕЛА

Компиляторы строят вспомогательные структуры данных, дополняющие информацию, которая хранится в определяющем ПП. Из этих структур лучше всех видно множество таблиц символов, которые отображают имя в исходном коде или ПП на множество ассоциированных с ним свойств.

В этом разделе было рассмотрено несколько вопросов проектирования таких вспомогательных таблиц. Показано, как связывание таблиц с образованием явных путей поиска может помочь в моделировании правил лексических областей видимости и наследования. Обсуждались компромиссы при реализации механизма отображения и репозитория для этих таблиц.

ВОПРОСЫ

1. Какова сложность вставки нового имени в таблицу для текущей области видимости при использовании схемы «связки таблиц»? Какова сложность поиска имени в наборе таблиц? Насколько глубоки иерархии лексических областей видимости и наследования в написанных вами программах?
2. Когда компилятор инициализирует область видимости, ему, вероятно, необходим начальный размер таблицы символов. Как парсер мог бы оценить этот размер? Как могли бы оценить его последующие проходы компилятора?

4.6. Пространства имен

В большинстве обсуждений пространств имен наибольшее внимание уделяется пространству имен исходной программы: правилам лексических областей видимости и наследования. Но не менее важно, с точки зрения качества откомпилированного кода, пространство имен, создаваемое в ПП компилятора. Выбранная дисциплина именования может либо открыть, либо похоро-

нить возможности для оптимизации. Решения, принимаемые компилятором в части имен, в значительной степени определяют, какие вычисления можно анализировать и оптимизировать.

Альтернатива хешированию

Таблицы символов часто реализуются с помощью хеш-таблиц в силу ожидаемой эффективности хеширования. Если автор компилятора обеспокоен маловероятным, но все же возможным поведением хеширования в худшем случае, то интересную альтернативу может составить разделение мультимножества. Оно позволяет избежать поведения в худшем случае ценой специального «офлайнового» прохода для построения индекса.

Чтобы воспользоваться разделением мультимножества, компилятор сначала просматривает всю программу и строит кортеж ⟨*name*, *pos*⟩ для каждого экземпляра идентификатора, где *name* – лексема идентификатора, а *pos* – номер ее позиции в списке классифицированных слов, или *токенов*. Все кортежи помещаются в большое множество.

Затем компилятор сортирует это множество лексикографически. По сути дела, он создает множество подмножеств, по одному на каждый идентификатор. В каждом подмножестве находятся кортежи для всех вхождений данного идентификатора. Поскольку каждый кортеж ссылается на определенный токен по его *позиции*, компилятор может использовать отсортированное множество для модификации потока токенов. Компилятор линейно проходит по множеству, обрабатывая каждое подмножество. Он назначает каждому уникальному идентификатору индекс в таблице символов, а затем переписывает токены, включая в них индексы. Парсер может прочитать индексы в таблице символов прямо из токенов. Если компилятору нужна функция поиска по тексту, то результирующая таблица упорядочивается по алфавиту для двоичного поиска.

Эта техника несколько увеличивает стоимость компиляции. Необходим дополнительный проход по потоку токенов, а также лексикографическая сортировка. Но взамен мы устраняем саму возможность худшего случая при хешировании, а начальный размер таблицы символов становится очевиден еще до начала синтаксического разбора. Таким образом, мы можем заменить хеш-таблицу почти во всех приложениях, где офлайновое решение будет работать.

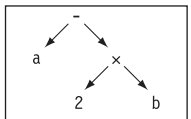
Пространство имен ПП тесно связано с моделью памяти, используемой при трансляции. Компилятор может предполагать, что все значения хранятся в памяти, за исключением тех случаев, когда они активно используются в вычислении. Компилятор может предполагать, что значения хранятся в регистрах, когда это возможно. Если компилятор пользуется кодом для стековой машины в качестве ПП, то эти активные значения будут храниться в стеке. Эти предположения радикально влияют на то, какое множество значений может быть поименовано, проанализировано и оптимизировано.

В этом разделе мы сосредоточимся на вопросах проектирования пространства имен. Также мы познакомимся с одним важным примером: *формой с единственным статическим присваиванием* (static single-assignment – SSA). В следующем разделе рассматриваются вопросы, возникающие в связи с выбором модели памяти.

4.6.1. Пространства имен в ПП

Проектируя ПП, автор компилятора должен также уделить внимание дисциплине именования. Выбор пространства имен неотделим от выбора ПП; некоторые ПП допускают широкую свободу выбора имен, тогда как в других большинство имен неявно скрыто в представлении.

Явные и неявные имена



АСД для a - 2 × b

В древовидных ПП для некоторых значений используются неявные имена. Рассмотрим АСД для выражения a - 2 × b, показанного во врезке. В нем имеются узлы для a, 2, b, 2 × b и a - 2 × b. У внутренних узлов, 2 × b и a – 2 × b, нет явных имен, с которыми компилятор мог бы иметь дело.

С другой стороны, в трехадресном коде используются только явные имена, что дает компилятору контроль над дисциплиной именования. Он может присвоить явные имена всем или только некоторым значениям, вычисляемым в коде. Рассмотрим, к примеру, ILOC-код выражения a - 2 × b, показанный во врезке. В верхней версии уникальные имена присваиваются всем неизвестным значениям и выражениям – имена регистров от r_0 до r_3. После выполнения каждое значение остается в своем регистре. В нижней версии используется другая дисциплина именования, рассчитанная на экономию имен. После выполнения остаются только величины: a в r_0 и a - 2 × b в r_1.

```
load   @b      ⇒ r0
multI  2, r0   ⇒ r1
load   @a      ⇒ r2
subI   r2, r1  ⇒ r3
```

ILOC-код для a - 2 × b
с уникальными именами

```
load   @b      ⇒ r0
multI  2, r0   ⇒ r1
load   @a      ⇒ r0
subI   r0, r1  ⇒ r1
```

ILOC-код для a - 2 × b
с повторно используемыми именами

Из-за этого примера может показаться, что в графовых ПП используются неявные имена, а в линейных – явные. Но все не так просто. Код для стековых машин опирается на неявную структуру стека, из-за чего многие значения имеют неявные имена. В ГПУ все узлы имеют явные имена, чтобы к ним можно было присоединить соответствующие фрагменты кода. Даже АСД можно переписать, включив явные имена; для любого выражения или подвыражения, которому компилятор хочет назначить имя, он может вставить присваивание и далее использовать сгенерированное им имя.

Переменные и значения

В исходной программе множество доступных имен определяется правилами языка и деталями программы. Объявленные именованные переменные видимы, промежуточные подвыражения – нет. Возьмем предложение

```
a ← 2 * b + cos(c / 3)
```

Здесь a, b и c можно использовать в последующих предложениях, а значения подвыражений 2 * b, c / 3 и cos(c / 3) нельзя.

Виртуальное имя
Сгенерированное компилятором имя часто называют *виртуальным* в том же смысле, в каком говорят о *виртуальной памяти* или *виртуальном регистре*.

Автор компилятора должен решить, какие значения сделать видимыми в ПП, присвоив им единообразные явные имена. Компилятор может использовать столько имен, сколько необходимо; авторы компиляторов часто называют такие сгенерированные имена *виртуальными*. Компилятор мог бы транслировать приведенное выше предложение, присвоив всем трем подвыражениям уникальные имена. С другой стороны, используя имена повторно, он мог бы поставить крест на попытках использования значения в будущем. Это решение напрямую влияет на то, что может делать компилятор в процессе последующей оптимизации.

Влияние схемы именования

В конце 1980-х годов мы экспериментировали со схемами именования в компиляторе FORTRAN 77. Первая версия генерировала новое имя для каждого вычисления; для этого просто увеличивался на единицу счетчик. Такой подход порождал большие пространства имен; например, для реализации алгоритма сингулярного разложения (SVD), насчитывавшего 210 строк, было создано 985 имен. Объективно кажется, что это слишком много. В итоге – проблемы с быстродействием и потреблением памяти в распределителе регистров, где размер пространства имен определяет размеры многих структур данных. (Сегодня в нашем распоряжении более совершенные структуры данных и гораздо более быстрые машины с большим объемом памяти.)

Во второй версии для управления именами использовался протокол выделения-освобождения. В фазе анализа временные имена создавались по запросу, а освобождались, когда в них отпадала необходимость. При такой схеме пространство имен значительно сократилось; для SVD требовалось примерно 60 имен. Выделение памяти ускорилось; например, время вычисления множества LiveOut для SVD уменьшилось на 60 % (см. раздел 8.6.1).

К сожалению, повторное использование имен маскировало поток значений и снижало качество оптимизации. Снижение качества кода затмило весь выигрыш на этапе компиляции.

Дальнейшие эксперименты привели к короткому набору правил, обеспечивавших хорошую оптимизацию вкупе с сокращением пространства имен.

1. Каждое текстовое выражение получало уникальное имя, которое формировалось с помощью хеширования. Таким образом, все вхождения одного и того же выражения, например $r_{17} + r_{21}$, связывались с одним и тем же регистром.
2. В операции ⟨op⟩ $r_i, r_j \Rightarrow r_k$ k выбиралось так, чтобы i, j < k.
3. Операция копирования регистров, $r_i \Rightarrow r_j$ с i>j, была разрешена, только если r_j соответствовало объявленной скалярной переменной. Регистры для переменных определялись только операциями копирования. Результат вычисления выражения записывался в его «естественный» регистр, а затем перемещался в регистр, выделенный для переменной.
4. За любой операцией сохранения, $r_i \Rightarrow$ MEM(r_j), следовало копирование из r_i в именованный регистр переменной. (Правило 1 гарантирует, что загрузка из этого адреса памяти всегда производится в один и тот же регистр. Правило 4 гарантирует, что виртуальный регистр и адрес в памяти содержат одно и то же значение.)

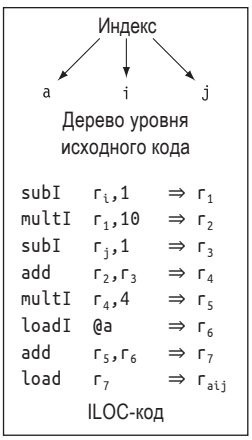
При таком пространстве имен компилятор использовал около 90 имен для реализации SVD и поддерживал при этом все оптимизации, возможные для первой схемы пространства имен. Эти правила действовали, пока мы не перешли на пространство имен SSA.

Автор компилятора следует этим решениям, кодифицировав их в правилах трансляции, на основе которых генерируется ПП. Эти решения влияют на эффективность как самого компилятора, так и генерируемого им кода.

Конечно, возникает искушение назначить уникальное имя каждому подвыражению, чтобы максимизировать возможности, предоставляемые оптимизатору. Однако не все возможности представляют интерес. Значению, встречающемуся только в одном блоке, не нужно имя, существующее на протяжении всей процедуры. Даже если алгоритм анализа и оптимизации процедуры будет видеть такое значение, код от этого вряд ли изменится.

Обратное тоже верно. Любое значение, которое живо или вычисляется в нескольких блоках, заслуживает явного постоянного имени. Выражения, вычисляемые в нескольких блоках, на нескольких путях, – главные цели целого ряда классических глобальных оптимизаций. Наличие одного имени в нескольких определениях и использованиях может сделать выражение доступным для анализа и преобразований, которые улучшают откомпилированный код (см. главы 8–10).

Наконец, выбор дисциплины именования тесно связан с решениями об уровне абстракции ПП. Снова рассмотрим два представления ссылки на элемент массива, `a[i,j]`, показанные во врезке. АСД уровня исходного кода вкупе с таблицей символов содержит всю информацию, необходимую для анализа и трансляции ссылки. (В таблице символов будет храниться тип, область данных и смещение от начала этой области, а также количество размерностей массива и их верхние и нижние границы.) Соответствующий ILOC-код раскрывает больше деталей о вычислениях адреса и назначает явные имена каждому подвыражению, встречающемуся в вычислении.



Эти два представления раскрывают и именуют различные величины. АСД делает явным тот факт, что вычисляется адрес `a[i,j]`, но скрывает детали этого вычисления. ILOC-код раскрывает тот факт, что вычисление адреса состоит из семи разных подвыражений, каждое из которых могло бы встречаться еще где-то. Какое ПП лучше, зависит исключительно от того, как автор компилятора намеревается использовать информацию.

4.6.2. Форма с единственным статическим присваиванием

SSA-форма
ПП, в котором система имен, основанная на значениях, создана путем переименования и использования псевдоопераций, называемых φ-функциями.

SSA кодирует и поток управления, и поток значений. Она широко используется в оптимизации (см. раздел 9.3).

Форма с единственным статическим присваиванием (SSA) – это ПП и дисциплина именования, которая во многих современных компиляторах используется для кодирования информации и о потоке управления, и о потоке значений в программе. В SSA-форме каждому имени соответствует одна точка определения в коде. Термин «с единственным статическим присваиванием» отражает этот факт. Отсюда следует, что каждое имя в операции кодирует информацию о месте происхождения значения; текстовое имя ссылается на конкретную точку определения.

φ-функция
φ-функция принимает несколько имен и объединяет их, определяя новое имя.

Говорят, что программа представлена в SSA-форме, если она удовлетворяет двум условиям: (1) каждое определение имеет уникальное имя и (2) каждое использование ссылается на одно определение. Чтобы преобразовать ПП программы в SSA-форму, компилятор вставляет *φ-функции* в местах, где различные потоки управления сливаются, а затем переименовывает переменные, так чтобы выполнялось свойство единственного присваивания.

Чтобы пояснить эти правила, рассмотрим небольшой цикл на рис. 4.7(a). На рис. 4.7(b) тот же код показан в SSA-форме. Имена переменных включают индексы, чтобы в каждом определении имена различались. В точках, где начала блока могут достичь нескольких разных значений, вставлены φ-функции. Наконец, конструкция `while` переписана на более низком уровне абстракции, чтобы сделать наглядным тот факт, что начальная проверка ссылается на x_0, тогда как проверка в конце цикла ссылается на x_2.

Форма кода для цикла обсуждается в разделе 7.5.2.

```
x ← ...
y ← ...
while (x < 100)
  x ← x + 1
  y ← y + x
```

(a) Оригинальный код

```
      x0 ← ...
      y0 ← ...
      if (x0 ≥ 100) goto next
loop: x1 ← φ(x0,x2)
      y1 ← φ(y0,y2)
      x2 ← x1 + 1
      y2 ← y1 + x2
      if (x2 < 100) goto loop
next: x3 ← φ(x0,x2)
      y3 ← φ(y0,y2)
```

(b) Код в SSA-форме

Рис. 4.7 Небольшой цикл в SSA-форме

Семантика φ-функции необычна. Она работает как операция копирования, выбирающая значение того из своих аргументов, что соответствует ребру, по которому управление вошло в блок. Таким образом, когда поток управления входит в блок из блока над циклом, φ-функции в начале тела цикла копируют значения x_0 и y_0 в x_1 и y_1 соответственно. Когда же поток управления входит в цикл из проверки в конце цикла, φ-функции выбирают другие свои аргументы, x_2 и y_2.

Семантика выполнения φ-функций не такая, как у других операций. При входе в блок все его φ-функции параллельно читают значения подходящих аргументов. Затем все они параллельно определяют свои целевые SSA-имена. Такое определение поведения позволяет алгоритмам, которые манипулируют SSA-формой, игнорировать порядок φ-функций в начале блока – важное упрощение. Однако процесс трансляции SSA-формы обратно в исполняемый код при этом усложняется (см. раздел 9.3.5).

Определение SSA-формы препятствует определению двумя φ-функциями одного и того же SSA-имени.

SSA-форма была придумана для анализа и оптимизации. Место размещения φ-функций в SSA-форме кодирует информацию о создании и использовании индивидуальных значений. Свойство пространства имен, заключающееся в единственности присваивания, позволяет компилятору игнорировать многие проблемы, связанные со временем жизни значений; например, поскольку имена не переопределяются, значение имени доступно на любом пути, проходящем через эту операцию. Эти два свойства упрощают и улучшают многие оптимизации.

Время жизни
Время жизни значения `a` простирается от его первого определения до его последнего использования.

Построение SSA-формы

Форма с единственным статическим присваиванием – единственное из описываемых нами ПП, для которого нет очевидного алгоритма построения. В разделе 9.3 один такой алгоритм описан подробно. Однако набросок процедуры построения поможет прояснить некоторые вопросы. Предположим, что входная программа уже записана в форме ILOC-кода. Чтобы преобразовать ее в эквивалентную линейную SSA-форму, компилятор должен сначала вставить φ-функции, а затем переименовать виртуальные регистры ILOC.

Самый простой способ вставить φ-функции – найти все блоки, имеющие несколько предшественников в ГПУ, и добавить φ-функцию для каждого виртуального регистра ILOC в начале блока. При этом будет вставлено много лишнего; сложность полного алгоритма как раз и связана с устранением таких избыточных φ-функций.

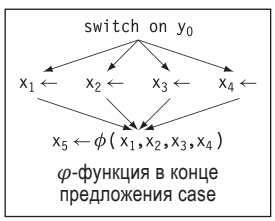
Чтобы переименовать виртуальные регистры ILOC, компилятор может обрабатывать блоки в порядке поиска в глубину. С каждым виртуальным регистром связывается счетчик. Когда компилятор встречает определение r_i, он увеличивает счетчик, связанный с r_i. Если ему присвоено новое значение `k`, то определение переписывается с использованием имени r_{ik}. При обходе блока алгоритм заменяет каждое вхождение r_i на r_{ik}, пока не встретит еще одно определение r_i. (Это определение приводит к увеличению значения счетчика до `k` + 1.) В конце блока компилятор про-

сматривает каждое ребро потока управления и переписывает соответствующий параметр φ-функции для r_i в каждом блоке, имеющем несколько предшественников.

После переименования код удовлетворяет обоим правилам SSA-формы. Каждое определение создает уникальное имя. При каждом использовании производится ссылка на единственное определение. Существует несколько улучшенных алгоритмов построения SSA-формы, которые вставляют меньше φ-функций, чем в этом простом подходе.

В этом примере присутствует несколько странностей SSA-формы, которые нуждаются в объяснении. Рассмотрим φ-функцию, определяющую x_1. Ее первый аргумент, x_0, определен в блоке, предшествующем циклу. Ее второй аргумент, x_2, определен позже в блоке с меткой `loop`. Таким образом, когда φ-функция исполняется в первый раз, один из ее аргументов не определен. Во многих контекстах языка программирования это могло бы привести к проблемам. Но поскольку φ-функция читает только один аргумент, а именно тот, что соответствует последнему выбранному ребру ГПУ, то читать неопределенное значение она не будет никогда.

φ-функция принимает любое число операндов. Чтобы привести к SSA-форме трехадресное ПП, автор компилятора должен включить механизмы, позволяющие обрабатывать более длинные списки операндов и связывать эти операнды с конкретными входящими ребрами. Рассмотрим блок в конце предложения выбора, показанный во врезке.



φ-функция для x_5 должна иметь аргумент для каждой ветви `case`. Количество нужных ей аргументов ограничено только числом путей, входящих в блок. Таким образом, операция для представления этой φ-функции в линейном ПП должна принимать произвольное число аргументов. Это не вписывается в трехадресную схему с фиксированной арностью.

При простом представлении трехадресного кода массивом автору компилятора потребуется побочная структура данных для хранения аргументов φ-функции. В двух других схемах реализации трехадресного кода, показанных на рис. 4.5, компилятор может вставлять кортежи переменного размера. Например, в кортежах для команд загрузки и непосредственной загрузки может быть место только для двух имен, тогда как кортеж для φ-операции мог бы быть достаточно большим, чтобы вместить все ее операнды плюс счетчик операндов.

ИТОГИ РАЗДЕЛА

Компилятор должен генерировать внутренние имена для всех значений, вычисляемых в программе. Эти имена могут быть явными или неявными. Правила генерирования имен самым непосредственным образом влияют на способности компилятора анализировать и оптимизировать ПП. Продуманная схема работы с именами позволяет кодировать и раскрывать факты для последующего использования в процессе оптимизации. Слишком большое число имен ведет к увеличению структур данных и замедлению компиляции.

Пространство имен SSA кодирует свойства, которые могут оказаться полезны для анализа и оптимизации; например, это позволяет оптимизациям не принимать в расчет переопределение имен (см. раздел 8.4.1). Такая дополнительная точность именования может упростить алгоритмы и улучшить результаты оптимизации.

ВОПРОСЫ

1. В ILOC-коде во врезке стр. 211 использует столько имен виртуальных регистров, сколько практически необходимо, – в предположении, что r_i и r_j не могут быть переименованы, потому что представляют переменные программы. Постройте эквивалентный код, в котором используется наименьшее возможное число виртуальных регистров.
2. Преобразуйте код, показанный во врезке, в SSA-форму, следуя пояснениям выше. Каждая ли вставленная вами φ-функция служит какой-то цели?

```
read x, y, z;
while (x >
0) {
    y = y + z;
    x = x - 1;
}
print y;
```

4.7. Размещение значений в памяти

Почти с каждым ПП ассоциирована карта памяти. Компилятор должен сопоставить место для хранения каждого значения, которое откомпилированный код вычисляет или использует. Это место может находиться в регистре или в памяти. Место может описываться конкретно: физический регистр или пара ⟨*базовый адрес*, *смещение*⟩ – либо символически: виртуальный регистр или символическая метка. Время жизни места размещения должно совпадать со временем жизни значения, т. е. оно должно быть доступно и зарезервировано для данного значения от момента его создания до момента его последнего использования.

Мы начнем этот раздел с обсуждения моделей памяти – неявных правил, применяемых для отнесения значений к областям данных. Далее будет более подробно описано назначение и размещение областей данных.

4.7.1. Модели памяти

Прежде чем транслировать исходную программу в промежуточное представление, компилятор должен для каждого вычисляемого в коде значения понять, где оно будет храниться. Компилятору необязательно перечислять все значения и их местоположения, но должен существовать механизм, позволяющий принимать такие решения согласованно и постепенно по ходу трансляции. Обычно авторы компиляторов формулируют набор решений, а затем применяют их в процессе трансляции. В совокупности эти правила образуют модель памяти откомпилированного кода.

Модели памяти помогают определить базовую модель вычислений: где операция находит свои аргументы? Они играют важнейшую роль в определении того, какие проблемы компилятор должен решить и сколько места и времени потребуется для их решения.

Активное значение
Значение *активно* в близкой окрестности того места, где оно определяется или используется.

Чаще всего применяются три модели: *память-память*, *регистр-регистр* и *стековая*. У них много общих характеристик, а различаются они тем, где хранятся значения, активные в текущем вычислении.

Модель память-память. Основное место расположения данных – память. Либо ПП поддерживает операции типа память-память, либо код перемещает активные значения в регистры, а неактивные – обратно в память.

Модель регистр-регистр. По возможности значения хранятся в *виртуальных регистрах*; некоторые локальные скалярные значения могут находиться только в виртуальном регистре. Глобальные значения хранятся в памяти (см. раздел 4.7.2).

Стековая модель. Значения хранятся на постоянной основе в памяти. Компилятор перемещает активные значения в стек и обратно с помощью явных операций (например, `push` и `pop`). Стековые ПП и ISA часто предоставляют операции для переупорядочения стека (например, `swap`).

Однозначное значение
Значение, доступ к которому возможен только по одному имени, называется однозначным.

Неоднозначное значение
Значение, доступ к которому возможен по нескольким именам, называется неоднозначным.

На рис. 4.8 показана одна и та же операция сложения в каждой из этих моделей. На рисунке (a) показана операция при двух разных предположениях. В левом столбце предполагается, что операнды `add` находятся в памяти и обозначены символическими метками. В правом столбце предполагается, что `add` – операция типа регистр-регистр, а значения находятся в памяти. Выбор одного из этих двух вариантов, вероятно, зависит от ISA целевой машины. На рисунке (b) показана та же операция `add` в модели регистр-регистр. Предполагается, что `a`, `b` и `c` – однозначные скалярные значения, находящиеся в виртуальных регистрах: vr_a, vr_b и vr_c соответственно. На рисунке (c) показана эта же операция в стековой модели; предполагается, что переменные находятся в памяти и имеют символические имена.

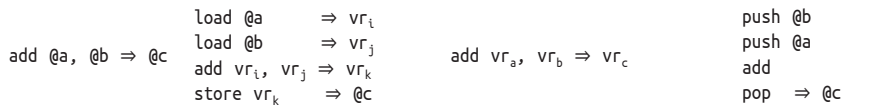


Рис. 4.8 Сложение с тремя операндами для разных моделей памяти

От модели памяти существенно зависит форма кода в ПП и приоритеты оптимизатора и фазы синтеза.

- В модели память-память неоптимизированная форма кода может использовать лишь небольшое число регистров. В этой ситуации приоритет отдается оптимизациям, которые помещают значения в неиспользуемые регистры на срок, составляющий нетривиальную часть их времени жизни. В фазе синтеза алгоритм распределения регистров уде-

ляет больше внимания отображению имен, нежели уменьшению спроса на физические регистры.

- В модели регистр-регистр неоптимизированный код может использовать гораздо больше виртуальных регистров, чем физически имеется в целевой машине. В этой ситуации поощряются оптимизации, которые не слишком сильно увеличивают спрос на регистры. В фазе синтеза распределение регистров необходимо для обеспечения правильности кода и является одним из основных факторов, влияющих на производительность.
- В стековой модели структура целевой машины становится критичной. Если в ISA имеются операции со стеком, как в виртуальной машине JAVA, то оптимизация отдает приоритет улучшению вычислений с использованием стека. Если ISA является CISC- или RISC-процессором, то компилятор, вероятно, будет транслировать код для стековой машины в какую-то другую форму перед генерированием кода.

Компилятор HotSpot Server, написанный на JAVA, транслировал байт-код JAVA в графовое ПП для оптимизации и генерирования кода.

В конечном итоге выбор модели памяти оказывает сильное влияние на проектирование оптимизатора и фазы синтеза компилятора.

Иерархия операций с памятью в ILOC 9X

Какой бы ни была модель памяти, автор компилятора обязан искать способы закодировать в ПП больше фактов о значениях. В 1990-х годах мы написали исследовательский компилятор, в котором использовалось ПП под названием ILOC 9x. В этом ПП была иерархия операций с памятью, позволявшая компилятору кодировать знания о значениях, хранящихся в памяти. В самом низу иерархии компилятор ничего или почти ничего не знал о значении, а на верхнем уровне он знал фактическое значение. Вот эти операции:

Операция	Семантика
Загрузка непосредственная	Загружает в регистр константу
Загрузка неизменяемая	Загружает значение, которое не изменяется во время выполнения. Компилятор не знает значение, но может доказать, что программа его не изменяет
Скалярная загрузка и сохранение	Операция со скалярным значением, а не элементом массива, структуры или значением указателя
Общая загрузка и сохранение	Операция со значением, которое может быть или не быть скалярным. Это самый общий случай

При такой иерархии фаза анализа кодировала знания о целевом значении прямо в коде ILOC 9x. Другие проходы могли переписывать более общие операции в более конкретные при установлении новых фактов. Например, если компилятор обнаруживал, что операция загрузки всегда порождает известное постоянное значение, то мог заменить общую или скалярную загрузку непосредственной.

Оптимизации были основаны на закодированных таким образом фактах. Например, сравнение между результатом неизменяемой загрузки с константой само должно быть инвариантным – факт, который трудно или невозможно доказать для общей операции загрузки.

4.7.2. Хранение значений в регистрах

Выгрузка
Распределитель регистров *выгружает* (spill) значение, сохраняя его в отведенном месте в памяти. Позже он может *восстановить* значение в регистр.

В модели регистр-регистр компилятор пытается назначить как можно большему числу значений виртуальные регистры. Этот подход полагается на то, что в окончательном коде распределитель регистров отобразит виртуальные регистры ПП на физические и будет *выгружать* в память виртуальные регистры, которым невозможно назначить физический.

Компилятор не может сохранять неоднозначное значение в регистре, когда происходит какое-то присваивание. В случае однозначного значения *x* компилятор точно знает, где значение *x* может измениться: там, где имеет место присваивание *x*. Таким образом, компилятор может безопасно генерировать код, который оставляет *x* в регистре.

В случае же неоднозначного значения *x* присваивание какому-то другому неоднозначному значению *y* потенциально может изменить значение *x*. Если компилятор попытается оставить *x* в регистре на время присваивания *y*, то в регистр не будет записано новое значение. Хуже того, в одной процедуре *x* и *y* могут указывать на одно и то же место хранения в одних вызовах, но на разные – в других. Это затрудняет компилятору задачу генерирования правильного кода, сохраняющего *x* в регистре. Перенос *x* в память оставляет решение задачи о том, какие присваивания должны изменять *x*, а какие не должны, на усмотрение аппаратуры формирования адресов.

Если вызывающая сторона помещает глобальное имя в параметр, передаваемый по ссылке, то вызываемая сторона может обратиться к значению как по глобальному имени, так и по имени формального параметра.

Тот же эффект имеет место, когда вызывающая сторона передает имя *x* в двух разных параметрах, передаваемых по ссылке.

На практике компиляторы решают, какие значения считать однозначными, а все неоднозначные значения хранят в памяти – в одной из областей данных или в куче, – а не в регистрах. Неоднозначность может возникать разными путями. Значения, хранящиеся в переменных, на которые ведет указатель, могут быть неоднозначными. Многие компиляторы считают элементы массива также неоднозначными, потому что компилятор не может сказать, относятся ли две ссылки вида `A[i,j]` и `A[m,n]` к одному и тому же элементу.

Как правило, компилятор сосредоточен на доказательстве того, что данное значение однозначно. Анализ может быть поверхностным и локальным. Например, в языке C всякая локальная переменная, к которой никогда не применяется операция взятия адреса, однозначна. При более сложных видах анализа для каждой указательной переменно строятся множества возможных имен; любая переменная, для которой это множество содержит ровно один элемент, однозначна. Анализ не способен разрешить все неоднозначности; недоказуемые случаи трактуются так, будто была доказана неоднозначность.

Особенности языка могут повлиять на способность компилятора к анализу неоднозначности. Например, в ANSI C имеются два ключевых слова, напрямую сообщающих информацию о неоднозначности. Ключевое слово `restrict` информирует компилятор о том, что указатель однозначен. Оно часто используется, когда процедура передает адрес непосредственно в месте вызова. Ключевое слово `volatile` позволяет программисту объявить о том, что содержимое переменной может измениться в любой момент без какого-либо уведомления. Оно используется для регистров аппаратных устройств и для переменных, которые могут быть изменены из функции обработки прерываний или из другого потока выполнения.

4.7.3. Отнесение значений к областям данных

Область данных
Область памяти, отведенная для хранения значений данных. С каждой областью памяти ассоциирована область видимости.

Примерами могут служить области локальных данных в процедурах и области глобальных данных.

Компилятор должен не только выбирать имя для каждого значения в программе, но и решать, где эти значения будут находиться во время выполнения. Модель памяти определяет, где должны находиться значения, когда они активны, и каждая модель из рассмотренных в разделе 4.7.1 вверяет некоторые значения памяти, когда они неактивны. Для каждого такого значения компилятор должен решить, где оно будет храниться на протяжении времени его жизни.

Большинство временных значений будут находиться в месте, зарезервированном для активных значений, – в регистрах или в области локальных данных в памяти, – как то диктует модель памяти с учетом доступности места. Каждой переменной, объявленной в исходной программе, компилятор назначает место постоянного хранения, исходя из ее свойств: времени жизни, видимости и области видимости объявления.

Время жизни. Под *временем жизни* значения понимается промежуток времени, в течение которого его можно определять или на него можно ссылаться. Вне своего времени жизни значение не определено.

Регион видимости (region of visibility)[1]**.** Значение *видимо*, если его можно поименовать, т. е. код может прочитать или записать значение. То есть регион видимости значения – это просто код, в котором оно видимо.

Область видимости объявления. Время жизни и видимость переменной зависят от области видимости (scope), в которой она объявлена. Например, время жизни статической относительно файла переменной в языке C совпадает со всем временем работы программы, но видима она только в том файле, в котором объявлена.

---

[1] К сожалению, авторы различают понятия *scope* и *region of visibility*, для которых на русском нет различных эквивалентов. Приходится называть *region* «регионом», чтобы хоть как-то их отличить. – *Прим. перев.*

В языках программирования имеются правила, определяющие время жизни и видимость каждого имени.

Чтобы упростить управление памятью, большинство компиляторов создают набор *областей данных*, ассоциированных с различными областями видимости. Для переменных, хранящихся в памяти, комбинация времени жизни и видимости определяет, в какую область данных следует поместить значение переменной.

С точки зрения места хранения, компилятор выделяет три категории времен жизни.

Автоматическая переменная
Имя, время жизни которого совпадает с одной активацией области видимости, в которой оно объявлено, называется *автоматической* переменной.

Автоматическая. Время жизни автоматической переменной совпадает с одной активацией его области видимости (процедурой или блоком). Значение определяется и используется в этой области видимости и перестает существовать по выходе из нее. Локальные переменные обычно являются автоматическими по умолчанию.

Мы называем такие переменные «автоматическими», потому что выделение и освобождение памяти для них может быть организовано как часть входа в соответствующую область видимости и выхода из нее. На этапе выполнения у каждого вызова процедуры имеется собственная область локальных данных, в которой могут быть размещены автоматические переменные.

Статическая переменная
Имя, которое сохраняет свое значение на протяжении нескольких активаций его области видимости, называется *статической* переменной.

Статическая. Время жизни статической переменной может охватывать несколько активаций области видимости, в которой она объявлена. Присвоенное ей значение будет сохраняться и после того, как она покинет область видимости, в которой имело место присваивание.

Компилятор может выделить память для таких переменных один раз, перед началом выполнения; по сути дела, они присутствуют всегда. Статические переменные хранятся в заранее выделенной области данных, ассоциированной с объявленной областью видимости. Компилятор может объединять статические области данных для нескольких областей видимости.

Константы – особый случай; это статические значения, которые можно инициализировать директивой на уровне языка ассемблера. Обычно компилятор создает для них отдельную область данных, называемую пулом констант.

Нерегулярная
Сущность, время жизни которой зависит от явного выделения памяти и явного или неявного ее освобождения, называется *нерегулярной*.

Нерегулярная. Время жизни нерегулярной переменной не связано с какой-либо одной областью видимости. Обычно память для нее выделяется явно, а освобождается явно или неявно. Примерами могут служить объекты в JAVA и строки, создаваемые с помощью `malloc`, в C.

Память для переменных с нерегулярной областью видимости обычно выделяется из динамической кучи (см. раздел 5.6.1).

Компилятор может классифицировать значения по времени жизни и области видимости. Эта классификация предполагает отведение конкретной области данных для хранения значений. На рис. 4.9 показана типичная схема, которой компилятор мог бы придерживаться при помещении переменных в регистры и области данных.

	Область видимости	Время жизни	Местоположение
ALGOL-подобные языки	Локальная	Автоматическое	Регистры или область локальных данных объявленной области видимости
	Локальная	Статическое	Процедура или область статических данных файла
	Файл	Статическое	Область статических данных файла
	Глобальная	Статическое	Область глобальных данных
Объектно-ориентированные языки	Метод	Автоматическое	Регистры или область локальных данных объявленной области видимости
	Метод	Статическое	Класс или область статических данных метода
	Класс	Статическое	Класс
	Глобальная	Статическое	Область глобальных данных
	Любая область видимости	Нерегулярное	Память явно выделяется из кучи

Рис. 4.9 Местоположение переменной в зависимости от области видимости и времени жизни

Имея отображение значений на области данных, компилятор должен назначить каждому хранящемуся в памяти значению адрес. Он обходит области данных, а внутри каждой области данных – значения, отнесенные к этой области. Каждому значению сопоставляется смещение от начала его области данных. Алгоритмы такого сопоставления обсуждаются в разделе 5.6.3.

Значениям, которые должны храниться в регистрах, компилятор назначает имя виртуального регистра. Фактическое отображение виртуальных регистров на аппаратные оставляется на усмотрение распределителя регистров.

ИТОГИ РАЗДЕЛА

Для каждого значения, вычисляемого программой, компилятор должен определить, где это значение будет находиться во время выполнения. Решение этой задачи зависит от языка программирования, принятой компилятором модели памяти, информации о времени жизни значений и от знаний о системной архитектуре целевой машины, которыми обладает автор компилятора. Компилятор систематически относит каждое значение к регистру или области данных и сопоставляет индивидуальным значениям смещения от начала области данных.

Решения о размещении значений могут оказывать влияние на производительность откомпилированного кода. Размещение в памяти может изменить поведение программы с точки зрения локальности. В решениях об отнесении к областям хранения могут быть закодированы тонкие знания о свойствах исходного кода, например о неоднозначности значений.

ВОПРОСЫ

1. Рассмотрим функцию `fib`, показанную во врезке. Напишите ILOC-код, который фаза анализа могла бы сгенерировать для этого кода, используя модель регистр-регистр и память-память. Сравните код для обеих моделей.
2. Напишите псевдокод алгоритма, который принимает список переменных, отнесенных к некоторой области данных, и назначает им смещения. Объясните, какая информация о каждой переменной нужна компилятору.

```
int fib(int n) {
  int x = 0;
  int y = 1;
  int z = 0;

  while (n > 0) {
     z = x + y;
     x = y;
     y = z;
     n = n - 1;
  }
  return z;
}
```

4.8. Итоги и перспективы

Выбор ПП оказывает большое влияние на дизайн, реализацию, скорость и эффективность компилятора. Ни одна из промежуточных форм, описанных в этой главе, не является идеальным решением для всех компиляторов или всех задач, решаемых данным компилятором. Автор компилятора должен принимать во внимание общие цели проекта при выборе ПП, проектировании ее реализации и добавлении вспомогательных структур, таких как таблицы символов и меток.

В современных системах компиляции используются всевозможные ПП, начиная от деревьев разбора и абстрактных синтаксических деревьев (которые часто используются в транспиляторах) и кончая линейными кодами на уровне более низком, чем машинный (например, в GCC). Во многих компиляторах используется несколько ПП – второе или третье строится для выполнения определенного анализа или преобразования и последующей модификации определяющего ПП с целью отразить результат.

Замечания к этой главе

Литература по ПП и опыту их использования скудна. Тем не менее ПП оказывают важное влияние на структуру и поведение компилятора. Классические формы ПП, такие как синтаксические деревья, АСД, ОАГ, четверки, тройки и одноадресный код, описывались в учебниках начиная с 1970-х годов [8, 36, 157, 181]. Более новые формы ПП типа SSA [56, 120, 279] описываются в литературе по анализу и оптимизации. В оригинальном компиляторе JAVA HotSpot Server в качестве определяющего ПП использовался вариант графа зависимостей в программе [92]. Мучник подробно обсуждает ПП и уделяет внимание использованию нескольких уровней ПП в одном компиляторе [279].

Идея о том, что несколько проходов по коду могут дать более эффективный код, восходит к работе Флойда [160]; это фундаментальное наблюдение обосновывает необходимость ПП и оправдывает дополнительные затраты на многопроходный компилятор. Она находит применение во многих контекстах внутри компилятора.

Идея использования хеш-функции для распознавания текстуально идентичных операций восходит к работе Ершова [150]. Ее конкретное воплощение в системах Lisp, похоже, относится к началу 1970-х годов [135, 174]; к 1980 году она уже была настолько расхожей, что Маккарти упоминает ее без цитирования [267].

Кэй и Пэйдж ввели разделение мультимножества как альтернативу хешированию [71]. Они хотели создать эффективный механизм поиска с гарантированно постоянным временем работы. Регулярные выражения, свободные от замыканий, описанные в разделе 2.6.2, могут дать аналогичный эффект. Работа по уменьшению размера АСД $\mathcal{R}^n$ была проделана Дэвидом Шварцем и Скоттом Уорреном.

Упражнения

К разделу 4.3

1. И дерево разбора, и абстрактное синтаксическое дерево сохраняют информацию о форме исходной программы.
   - **a.** Какая связь имеется между размером дерева разбора и размером входной программы?
   - **b.** Какая связь имеется между размером абстрактного синтаксического дерева и размером входной программы?
   - **c.** Какой связи между размером дерева разбора и размером абстрактного синтаксического дерева вы ожидаете? Иными словами, каким, на ваш взгляд, будет значение |дерево разбора| / |абстрактное синтаксическое дерево|?
2. Напишите алгоритм преобразования дерева выражения в ОАГ.

```
procedure main
    integer a, b, c;
    procedure f1(w, x)
        integer a, x, y;
        call f2(w, x);
        end;
    procedure f2(y, z)
        integer a, y, z;
        procedure f3(m, n)
            integer b, m, n;
Здесь ──►   c = a * b * m * n;
            end;
        call f3(c, z);
        end;
    ...
    call f1(a, b);
    end;
```

Рис. 4.10 Программа для упражнения 5

К разделу 4.4

3. Рассмотрите следующий фрагмент кода. Покажите, как его можно представить в виде абстрактного синтаксического дерева, в виде графа потока управления и в виде трехадресного кода.

```
if (c[i] = 0)
    then a[i] ← b[i] ÷ c[i];
    else a[i] ← b[i];
```

Обсудите преимущества каждого представления. Для каких приложений одно представление предпочтительнее других?

4. Алгоритм построения ГПУ, показанный на рис. 4.6, предполагает, что операция условного перехода, `cbr`, определяет метки как для выбранного, так и для неявного перехода.
Модифицируйте функции *FindLeaders* и *BuildGraph*, так чтобы они могли обрабатывать код, в котором операция `cbr` задает только выбранную ветвь.

К разделу 4.5

5. Вы пишете компилятор для простого языка с лексическими областями видимости. Рассмотрите программу на рис. 4.10.
   a. Постройте таблицу символов и ее содержимое непосредственно перед строкой кода, на которую указывает стрелка.
   b. Для каждого имени в предложении, на которое указывает стрелка, покажите, в каком объявлении оно определено.

6. Рассмотрите фрагмент кода на рис. 4.11. Нарисуйте для него ГПУ.

```
x ← ...
y ← ...
a ← y + 2
b ← 0
while (x < a)
    if (y < x) then
        x ← y + 1
        y ← b × 2
    else
        x ← y + 2
        y ← a ÷ 2;
w ← x + 2
z ← y × a
y ← y + 1
```

Рис. 4.11 Фрагмент кода для упражнения 6

К разделу 4.6

7. Напишите трехадресный код и код для стековой машины, который вычислял бы выражение `a × (b + c) ÷ d`. Предполагайте, что ПП может представить загрузку значения `a` с помощью загрузки из ячейки с меткой `@a`.

a. Сколько имен используется в трехадресном коде?

b. Сколько имен используется в коде для стековой машины?

8. Трехадресный и двухадресный коды различаются способами взаимодействия операций с пространством имен. В случае трехадресного кода есть возможность перезаписать значение с некоторым именем *n*. В случае двухадресного кода обыкновенные арифметические операции, такие как `add`, перезаписывают один из двух аргументов. Таким образом, работая с двухадресным кодом, компилятор должен выбрать, какой операнд сохранить, а какой перезаписать.

Выпишите три способа, которыми компилятор может транслировать выражение $a \leftarrow b \times c$ в низкоуровневый двухадресный код. Можете предполагать, что перед операцией умножения `b` и `c` находятся в регистрах r_b и r_c.

Как компилятор мог бы выбрать один из этих способов трансляции операции в двухадресный код?

К разделу 4.7

9. Рассмотрите три функции на языке C, показанные на рис. 4.12.

a. Если компилятор использует модель памяти регистр-регистр, то какие переменные в функциях `A`, `B` и `C` он вынужден был бы хранить в памяти? Обоснуйте свой ответ.

b. Предположим, что компилятор использует модель память-память. Рассмотрите выполнение двух предложений в ветви `if` конструкции `if-else`. Если в этот момент вычисления у компилятора имеется два свободных регистра, то сколько операций загрузки и сохранения ему потребуется, чтобы загрузить значения в регистры и затем снова сохранить их в памяти при выполнении этих двух предложений? А что, если у компилятора имеется три свободных регистра?

```
static int max = 0;

void A(int b, int e)
{
  int a, c, d, p;

  a = B(b);
  if (b > 100) {
    c = a + b;
    d = c * 5 + e;
  }
  else
    c = a * b;
  p = c;
  C(&p);
}
```

```
int B(int k)
{
  int x, y;

  x = pow(2, k);
  y = x * 5;
  return y;
}


void C(int *p)
{
  if (*p > max)
    max = *p;
}
```

Рис. 4.12 Код для упражнения 9

10. В языке FORTRAN имеется предложение `equivalence`, позволяющее разместить две переменные по одному адресу в памяти. Например, следующее предложение приводит к тому, что `a` и `b` занимают одну область памяти:

```
equivalence (a,b)
```

Может ли компилятор оставить локальную переменную в регистре на протяжении всей процедуры, если эта переменная встречается в предложении `equivalence`? Обоснуйте свой ответ.

Глава 5

Синтаксически управляемая трансляция

АННОТАЦИЯ

Задача компилятора – транслировать входную программу в форму, допускающую непосредственное выполнение на целевой машине. Сканер и парсер могут проанализировать код, предъявленный для компиляции, и определить, является ли он правильной программой на исходном языке. Но если компилятору предстоит выполнять трансляцию, оптимизацию и генерирование кода, то он должен построить промежуточное представление кода для последующего использования. Для этого требуется, чтобы компилятор мог рассуждать о коде на уровне более глубоком, чем контекстно-свободный синтаксис.

В этой главе будут рассмотрены некоторые проблемы, возникающие при выполнении компилятором первого этапа трансляции исходного кода в ПП в фазе анализа, а также механизмы, применяемые для решения этих проблем. В центре внимания будет стиль синтаксически управляемого вычисления, популяризированный генераторами парсеров и ставший общеупотребительной практикой.

КЛЮЧЕВЫЕ СЛОВА

Синтаксически управляемая трансляция, генерирование ПП, таблицы символов

5.1. Введение

По существу, компилятор является программой, которая читает другую программу, строит представление ее смысла, анализирует и улучшает код, представленный в такой форме, и транслирует этот код, так чтобы его можно было выполнить на целевой машине. Трансляция, анализ, оптимизация и генерирование кода требуют глубокого понимания входной программы. Цель

синтаксически управляемой трансляции – начать сбор знаний, которые понадобятся на более поздних этапах компиляции.

По мере разбора входной программы компилятор строит ПП-версию кода. Он аннотирует ПП обнаруженными фактами, такими как типы и размеры переменных, а также выведенными фактами, такими, например, как место хранения каждого значения. Компиляторы используют два механизма для построения ПП и вспомогательных структур данных: (1) синтаксически управляемая трансляция, т. е. форма вычислений, встроенная в парсер и выстраиваемая в последовательность действиями парсера, и (2) последующие обходы ПП для выполнения более сложных вычислений.

Концептуальная дорожная карта

Основное назначение компилятора – транслировать код с исходного языка на целевой. В этой главе рассматривается механизм, которым авторы компиляторов пользуются для трансляции исходного кода программы в ПП. Автор компилятора планирует трансляцию на уровне продукций грамматики исходного языка, а инструменты выполняют шаги этого плана по мере того, как парсер распознает отдельные продукции грамматики. Конкретная последовательность действий, выполняемых на этапе компиляции, зависит как от плана, так и от разбора.

В процессе трансляции компилятор нарабатывает понимание смысла исходной программы на операционном уровне. Компилятор строит модель пространства имен входной программы. Он использует эту модель для вывода информации о типе каждой именованной сущности. Модель используется также для того, чтобы решить, где будет размещаться каждое вычисленное в коде значение на этапе выполнения. В совокупности эти факты позволяют компилятору сгенерировать начальную ПП-программу, улавливающую смысл оригинальной программы на исходном языке.

Несколько слов о времени

Трансляция выводит на свет божий все временные проблемы, возникающие при конструировании компилятора. Во время разработки автор компилятора планирует как поведение во время выполнения, так и механизмы времени компиляции, чтобы создать код, который будет проявлять это поведение. Он кодирует эти планы в синтаксически управляемые правила, ассоциированные с продукциями грамматики. Еще не выходя за рамки этапа проектирования, он должен рассуждать как о поддержке трансляции во время компиляции в форме таких структур, как таблицы символов, и таких процессов, как проверка типов, так и о поддержке во время выполнения, чтобы у кода была возможность находить значения и обращаться к ним. (Мы воочию увидим эту поддержку в главах 6 и 7, но автор компилятора должен думать о том, как все это создать, использовать и поддерживать еще на этапе проектирования и реализации начальных стадий трансляции.)

Система времени выполнения совокупность функций, реализующих такие абстракции, как куча и ввод-вывод.

Во время сборки компилятора генератор парсеров преобразует грамматики и правила синтаксически управляемой трансляции в исполняемый парсер. Во время компиляции парсер выявляет поведения и привязки, которые вступят в силу во время выполнения, и кодирует их в оттранслированной программе. Во время выполнения скомпилированный код во взаимодействии с системой времени выполнения создает поведения, которые автор компилятора планировал еще на этапе проектирования.

Краткий обзор

Автор компилятора создает инструмент – компилятор, – который транслирует входную программу в форму, допускающую непосредственное выполнение на целевой машине. Стало быть, компилятору нужен план реализации, модель пространства имен и механизм для связывания операций с моделью и генерирования ПП со структурой и синтаксисом входной программы. Чтобы решить эти задачи:

- компилятор нуждается в механизме, который связывает процессы сбора информации и построения ПП с синтаксической структурой и семантическими деталями входной программы;
- компилятор должен понимать видимость каждого имени в коде, т. е. для заданного имени *x* должен знать, к какой сущности *x* привязано. Имея эту привязку, он нуждается еще в полной информации о типе *x* и в методе доступа к *x*;
- компилятор нуждается в схеме реализации каждой конструкции языка программирования – от ссылки на переменную до предложения выбора и от вызова процедуры до выделения памяти из кучи.

В этой главе нас будет интересовать механизм, который часто используется для описания синтаксически управляемого вычисления. Автор компилятора описывает действия, которые должны быть выполнены, когда парсер сворачивает заданную продукцию. Генератор парсеров организует выполнение этих действий в нужных точках разбора. Авторы компиляторов используют эти механизмы для управления сбором базовой информации, генерирования ПП и проверки ошибок на уровнях более глубоких, чем синтаксис (например, ссылается ли предложение на необъявленный идентификатор?).

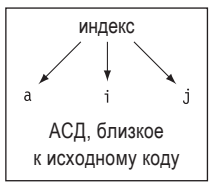
В главах 6 и 7 обсуждается реализация других типичных конструкций языков программирования.

В разделе 5.3 мы познакомимся с механизмом, который часто используется для трансляции исходного кода в ПП. В качестве примеров мы рассмотрим схемы реализации вычисления выражений и некоторых простых управляющих конструкций. В разделе 5.4 объясняется, как компиляторы управляют таблицами символов и используют их для моделирования окружения именования и хранения атрибутов имен. В разделе 5.5 приводится введение в анализ типов; полное изложение системы типов выходит

за рамки этой книги. Наконец, в разделе 5.6 мы увидим, как компилятор назначает значениям место хранения.

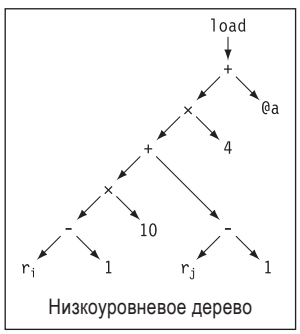
5.2. Предварительные сведения

Компилятор принимает многочисленные решения о деталях реализации программы. Поскольку одно решение тянет за собой другое, авторы компиляторов часто применяют стратегию прогрессивной трансляции. В фазе анализа с использованием информации, доступной парсеру, строится начальное ПП программы и набор аннотаций. Затем компилятор анализирует ПП для вывода дополнительной информации и по мере ее получения уточняет детали ПП и аннотаций.



АСД, близкое к исходному коду

Чтобы понять, почему прогрессивная трансляция необходима, рассмотрим древовидное представление ссылки на элемент массива `a[i,j]`. Парсер легко может построить сравнительно абстрактное ПП, например АСД, близкое к исходному коду, которое показано во врезке. В АСД закодированы только те факты, которые неявно присутствуют в коде.



Низкоуровневое дерево

Чтобы сгенерировать ассемблерный код ссылки, компилятору нужно знать гораздо больше деталей, чем дает АСД, близкое к исходному коду. Низкоуровневое дерево, показанное во врезке, содержит эти детали и раскрывает ряд фактов, которые нельзя увидеть в АСД, приближенном к исходному коду. Все эти факты находят отражение в окончательном коде.

- Компилятор должен знать, что `a` является массивом четырехбайтовых целых чисел размера 10×10 с нижней границей 1 по каждому измерению. Эти факты выведены из предложений, где объявляется или создается `a`.
- Компилятор должен знать, что массив `a` хранится по строкам (см. рис. 5.16). Это решение было принято проектировщиком языка или автором компилятора еще до начала работы над компилятором.
- Компилятор должен знать, что `@a` – ассемблерная метка, результатом вычисления которой является адрес первого элемента `a` на этапе выполнения (см. раздел 7.3). Этот факт вытекает из стратегии именования, принятой автором компилятора на этапе проектирования.

Чтобы сгенерировать исполняемый код для `a[i,j]`, компилятор должен вывести эти факты в процессе трансляции.

В этой главе рассматривается механизм синтаксически управляемой трансляции и его использование для первого этапа трансляции – исходного кода в ПП. Механизм, который мы опишем, был впервые применен в раннем гене-

раторе LR(1)-парсеров, `yacc`. Нотация позволяет автору компилятора задавать небольшой фрагмент кода, называемый *действием*, который будет выполняться в момент свертки соответствующей продукции грамматики парсером.

Синтаксически управляемая трансляция позволяет автору компилятора задавать действия и оставляет парсеру решение о том, когда это действие следует применить. Синтаксис входной программы определяет последовательность применения действий. Действия могут содержать произвольный код, так что автор компилятора имеет возможность создавать и постепенно заполнять сложные структуры данных. При должной предусмотрительности и расчетливом планировании автор компилятора может использовать механизм синтаксически управляемой трансляции для реализации нетривиального поведения.

Применяя синтаксически управляемую трансляцию, компилятор приобретает о программе знания, выходящие за пределы контекстно-свободного синтаксиса входного кода. Синтаксически ссылка на переменную *x* – это просто имя. Но в процессе трансляции компилятор обнаруживает и выводит из контекстов, где имя встречается, гораздо больше информации об *x*.

- Исходный код может определять и манипулировать несколькими разными сущностями с именем *x*. Компилятор должен отобразить каждую ссылку на *x* в правильный экземпляр *x* времени выполнения; он должен привязать *x* к конкретной сущности, исходя из окружения именования, в котором встретилась ссылка. Для этого он строит и использует подробную модель пространства имен входной программы.
- Разобравшись с привязкой *x* в текущей области видимости, компилятор должен понять, какие значения может принимать *x*, их размер, структуру и время жизни. Эта информация позволяет компилятору определить, какие операции применимы к *x*, и предотвратить недопустимые манипуляции, а также определить тип *x* и способ взаимодействия этого типа с контекстами, в которых *x* встречается.
- Чтобы сгенерировать код, манипулирующий значением *x*, компилятор должен знать, где это значение будет находиться во время выполнения. Если *x* имеет внутреннее строение, как обстоит дело в случае массива, структуры, строки или записи, то компилятору нужна формула для доступа к отдельным элементам *x*. Для каждого значения, которое будет вычисляться в программе, компилятор должен определить, где оно будет находиться.

Но и это еще не все трудности. Исполняемые программы обычно включают код, откомпилированный в разное время. Автор компилятора должен придумать механизмы, благодаря которым результаты раздельных компиляций смогут правильно и органично взаимодействовать друг с другом. Процесс начинается с синтаксически управляемой трансляции, которая строит промежуточное представление кода. Далее следуют дополнительный анализ и уточнение. Все это опирается на тщательно спланированные взаимодействия между процедурами и пространствами имен (см. главу 6).

5.3. Синтаксически управляемая трансляция

Синтаксически управляемая трансляция – это совокупность приемов, которыми авторы компиляторов пользуются для привязки действий этапа компиляции к грамматической структуре входной программы. Эта структура проявляется в процессе разбора кода в фазе анализа. Автор компилятора описывает вычисления, которые парсер инициирует в определенных точках разбора. В случае LR(1)-парсера действия инициируются, когда парсер выполняет свертку.

5.3.1. Первый пример

На рис. 5.1(a) показана простая грамматика, описывающая множество положительных целых чисел. Мы можем использовать синтаксически управляемые действия, привязанные к этой грамматике, для вычисления значения любого допустимого положительного целого числа.

```
1  Number  → DList
2  DList   → DList digit
3          | digit
```

(a) Грамматика положительных целых чисел

	Action		*Goto*
Состояние	**eof**	**digit**	**DList**
0		s 2	1
1	acc	s 3	
2	r 3	r 3	
3	r 2	r 2	

(b) Ее таблицы *Action* и *Goto*

Рис. 5.1 Грамматика положительных целых чисел

На рисунке (b) показаны таблицы *Action* и *Goto* для этой грамматики. В парсере имеется три возможные свертки, по одной для каждой продукции.

- Парсер сворачивает по правилу 3, *DList* → `digit`, для самой левой цифры числа.
- Парсер сворачивает по правилу 2, *DList* → *DList* `digit`, для каждой цифры, кроме первой.
- Парсер сворачивает по правилу 1, *Number* → *DList*, после того как свернул последнюю цифру.

Эту стратегию программисты на ассемблере использовали десятилетиями.

Парсер может вычислить значение целого числа с помощью последовательности задач, определенных для продукций. Он может аккумулировать значение слева направо – после каждой цифры умножать частично вычисленное значение на десять и прибавлять только что прочитанную цифру. С каждым символом, встретившимся при разборе, связано значение. Мы можем закодировать эту стратегию в правилах для продукций, которые применяются во время свертки парсером.

В нотации, получившей распространение благодаря генераторам парсеров `yacc` и `bison`, правила могут выглядеть следующим образом:

```
Number : DList         { return $1; } ;
DList  : DList digit   { $$ = $1 * 10 + CToI($2); } ;
       | digit         { $$ = CToI($1); } ;
```

Символы `$$`, `$1` и `$2` ссылаются на значения, связанные с символами грамматики в продукции. `$$` ссылается на нетерминальный символ в левой части (LHS) правила. Символ `$`*i* ссылается на значение *i*-го символа в правой части правила (RHS).

В примере предполагается, что функция `CToI()` преобразует литеру лексемы в целое число. Автор компилятора должен обращать внимание на типы значений в стеке, представленных символами `$$`, `$1` и т. д.

Используя таблицы *Action* и *Goto* на рис. 5.1(b) для разбора строки «`175`», LR(1)-парсер получил бы последовательность действий, показанную на рис. 5.2. Свертки следуют в таком порядке: *reduce 3*, *reduce 2*, *reduce 2*, *accept*.

Напомним, что начальный `$` в стеке представляет пару ⟨*INVALID, INVALID*⟩.

- *Reduce 3* применяет действие правила 3 с целым числом 1 в качестве значения `digit`. Правило присваивает единицу левой части *DList*.
- *Reduce 2* применяет действие правила 2 с числом 1 в качестве значения *DList* в правой части и целым 7 в качестве `digit`. Оно присваивает 1×10 + 7 = 17 символу *DList* в левой части.
- *Reduce 2* применяет действие правила 2 с числом 17 в качестве значения *DList* в правой части и целым 5 в качестве `digit`. Оно присваивает 17×10 + 5 = 175 символу *DList* в левой части.
- Действие приемки, которое тоже является сверткой по правилу 1, возвращает значение *DList* в левой части, равное 175.

Итерация	Состояние	Слово	Стек	Действие
0	–	1	$ ⟨*Number* 0⟩	*shift 2*
1	2	7	$ ⟨*Number* 0⟩ ⟨`digit` 2⟩	*reduce 3*
2	1	7	$ ⟨*Number* 0⟩ ⟨*DList* 1⟩	*shift 3*
3	3	5	$ ⟨*Number* 0⟩ ⟨*DList* 1⟩ ⟨`digit` 3⟩	*reduce 2*
4	1	5	$ ⟨*Number* 0⟩ ⟨*DList* 1⟩	*shift 3*
5	3	`eof`	$ ⟨*Number* 0⟩ ⟨*DList* 1⟩ ⟨`digit` 3⟩	*reduce 2*
6	1	`eof`	$ ⟨*Number* 0⟩ ⟨*DList* 1⟩	*accept*

Рис. 5.2 Действия парсера для числа «175»

Правила свертывания, применяемые в порядке действий, выполняемых парсером, создают простой каркас для вычисления значения целого числа.

Важнейшее наблюдение состоит в том, что парсер применяет эти правила в предсказуемом порядке, управляемом структурой грамматики и разбором входной строки. Автор компилятора задает действие для каждой

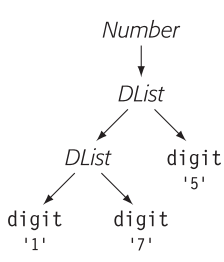
свертки; последовательность и результаты применения этих действий зависят исключительно от грамматики и входной строки. Такое синтаксически управляемое вычисление лежит в основе парадигмы программирования, которая играет центральную роль в трансляции и находит много других применений.

```
int stoi( char *s ) {
  int i = 0;
  while('0'≤*s≤'9') {
    i = i * 10
      + ((int)*s - '0');
    s++;
  }
  return i;
}
```

Конечно, этот пример искусственный. В реальной системе то же самое вычисление почти наверняка выполнялось бы небольшим специализированным фрагментом кода, похожим на тот, что приведен во врезке. Результат тот же самый, но без накладных расходов, связанных с алгоритмами сканирования и разбора. На практике этот код с большой вероятностью был бы встраиваемым, и никакая функция не вызывалась бы. (Накладные расходы на вызов функции, скорее всего, превосходят стоимость цикла.) Тем не менее пример прекрасно подходит для объяснения принципов синтаксически управляемых вычислений.

Эквивалентная формулировка с применением обхода дерева

Вычисление значения в соответствии с грамматикой целого числа можно также записать в виде рекурсивного обхода синтаксического дерева. На рис. 5.3(a) показано синтаксическое дерево для строки «175» и леворекурсивной грамматики. На рисунке (b) показан простой обход дерева для вычисления ее значения. В нем используется функция *integer(c)* для преобразования одной литеры в целое значение.



(a) Синтаксическое дерево для «175» и леворекурсивной грамматики

```
Value( root )
    if (root is type Number) then
        return Value( son( root ))
    else if (root is type DList) then
        return 10 × Value( left( root ))
                + Value( right( root ))
    else if (root is type digit) then
        return integer( root's lexeme )
```

(b) Обход синтаксического дерева для вычисления значения

Рис. 5.3 Вычисления для грамматики целого положительного числа в процессе обхода дерева

Формулировка с обходом дерева раскрывает несколько важных аспектов синтаксически управляемого вычисления в стиле yacc. Поток информации в синтаксическом дереве направлен от листьев к корню. В действии, ассоциированном с продукцией, имена имеются только для значений, связанных с символами грамматики, поименованными в продукции. Восходящий поток информации в такой парадигме работает хорошо, нисходящий – нет.

Ограничение на направление потока информации – снизу вверх – может показаться проблематичным. Но на самом деле автор компилятора может обойти эту парадигму и ограничения, воспользовавшись нелокальными переменными и структурами данных в «действиях». Действительно, одно из применений таблицы символов компилятора как раз в том и состоит, чтобы предоставить нелокальный доступ к данным, выведенным в процессе синтаксически управляемых вычислений.

В принципе, любую проблему, нуждающуюся в нисходящем потоке информации, можно решить, оставаясь в рамках нисходящего каркаса, если передавать всю информацию вверх по дереву до общего предка и решать задачу в этой точке. На практике эта идея работает плохо, потому что (1) разработчик должен спланировать весь поток информации; (2) он должен написать код, реализующий этот план, и (3) вычисленный результат появляется в точке дерева, далеко отстоящей от места, где он нужен. Поэтому лучше переосмыслить вычисление, чем передавать всю информацию по дереву.

Форма грамматики

Поскольку грамматика диктует последовательность действий, ее форма влияет на стратегию вычислений. Рассмотрим праворекурсивную версию грамматики целых положительных чисел. В ней сначала сворачивается самая правая цифра, что наводит на мысль о следующем подходе:

```
Number : DList        { return second($1); } ;
DList  : digit DList  { $$ = pair(10 * first($2),
                           first($2) * CToI($1) + second($2)); }
       | digit        { $$ = pair(10,CToI($1)); } ;
```

В этой схеме множитель и значение аккумулируются справа налево. Чтобы сохранить оба значения в *DList*, используется конструктор `pair` и функции `first` и `second` для доступа к компонентам пары. Хотя такая парадигма работает, понять ее значительно труднее, чем механизм для леворекурсивной грамматики.

При проектировании грамматики автор компилятора должен принимать во внимание виды вычислений, которые должен выполнять парсер. Иногда изменение грамматики может сделать вычисления более простыми и быстрыми.

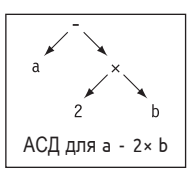
5.3.2. Трансляция выражений

Выражения образуют значительную часть большинства программ. Если рассматривать их как деревья – а не ориентированные ациклические графы, – то они дают естественный пример синтаксически управляемой трансляции. На рис. 5.4 приведен простой пример синтаксически управляемого каркаса построения абстрактного синтаксического дерева для выражений. Правила просты:

- если продукция содержит оператор, то строится внутренний узел для представления этого оператора;
- если продукция выводит имя или число, то строится листовый узел, в котором запоминается лексема;
- если существует продукция, обеспечивающая предшествование, то она передает АСД подвыражения наверх.

Продукция			Семантически управляемые действия
Expr	→	*Expr* + *Term*	{ $$ ← *MakeNode2*(plus, $1, $3) ; };
	|	*Expr* – *Term*	{ $$ ← *MakeNode2*(minus, $1, $3) ; };
	|	*Term*	{ $$ ← $1 ; };
Term	→	*Term* × *Factor*	{ $$ ← *MakeNode2*(times, $1, $3) ; };
	|	*Term* ÷ *Factor*	{ $$ ← *MakeNode2*(divide, $1, $3) ; };
	|	*Factor*	{ $$ ← $1 ; };
Factor	→	(*Expr*)	{ $$ ← $2 ; };
	|	number	{ $$ ← *MakeLeaf*(number, *lexeme*) ; };
	|	name	{ $$ ← *MakeLeaf*(name, *lexeme*); };

Рис. 5.4 Построение абстрактного синтаксического дерева



АСД для a - 2× b

В коде используется два конструктора для построения узлов. MakeNode2*(a, b, c)* строит бинарный узел типа *a* с сыновними узлами *b* и *c*. MakeLeaf(name*, a)* строит листовый узел и ассоциирует с ним лексему *a*. Для выражения a - 2× b такая схема трансляции построила бы АСД, показанное во врезке.

АСД имеют прямую и очевидную связь с грамматической структурой входной программы. Трехадресный режим лишен такого прямого отображения. Тем не менее синтаксически управляемый каркас легко может сгенерировать трехадресный код для выражений и присваиваний. На рис. 5.5 показан синтаксически управляемый каркас для генерирования ILOC-подобного кода по классической грамматике выражений. Каркас предполагает, что в начале обработки выражения значения находятся в памяти.

```
Expr   → Expr + Term     { $$ ← NextRegister() ;
                           Emit( add, $1, $3, $$ ) ; };
       | Expr – Term      { $$ ← NextRegister() ;
                           Emit( sub, $1, $3, $$ ) ; };
       | Term             { $$ ← $1 ; };
Term   → Term × Factor    { $$ ← NextRegister() ;
                           Emit( mult, $1, $3, $$ ) ; };
       | Term ÷ Factor    { $$ ← NextRegister() ;
                           Emit( div, $1, $3, $$ ) ; };
       | Factor           { $$ ← $1 ; };
Factor → ( Expr )         { $$ ← $2 ; };
       | number           { $$ ← NumberIntoReg( lexeme) ; };
       | name             { entry ← STLookup( lexeme );
                           $$ ← ValueIntoReg( entry ) ; };
```

Рис. 5.5 Генерирование трехадресного кода для выражений

Чтобы упростить каркас, автор компилятора предоставил высокоуровневые функции, которые абстрагируют детали хранения значений:

- *NextRegister* возвращает номер нового регистра;
- *NumberIntoReg* возвращает номер регистра, в котором находится константное значение из лексемы;
- *STLookup* принимает на входе имя и возвращает запись таблицы символов для сущности, к которой в данный момент это имя привязано;
- *ValueIntoReg* возвращает номер регистра, в котором находится текущее значение имени из лексемы.

Если бы грамматика включала присваивание, то понадобилась бы вспомогательная функция *RegIntoMemory*, которая перемещает значение из регистра в память.

Вспомогательные функции, такие как *NumberIntoReg* и *ValueIntoReg*, должны генерировать трехадресный код, представляющий методы доступа к именованным сущностям. Если в ПП имеются только низкоуровневые операции, как в ILOC, то эти функции могут оказаться сложными. Альтернативный подход – ввести в трехадресный код высокоуровневые операции, которые сохраняют существенную информацию, и отложить реализацию этих операций до того момента, когда станут окончательно понятны карта памяти и методы доступа.

Реализация большинства функций очевидна. Например, *NumberIntoReg*('2') может генерировать операцию непосредственной загрузки.

Применение этой синтаксически управляемой схемы трансляции к выражению a - 2 × b порождает ILOC-код, показанный во врезке. В нем предполагается, что r_{arp} содержит указатель на область локальных данных процедуры, а @a и @b – смещения от r_{arp} адресов, по которым программа хранит значения a и b. Результат остается в r_5.

```
loadAI  rarp, @a  ⇒ r1
loadI   2         ⇒ r2
loadAI  rarp, @b  ⇒ r3
mult    r2, r3    ⇒ r4
sub     r1, r4    ⇒ r5
```

ILOC-код для a - 2 × b

Реализация в LR(1)-парсере

Этот стиль синтаксически управляемых вычислений впервые появился в yacc, раннем генераторе LALR(1)-парсеров. Реализация требует внести два изменения в заготовку LR(1)-парсера. Понимание этих изменений проливает свет как на нотацию yacc, так и на способы ее эффективного использования. На рис. 5.6 показана модифицированная заготовка LR(1)-парсера. Изменения набраны ***полужирным шрифтом***.

Первое изменение заключается в создании места хранения значения, ассоциированного с символом грамматики во время вывода. В оригинальной заготовке состояние хранилось в стеке в парах ⟨*symbol*, *state*⟩, где *symbol* – символ грамматики, а *state* – состояние парсера. В модифицированном парсере эти пары заменены тройками ⟨*symbol*, *state*, *value*⟩, где *value* содержит сущность, сопо-

Генераторы парсеров различаются способом присваивания значения терминальному символу.

ставленную $$ в свертке, которая перенесла тройку в стек. В действиях сдвига используется значение лексемы.

```
поместить ⟨INVALID, INVALID, INVALID⟩ в стек
поместить ⟨start symbol, s0, INVALID⟩ в стек
word ← NextWord( )

while (true) do
  state ← state из тройки на вершине стека
  if Action[state,word] = "reduce A → β" then
    value ← PerformActions(A → β)
    вытолкнуть |β| пар из стека
    state ← state из тройки на вершине стека
    поместить ⟨A, Goto[state, A], value⟩ в стек
  else if Action[state,word] = "shift si" then
    поместить ⟨word, si, lexeme⟩ в стек
    word ← NextWord( )
  else if Action[state,word] = "accept" and word = eof
    then break
  else сообщить о синтаксической ошибке

сообщить об успехе /* выполнен случай "accept" */
```

Рис. 5.6 Заготовка LR(1)-парсера с поддержкой трансляции

Второе изменение заставляет парсер вызвать функцию *PerformActions* перед сверткой. Результат этого вызова парсер записывает в поле *value*, когда помещает новую тройку в стек.

Генератор парсеров конструирует *PerformActions* на основе действий трансляции, заданных для каждой продукции в грамматике. Заготовка парсера передает этой функции номер продукции, а сама функция состоит из предложения ветвления по номеру продукции с вызовом фрагмента кода, соответствующего редукции.

Осталось только транслировать символы $$, $1, $2 и т. д. из нотации `yacc` в конкретные ссылки в стеке. $$ представляет значение, возвращенное функцией *PerformActions*. Все остальные символы $*i* являются ссылками на поле *value* в тройке, соответствующей символу *i* в правой части продукции. Поскольку эти тройки помещаются в стек в порядке справа налево, то $*i* транслируется в поле *value* тройки, отстоящей на $|\beta| - i$ позиций от вершины стека.

Обработка нелокальных вычислений

До сих пор мы видели только примеры локальных вычислений в грамматике. Каждое правило могло ссылаться лишь на имена символов в той же самой продукции. Но многие задачи, решаемые компилятором, нуждаются в информации из других частей вычисления; если мы говорим об обходе дерева, то речь идет о данных из отдаленных частей синтаксического дерева.

Примером нелокального вычисления в компиляторе может служить ассоциирование типа, времени жизни и информации о видимости с имено-

ванной сущностью, например переменной, процедурой, объектом или структурой. Компилятор узнает о сущности, когда впервые встречает ее в области видимости – это вхождение называется *определяющим*. Встретив определяющее вхождение имени x, компилятор должен определить свойства x. А для обработки последующих ссылочных вхождений компилятору нужен доступ к ранее установленным свойствам.

Определяющее вхождение
Первое вхождение имени в данной области видимости называется его определяющим вхождением.

Любое последующее использование называется *ссылочным вхождением*.

Те правила, с которыми мы познакомились в предыдущем примере, дают естественный механизм передачи информации вверх по дереву разбора и выполнения локальных вычислений – включающих значения, ассоциированные с узлом и его потомками. Но чтобы транслировать выражение вида x + y в низкоуровневое трехадресное ПП, компилятор должен обратиться к информации, которая находится далеко в дереве разбора – в объявлениях x и y. Если компилятор попытается сгенерировать низкоуровневый трехадресный код этого выражения, то ему также понадобится доступ к информации, выведенной из синтаксиса, например сведения о том, может ли код хранить x в регистре, т. е. является x однозначной или нет. Стандартный способ решить эту проблему – хранить информацию, необходимую для нелокальных вычислений, в глобально доступной структуре данных. В большинстве компиляторов для этой цели используется таблица символов (см. разделе 4.5).

Использование глобальной таблицы символов для предоставления нелокального доступа аналогично применению глобальных переменных в императивной программе.

На самом деле «таблица символов» – это сложный набор таблиц и путей поиска. Концептуально читатель может представлять ее как хеш-таблицу, адаптированную к каждой области видимости. В конкретной области видимости путь поиска состоит из упорядоченного списка таблиц, которые компилятор просматривает в поисках имени.

Представлением таблицы символов манипулируют различные части грамматики. Определяющее вхождение создает запись в таблице символов. Его объявления, если таковые имеются, устанавливают различные атрибуты и привязки. При каждом ссылочном вхождении производится поиск в таблице для определения атрибутов имени. Предложения, открывающие новую область видимости, такие как процедура, блок или объявление структуры, создают новые области видимости и включают их в путь поиска. Могут возникать и более тонкие проблемы; если в программе на C берется адрес переменной a, а именно &a, то компилятор должен пометить a как потенциально неоднозначную.

В динамически типизированном языке, например в PYTHON, предложения, определяющие x, могут изменять атрибуты x.

Тот же прием с использованием глобальной переменной для передачи информации между правилами трансляции, применяется и в других контекстах. Рассмотрим исходный язык с простым синтаксисом объявлений. Парсер

может создавать записи в таблице символов для каждого имени и заносить их атрибуты по мере обработки объявлений. Например, синтаксис исходного языка мог бы включать такой набор правил:

```
Declaration → TypeSpec NameList { CurType ← invalid; };
TypeSpec    → int               { CurType ← int; };
            | float             { CurType ← float; };
NameList    → NameList , name   { err ← SetType($2,CurType); };
            | name              { err ← SetType($1,CurType); };
```

где SetType создает новую запись для имени name, если таковой еще не существует, и сообщает об ошибке, если name существует и его объявленный тип отличается от CurType.

Тип объявленной переменной определяется в продукциях с *TypeSpec* в левой части. Действие для *TypeSpec* записывает тип в глобальную переменную CurType. Когда имя встречается в продукции *NameList*, действие заносит в тип name значение, хранящееся в CurType. Тем самым автор компилятора смог передать информацию из правой части одной продукции в правую часть другой.

Однопроходные компиляторы

На заре компиляции разработчики старались строить однопроходные компиляторы – трансляторы, которые генерировали ассемблерный или машинный код за один проход по исходной программе. Во времена, когда быстрыми считались компьютеры с производительностью порядка килофлопов, эффективность трансляции была важной проблемой.

Чтобы упростить однопроходную трансляцию, проектировщики языков изобретали правила, призванные избежать необходимости нескольких проходов. Например, в PASCAL требуется, чтобы все объявления располагались раньше первого исполняемого предложения; это ограничение позволяло компилятору разрешить имена и определиться с размещением в памяти еще до генерирования какого-либо кода. Оглядываясь назад, не ясно, были ли такие ограничения действительно необходимыми или желательными.

Выполнение нескольких проходов по коду позволяет компилятору собрать больше информации и во многих случаях сгенерировать более эффективный код, как отметил Флойд в 1961 году [160]. При наличии современных, более сложных процессоров почти все компиляторы выполняют несколько проходов по промежуточному представлению кода.

Форма грамматики

Форма грамматики может играть важную роль в определении способа вычислений. Чтобы избежать глобальной переменной CurType в предыдущем примере, автор компилятора мог бы следующим образом переписать грамматику синтаксиса объявления:

```
Declaration → int INameList
            | float FNameList
INameList   → NameList name   { err ← SetType($2, int); };
            | name            { err ← SetType($1, int); };
```

```
FNameList  → NameList name   { err ← SetType($2, float); };
           | name            { err ← SetType($1, float); };
```

Эта грамматика допускает тот же самый язык. Однако в ней создаются различные списки для имен типа `int` и `float`. Как видно, автор компилятора может воспользоваться этими двумя продукциями для кодирования типа прямо в синтаксически управляемом действии. Эта стратегия упрощает механизм трансляции и устраняет необходимость использования глобальной переменной для передачи информации между продукциями. Транслятор становится проще для написания, понимания и, вероятно, для сопровождения. Иногда приспособление формы грамматики к вычислениям может упростить синтаксически управляемые действия.

Адаптация выражений к контексту

Более тонкая форма нелокальных вычислений может возникать, когда автору компилятора необходимо принять решение на основе информации из нескольких продукций. Например, рассмотрим проблему расширения каркаса на рис. 5.5, так чтобы можно было сгенерировать операцию непосредственного умножения (`multI` в ILOC) при трансляции выражения. Скажем, в однопроходном компиляторе могла бы возникнуть необходимость сгенерировать `multI` в начальном ПП.

Для выражения `a × 2` каркас на рис. 5.5 породил бы код типа показанного во врезке. (В коде предполагается, что `a` находится в регистре r_a.) Свертка по правилу *Factor* → `number` генерирует `loadI`; она выполняется перед сверткой по правилу *Term* → *Term* × *Factor*.

```
loadI 2       ⇒ r_i
mult  r_a,r_i ⇒ r_j

ILOC -код для r_a × 2
```

Чтобы можно было распознать возможность генерирования `multI`, автору компилятора пришлось бы добавить в действие для продукции *Term* → *Term* × *Factor* код, который распознает, что `$3` содержит небольшую целую константу, и в таком случае генерирует `multI`. Симметричный случай требует аналогичной проверки для `$1`. Но даже при всех этих усилиях операция `loadI` останется. При последующей оптимизации ее можно было бы убрать (см. раздел 10.2).

Фундаментальная проблема заключается в том, что действия в нашей синтаксически управляемой трансляции способны получить доступ только к локальной информации, потому что могут именовать лишь символы в текущей продукции. Эта структура вынуждает транслятор генерировать операцию `loadI`, прежде чем он может узнать, что встречающееся в этой операции значение допускает «непосредственный» вариант.

Напрашивается идея переработать грамматику, отразив в ней случай `multI`. Если автор компилятора заменит *Term* → *Term* × *Factor* тремя продукциями, показанными во врезке, то сможет сгенерировать `multI` в действии для продукции *Term* → *Term* × `number`, которая включает случай `a × 2`. Но вот случай `2 × a` останется неохваченным. Прямая подстановка в левый операнд не работает, потому что грамматика леворекурсивная. В лучшем случае прямая подстановка может выявить либо непосредственный левый

```
Term → Term × ( Expr )
     | Term × number
     | Term × name
```

операнд, либо непосредственный правый операнд.

Самое полное решение этой проблемы – создать общую операцию умножения и оставить возможность переписать код на этапе последующей оптимизации или выбора команд. Обе техники выбора команд, описанные в главе 11, способны обнаружить возможность для использования `multI` и соответственно переписать код.

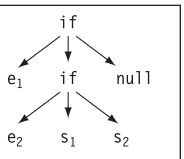
Оконная оптимизация
оптимизация, в которой сопоставление с образцом применяется для упрощения кода, находящегося в небольшом буфере.

Если компилятор все-таки должен генерировать `multI` на ранней стадии, то самый рациональный подход – завести небольшой буфер на три или четыре операции и выполнить *оконную оптимизацию* (peephole optimization) на этапе генерирования начального ПП (см. раздел 11.3.1). Так можно легко обнаружить и переписать неэффективные места, подобные только что описанному.

5.3.3. Трансляция управляющих конструкций

Как мы видели, ПП выражений близко следует синтаксису выражений, что ведет к прямолинейным схемам трансляции. Управляющие конструкции, например `if-then-else` или циклы, могут потребовать более сложных представлений.

Построение АСД



АСД для вложенных предложений if

Парсер может построить АСД для представления управляющих конструкций естественным образом. Рассмотрим вложение конструкций `if-then-else` с использованием грамматики на рис. 5.7. В АСД можно завести для представления `if` узел с тремя сыновними узлами. Один сыновний узел содержит управляющее выражение, другой – предложения в ветви `then`, третий – предложения в ветви `else`. Во врезке показано АСД для входной строки

`if` e_1 `then if` e_2 `then` s_1 `else` s_2

Действия для построения этого АСД очевидны.

```
1  Stmt     → if Expr then Stmt
2           | if Expr then WithElse else Stmt
3           | Other
4  WithElse → if Expr then WithElse else WithElse
5           | Other
```

Рис. 5.7 Однозначная грамматика If-Then-Else

Построение трехадресного кода

Чтобы транслировать конструкцию `if-then-else` в трехадресный код, компилятор должен закодировать передачи управления с помощью набора меток, условных и безусловных переходов. Трехадресное ПП напоминает очевидный ассемблерный код конструкции:

1) вычислить управляющее выражение;
2) перейти на часть `then` (s_1) или на часть `else` (s_2) в зависимости от результата;
3) в конце выбранной части выполнить безусловный переход на начало предложения, следующего за конструкцией `if-then-else` – «выход».

В этой схеме трансляции требуются метки для части `then`, части `else` и выхода, а вместе с ними условный переход и два безусловных.

Продукция 4 в грамматике на рис. 5.7 демонстрирует проблемы, возникающие в схеме трансляции для генерирования ILOC-подобного кода `if-then-else`. Другие продукции генерируют ПП для вычисления *Expr* и реализации частей `then` и `else`. Схема для правила 4 должна объединить эти разрозненные части в код для полной конструкции `if-then-else`.

Трудности с правилом 4 вызваны тем, что парсер должен генерировать ПП в нескольких разных точках: после распознавания *Expr*, после распознавания *WithElse* в части `then` и после распознавания *WithElse* в части `else`. В прямолинейном наборе правил действие для правила 4 выполнялось бы после разбора всех трех частей, и именно тогда было бы создано ПП для их вычисления.

В схеме для правила 4 должно быть несколько разных действий, активируемых в разных местах правила. Для достижения этой цели автор компилятора может модифицировать грамматику, создав свертки в тех точках разбора, где схема трансляции должна выполнять некоторое действие.

На рис. 5.8 показана переписанная версия продукции 4, где свертки создаются в критических точках разбора вложенной конструкции `if-then-else`. Введено три новых нетерминальных символа, каждый из которых определен ϵ-продукцией.

WithElse	→	`if` *Expr* *CreateBranch*
		`then` *WithElse* *ToExit1*
		`else` *WithElse* *ToExit2*
CreateBranch	→	ϵ
ToExit1	→	ϵ
ToExit2	→	ϵ

Рис. 5.8 Создание действий в середине продукции

Свертка для *CreateBranch* может создать все три метки, вставить условный переход и команду `nop` с меткой для части `then`. Свертка для *ToExit1* вставляет безусловный переход на метку

> Компилятор мог бы опустить код для *ToExit2*, положившись на неявную ветвь условного перехода.
>
> Но, сделав условный переход явным, мы оставляем больше свободы последующим проходам для переупорядочения кода (см. раздел 8.6.2).

выхода, за которым следует команда `nop` с меткой для части `else`. Наконец, *ToExit2* вставляет безусловный переход на метку выхода, за которым следует команда `nop` с меткой выхода.

Осталась еще одна сложность. Автор компилятора должен учитывать вложенные конструкции. Все три метки должны храниться способом, который связывает их с этим конкретным экземпляром *WithElse* и делает доступными из других действий, ассоциированных с правилом 4. Пока что наша нотация не предлагает решения этой проблемы. Генератор парсеров `bison` расширил нотацию `yacc` для ее решения, так чтобы автору компилятора не нужно было вводить явный стек троек с меточными значениями.

Решение `bison` заключается в том, чтобы позволить вставлять действие между любыми двумя символами в правой части продукции. Выглядит это так, как будто `bison` вставил новый нетерминальный символ в точке действия и ϵ-продукцию для этого символа. Затем он ассоциирует действие с этой новой продукцией. Автор компилятора должен вести аккуратный подсчет; присутствие действия в середине продукции создает дополнительное имя и увеличивает на единицу номера в именах символах справа от него.

При использовании этой схемы действия в середине продукции могут обращаться к позиции стека, ассоциированной с любым символом в расширенной продукции, включая символ в левой части правила 4. В случае `if-then-else` действие между *Expr* и `then` может временно сохранить тройку меток в позиции стека для левой части, `$$`. Позже действия, следующие за обоими нетерминальными символами *WithElse*, могут найти нужные им метки в `$$`. Решение не слишком элегантное, но все же позволяет организовать не совсем локальный доступ.

Предложения выбора и цикла создают аналогичные проблемы. Компилятору нужно закодировать поток управления оригинальной конструкции в наборе меток, условных и безусловных переходов. Стек разбора предлагает естественный способ отслеживать информацию для вложенных управляющих конструкций.

ИТОГИ РАЗДЕЛА

В процессе трансляции компилятор порождает промежуточное представление кода. Чтобы поддержать этот начальный этап трансляции, генераторы парсеров предлагают средство для задания синтаксически управляемых вычислений, которое связывает вычисление с грамматикой. Затем парсер организует последовательность выполнения этих действий, сообразуясь с фактическим синтаксисом входной программы.

Синтаксически управляемая трансляция дает эффективный механизм генерирования ПП. Она легко подстраивается под решения, требующие локальных знаний или знаний, собранных на более ранних стадиях разбора. Но она не позволяет принимать решения, основанные на фактах, которые устанавливаются позже в процессе разбора. Для этого необходимо несколько проходов по ПП, предназначенных для уточнения и улучшения кода.

ВОПРОСЫ

1. Грамматика во врезке определяет синтаксис простого калькулятора с четырьмя операциями. Калькулятор отображает текущий результат при каждой свертке *Expr* или *Term*. Напишите действия для синтаксически управляемой схемы, которые будут вычислять выражения в этой грамматике.
2. Рассмотрите грамматику на рис. 5.7. Напишите набор правил трансляции для построения АСД конструкции `if-then-else`.

```
Expr  → Expr + Term
      | Expr - Term
      | Term
Term  → Term × number
      | Term ÷ number
      | number
```

Грамматика простого калькулятора

5.4. Моделирование окружения именования

Современные языки программирования позволяют программисту создавать сложные пространства имен. Большинство языков поддерживают тот или иной вариант иерархии лексического именования, в которой видимость, тип и время жизни выражаются в привязке к структуре программы. Многие языки также поддерживают объектно-ориентированную иерархию именования, где видимость и тип связаны с наследованием, а время жизни определяется явным или неявным выделением и освобождением памяти. В процессе трансляции, оптимизации и генерирования кода компилятору нужны механизмы для моделирования и понимания этих иерархий.

Когда компилятор встречает имя, его синтаксически управляемые правила трансляции должны отобразить это имя на конкретную сущность, например переменную, объект или процедуру. Эта привязка имени к сущности играет важную роль в трансляции, поскольку определяет тип имени и метод доступа к нему, что, в свою очередь, влияет на код, который может генерировать компилятор. Компилятор использует свою модель пространства имен для определения такой привязки – этот процесс называется *разрешением имени*.

Статическая привязка
Когда компилятор может определить привязку имени к сущности, мы говорим, что эта привязка статическая, т. е. не будет изменяться во время выполнения.

Пространство имен программы может содержать несколько подпространств, или областей видимости. В главе 4 мы сказали, что область видимости – это участок программы, определяющий границы пространства имен. Внутри области видимости программист может определять новые имена. Имена видны в своей области видимости и, вообще говоря, не видны вне ее.

Основной механизм, применяемый для моделирования окружения именования, – это набор таблиц, которые сообща называются таблицей символов. Компилятор строит эта таблицы на начальном этапе трансляции. Для статически привязанных имен он аннотирует ссылки на имя ссылкой на конкретную таблицу имен. Для имен с динамической привязкой, например

Динамическая привязка
Когда компилятор не может определить привязку имени к сущности и должен отложить разрешение имени до этапа выполнения, мы говорим, что эта привязка динамическая.

виртуальных функций в C++, компилятор должен предусмотреть возможность разрешения привязки во время выполнения. В процессе разбора компилятор создает, модифицирует и отбрасывает части этой модели.

Прежде чем переходить к обсуждению построения и обслуживания модели видимости, уместно будет дать краткий обзор правил областей видимости.

5.4.1. Лексические иерархии

Лексическая область видимости
Области видимости, вложенные друг в друга в том порядке, в каком встречаются в программе, часто называются *лексическими*.

Большинство языков программирования предлагают вложенные лексические области видимости в той или иной форме. Общий принцип, лежащий в основе правил лексических областей видимости, прост:

> *В точке программы* p *вхождение имени* n *относится к сущности с именем* n*, которая была создана явно или неявно в области видимости, лексически ближайшей к* p.

Таким образом, если *n* используется в текущей области видимости, то оно ссылается на *n*, объявленное в текущей области видимости, если такое существует. Если же нет, то оно относится к объявлению *n* в ближайшей объемлющей области видимости. Самая внешняя область видимости, как правило, содержит имена, видимые во всей программе, которые обычно называются *глобальными*.

Создание нового имени

Языки программирования различаются способами объявления имен. В одних языках требуется объявлять каждую именованную переменную и процедуру. В других атрибуты имени определяются путем применения правил в том месте, где встретилось определяющее вхождение имени. В третьих вывод атрибутов имени зависит от контекста.

Трактовка определяющего вхождения имени *x* в область видимости *S* зависит от правил видимости в исходном языке программирования и от окружающего контекста:

- если *x* встречается в предложении объявления, то атрибуты *x* в *S* очевидны и корректно определены;
- если *x* встречается в виде ссылки и экземпляр *x* виден в области видимости, объемлющей *S*, то в большинстве языков *x* привязывается к этой сущности;
- если *x* встречается в виде ссылки и никакой экземпляр *x* не виден, то порядок обработки зависит от языка. В APL, PYTHON и даже в FORTRAN создается новая сущность. В C такая ссылка считается ошибкой.

Встретив определяющее вхождение, компилятор должен создать подходящие структуры, которые позволят сохранить имя и его атрибуты и сделать имя видимым при поиске.

Языки программирования различаются также по способу обозначения границ областей видимости. В PASCAL область видимости ограничена парой операторных скобок `begin-end`. В C область видимости находится между фигурными скобками { и }. Определения структур и записей создают область видимости, содержащую имена элементов. Определение класса в объектно-ориентированных языках создает новую область видимости для имен, видимых в этом классе.

В PASCAL фигурные скобки играют роль ограничителей комментария.

Чтобы сделать обсуждение более конкретным, рассмотрим программу на PASCAL, показанную на рис. 5.9. Она содержит пять областей видимости, по одной для каждой процедуры: `Main`, `Fee`, `Fie`, `Foe` и `Fum`. В каждой процедуре объявлены какие-то переменные с именами из набора `x`, `y`, `z`. В коде каждое имя снабжено индексом, обозначающим номер его уровня. Имена, объявленные в процедуре, всегда имеют уровень, на единицу больший, чем уровень имени процедуры. Таким образом, `Main` имеет уровень 0, а имена `x`, `y`, `z`, `Fee` и `Fie`, объявленные непосредственно в `Main`, имеют уровень 1.

Статическая координата
Для имени *x*, объявленного в области видимости *s*, его статической координатой является пара ⟨*l*, *o*⟩, где *l* – лексический уровень вложенности *s*, а *o* – смещение, по которому *x* хранится в области данных этой области видимости.

Для представления имен в языке с лексическими областями видимости компилятор может использовать *статическую координату* каждого имени. Статическая координата – это пара ⟨*l*, *o*⟩, где *l* – лексический уровень вложенности имени, а *o* – его смещение в области данных для уровня *l*. Компилятор получает статическую координату в процессе разрешения имен – отображения имени на конкретную сущность.

Моделирование лексических областей видимости

В процессе обработки входного кода парсер должен строить и обслуживать модель окружения именования. Модель изменяется по мере того, как парсер входит в различные области видимости и покидает их. В основе модели лежит таблица символов компилятора.

Компилятор может строить отдельную модель для каждой области видимости, как показано на рис. 5.10. На рисунке (a) представлена внешняя область видимости `J`, содержащая две внутренние области видимости. В области видимости `K` переменные `a` и `b` имеют тип `int`, а переменные `c` и `d` – тип `char`. В области видимости `L` переменные `a` и `c` имеют тип `int`, а переменные `b` и `d` – тип `float`.

Области видимости `K` и `L` вложены в область видимости `J`, и никакой другой связи между ними нет.

На рисунке (b) показаны соответствующие таблицы символов. Таблица для области видимости состоит из хеш-таблицы и ссылки на объемлющую область видимости. Стрелки обозначают путь поиска, отражающий вложенность кода. Таким образом,

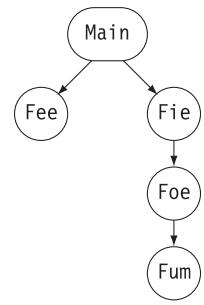
поиск a в области видимости K не приведет к успеху в таблице для K, затем перейдет по ссылке в область видимости J и там найдет определение a как переменной типа int.

```
program Main0(input0, output0);
  var x1, y1, z1: integer;
  procedure Fee1;
     var x2: integer;
     begin { Fee1 }
        x2 := 1;
        y1 := x2 * 2 + 1
     end;
  procedure Fie1;
     var y2: real;
     procedure Foe2;
        var z3: real;
           procedure Fum3;
              var y4: real;
              begin { Fum3 }
                 x1 := 1.25 * z3;
                 Fee1;
                 writeln('x = ',x1)
              end;
        begin { Foe2 }
           z3 := 1;
           Fee1;
           Fum3
        end;
     begin { Fie1 }
        Foe2;
        writeln('x = ',x1)
     end;
  begin { Main0 }
     x1 := 0;
     Fie1
  end.
```

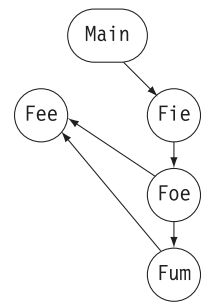
(a) Программа на PASCAL

Область видимости	x	y	z
Main	$\langle 1,0 \rangle$	$\langle 1,4 \rangle$	$\langle 1,8 \rangle$
Fee	$\langle 2,0 \rangle$	$\langle 1,4 \rangle$	$\langle 1,8 \rangle$
Fie	$\langle 1,0 \rangle$	$\langle 2,0 \rangle$	$\langle 1,8 \rangle$
Foe	$\langle 1,0 \rangle$	$\langle 2,0 \rangle$	$\langle 3,0 \rangle$
Fum	$\langle 1,0 \rangle$	$\langle 4,0 \rangle$	$\langle 3,0 \rangle$

(b) Статические координаты



(c) Отношения вложенности



(d) Отношения вызова

Рис. 5.9 Вложенные лексические области видимости в PASCAL

Этот подход позволяет компилятору создавать гибкие, допускающие поиск модели окружения именования в каждой области видимости. *Пусть поиска* – это просто список, или цепочка таблиц, которая определяет порядок просмотра таблиц. На этапе компиляции путь поиска имени начинается в текущей области видимости и продолжается вверх по цепочке объемлющих областей. Поскольку связь между областями видимости статическая (не изменяется), компилятор может строить пути поиска для конкретной области видимости

в процессе синтаксически управляемой трансляции и сохранять эти таблицы и пути для использования на более поздних этапах, а при необходимости и в других инструментах.

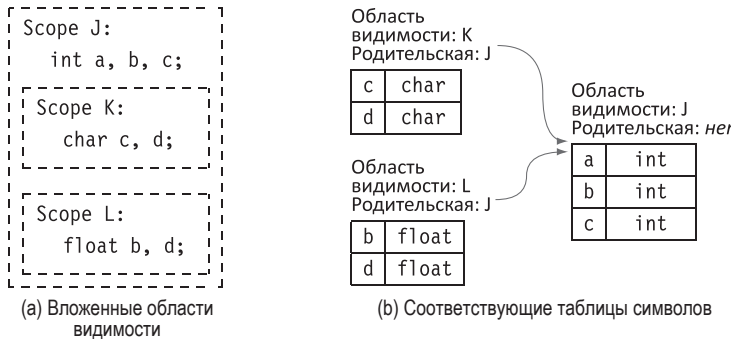


Рис. 5.10 Таблицы для лексической иерархии

Построение модели

Автор компилятора может организовать построение модели пространств имен в процессе синтаксически управляемой трансляции. Конструкции исходного языка, знаменующие вход и выход из областей видимости, могут инициировать действия для создания таблиц и путей поиска. Продукции для объявлений и ссылок могут создавать и уточнять записи таблицы с описанием имен.

- *Ограничители блоков*, такие как `begin` и `end`, { и }, а также конструкции входа в процедуры и выхода из них могут создавать новую таблицу при входе в область видимости и помещать ссылку на нее в начало пути поиска для блоков, ассоциированных с текущей областью видимости. При выходе действие должно пометить таблицу как завершенную.

Динамические области видимости

Альтернативой лексическим являются динамические области видимости. Различие между ними проявляется, только когда процедура ссылается на переменную, объявленную вне ее собственной области видимости; иногда такие переменные называются *свободными*.

В случае лексических областей видимости правило простое и единообразное: свободная переменная привязывается к объявлению своего имени, лексически ближайшему к месту использования. Если компилятор начинает поиск в области видимости, содержащей точку использования переменной, и последовательно проверяет объемлющие области видимости, то переменная привязывается к первому найденному объявлению. Объявление всегда находится в области видимости, охватывающей ссылку.

В случае динамических областей видимости правило столь же простое: свободная переменная привязывается к имени, которое было создано последним во время

выполнения. То есть когда в процессе выполнения встречается свободная переменная с некоторым именем, она привязывается к самому последнему экземпляру этого имени. В ранних реализациях создавался стек, в который имена помещались в момент обнаружения определяющего вхождения. Чтобы привязать свободную переменную, исполняемый код просматривал стек имен сверху донизу в поисках подходящего имени. Более поздние реализации стали эффективнее.

Хотя во многих ранних Lisp-системах использовались динамические области видимости, выбор все же был сделан в пользу лексической видимости. Динамическую видимость легко реализовать в интерпретаторе, но эффективная реализация в компиляторе несколько труднее. Возникающие при этом ошибки бывает трудно найти и понять. Динамическая видимость все еще встречается в некоторых языках, например в Common Lisp.

- Объявления переменных, если они существуют, создают записи для объявленных имен в локальной таблице и заполняют их атрибутами, указанными в объявлении. Если объявления не существует, то атрибуты, например тип, должны быть выведены из ссылок. Кое-какая информация о размере может быть выведена в точках, где выделяется память для агрегатов.
- *Ссылки* инициируют поиск на пути для текущей области видимости. В языке с объявлениями отсутствие имени в локальной таблице заставляет продолжить поиск вдоль всего пути. В языке без объявлений ссылка может послужить причиной для создания локальной сущности с таким именем, а может ссылаться на имя в объемлющей области видимости. Правила неявного объявления зависят от языка.

В FORTRAN создается имя с атрибутами по умолчанию, зависящими от первой буквы имени. В C производится поиск в объемлющих областях видимости, и ошибка диагностируется, только если ни в одной из них имя не найдено. В PYTHON действия зависят от того, чем является первое вхождение имени в области видимости: определением или использованием.

Примеры

Правила лексических областей видимости во всех языках программирования похожи. Однако проектировщикам языков удается включать неожиданные и неординарные штрихи. Автор компилятора должен адаптировать описанные выше общие схемы трансляции к конкретным правилам исходного языка.

В языке C используется простое пространство имен с лексической видимостью. Имена процедур и глобальных переменных находятся в глобальной области видимости. Каждая процедура создает собственную локальную область видимости для переменных, параметров и меток. В C не поддерживаются вложенные функции, хотя некоторые компиляторы, в т. ч. GCC, реализуют эту возможность в качестве расширения. Блоки, обозначаемые фигурными скобками { и }, создают собственные локальные области видимости; блоки могут быть вложенными.

Ключевое слово C `static` одновременно ограничивает видимость имени и определяет его время жизни. Статическое глобальное имя видимо только внутри файла, в котором объявлено. Статическое локальное имя имеет статическую видимость. Время жизни любого статического имени глобально, т. е. имя сохраняет свое значение между вызовами процедуры, в которой объявлено.

В языке SCHEME правила видимости похожи на действующие в языке C. Почти все сущности в SCHEME размещаются в одной глобальной области видимости. Сущности могут быть данными или исполняемыми выражениями. Системные функции, такие как `cons`, сосуществуют с написанным пользователем кодом и данными. Код, который состоит из исполняемого выражения, может создавать закрытые объекты с помощью выражения `let`. Путем вкладывания выражений `let` друг в друга можно создавать вложенные лексические области видимости произвольной глубины.

PYTHON – Algol-подобный язык, обходящийся без объявлений. Он поддерживает три вида областей видимости: локальную область для имен, определенных внутри функции; глобальную область для имен, определенных вне всех написанных программистом функций; встроенную область для имен, предоставленных реализацией, например `print`. Эти области видимости вкладываются в следующем порядке: локальная, глобальная, встроенная. Сами функции могут быть вложенными, что создает иерархию локальных областей видимости.

PYTHON не предлагает объявлений типов. Первое использование имени *x* в некоторой области видимости является его определяющим вхождением. Если при первом использовании *x* присваивается значение, то производится привязка *x* к новой локальной сущности, тип которой определен типом присвоенного значения. Если первое использование является ссылкой на значение *x*, то производится привязка *x* к глобальной сущности; если такой сущности еще не существует, то определяющее вхождение создает сущность. Если программист хочет, чтобы имя *x* было глобальным, но должен определить его перед использованием, то может дополнить имя объявлением `nonlocal`, означающим, что *x* должно быть помещено в глобальную область видимости.

Терминология объектно-ориентированных языков

Разнообразие объектно-ориентированных языков породило некоторую размытость терминологии, встречающейся в их обсуждениях. Чтобы устранить неоднозначность, в этой главе мы будем использовать следующие термины.

Объект. Объект – это абстракция с одним или несколькими членами. Это могут быть члены данных, код, манипулирующий членами данных, или другие объекты. Объект, который включает члены, содержащие код, называется *классом*. У каждого объекта имеется внутреннее состояние – данные, время жизни которых совпадает со временем жизни самого объекта.

Класс. Класс – это совокупность объектов, имеющих одинаковую абстрактную структуру и характеристики. Класс определяет множество членов данных в каждом *экземпляре* класса, а также множество членов с кодом, или *методов*, локальных

в данном классе. Некоторые методы являются *открытыми*, т. е. видимыми извне, другие – *закрытыми*, т. е. видимыми только внутри класса.

Наследование. Наследование – это отношение между классами, определяющее частичный порядок на множестве областей видимости классов. Класс *a* может наследовать члены от своего *суперкласса*. Если *a* является суперклассом *b*, то *b* называется *подклассом a*. Имя *x*, определенное в подклассе, маскирует все определения *x* в суперклассе. Некоторые языки допускают наследование нескольких суперклассов одним классом.

Получатель. Методы вызываются от имени некоторого объекта, называемого получателем метода. Внутри метода получатель обозначается зарезервированным словом, например `this` или `self`.

Мощь ООЯ в значительной степени обусловлена организационными возможностями, предоставляемыми его пространствами имен.

5.4.2. Иерархии наследования

Объектно-ориентированные языки (ООЯ) привносят вторую форму вложенного пространства имен благодаря наследованию. В ООЯ имеются *классы*. Класс состоит из совокупности (возможно, пустой) объектов, имеющих одинаковую структуру и поведение. Определение класса задает члены данных и кода, имеющиеся у любого объекта класса.

Полиморфизм
Способность сущности принимать различные типы часто называют полиморфизмом.

Мощь ООЯ в значительной степени проистекает из возможности создавать новые классы на основе определений уже существующих классов. В терминологии JAVA новый класс β может расширять существующий класс α; тогда объекты класса β наследуют определения членов данных и кода от определения α. β может переопределять имена из α, наделяя их новой семантикой и типами; новые определения маскируют предыдущие определения в α и его суперклассах. В других языках схожие возможности предлагаются под другими названиями.

Терминология, используемая при описании наследования, в разных языках разная. В JAVA подкласс *расширяет* свой суперкласс. В C++ подкласс является *производным* от суперкласса.

Полиморфизм подтипа
способность объекта подкласса ссылаться на члены суперкласса.

Расширение класса создает иерархию наследования: если α является *суперклассом* β, то любой метод, определенный в α, должен правильно работать с объектом класса β, при условии что этот метод виден в β. Обратное неверно. В методе подкласса могут использоваться члены, которые не определены в экземплярах суперкласса; такой метод не может правильно работать с объектом суперкласса.

В языке с одиночным наследованием, например JAVA, наследование индуцирует древовидную иерархию на множестве классов. В других языках классу β разрешено иметь несколько непосредственных суперклассов. Эта

концепция «множественного наследования» дает программисту возможность повторно использовать больше кода, но ценой создания более сложного окружения для разрешения имен.

Каждое определение класса создает новую область видимости. Имена членов данных и кода принадлежат определению класса. Многие языки предоставляют явный механизм управления видимостью имен членов. В некоторых языках определение класса может содержать другие классы для создания внутренней лексической иерархии. Наследование определяет второй путь поиска, основанный на отношении между подклассом и суперклассом.

В процессе трансляции компилятор должен отобразить пару ⟨*объект*, *член*⟩ в объявление конкретного члена в определении конкретного класса. Такая привязка дает компилятору информацию о типе и методе доступа, необходимую ему для трансляции ссылки. Компилятор ищет имя объекта в лексической иерархии; эта запись дает класс, который служит отправной точкой для поиска имени члена в иерархии наследования.

Моделирование иерархий наследований

Лексическая иерархия отражает синтаксическую вложенность. Иерархия наследования создается определениями, а не местом в синтаксической структуре.

Для разрешения имен членов компилятору нужна модель иерархии наследования, определенная множеством объявлений класса. Компилятор может построить отдельную таблицу для области видимости, ассоциированной с классом, в процессе разбора объявления класса. Фразы исходного языка, описывающие наследование, заставляют компилятор связывать области видимости классов, в результате чего образуется иерархия. В языке с одиночным наследованием иерархия имеет древовидную структуру – классы являются потомками своих суперклассов. В языке с множественным наследованием иерархия образует ациклический граф.

Компилятор использует для моделирования иерархии наследования те же инструменты, что и для моделирования лексической иерархии. Он создает таблицы, моделирующие каждую область видимости. Эти таблицы связываются для формирования путей поиска. Порядок поиска зависит от правил областей видимости и наследования, действующих в конкретном языке. Но технология создания и обслуживания модели от языка не зависит.

Разрешение имен во время компиляции и во время выполнения

Замкнутая структура классов
Если структура классов приложения фиксируется на этапе компиляции, то говорят, что ООЯ имеет замкнутую иерархию.

Главная трудность, присущая некоторым ООЯ, возникает не из-за присутствия иерархии наследования, а из-за того, когда эта иерархия определяется. Если ООЯ требует, чтобы определения классов присутствовали

на этапе компиляции и чтобы эти определения не могли изменяться, то компилятор может разрешить имена членов, выполнить необходимую проверку типов, определить подходящие методы доступа и сгенерировать код для ссылок на имена членов. Говорят, что такой язык имеет *замкнутую структуру классов*.

Открытая структура классов
Если приложение может изменять структуру классов во время выполнения, то говорят, что в нем *открытая иерархия*.

С другой стороны, если язык разрешает работающей программе изменять структуру классов – либо импортируя определения классов во время выполнения, как в JAVA, либо путем редактирования определений классов, как в SMALLTALK, то часть механизма разрешения имя и привязки может быть отложена до этапа выполнения. Говорят, что такой язык имеет *открытую структуру классов*.

Поиск имен при наличии наследования

Предположим на время, что структура классов замкнутая. Рассмотрим два случая.

Квалифицированное имя
составное имя, например `x.part`, где `part` – элемент агрегатной сущности с именем `x`

1. Встретив ссылку на неквалифицированное имя *n* в некоторой процедуре *p*, компилятор ищет *n* в лексической иерархии. Если *p* – метод, определенный в классе *c*, то *n* может быть также членом данных *c* или какого-то суперкласса *c*; таким образом, компилятор должен вставить часть иерархии наследования в подходящую точку пути поиска.
2. Встретив ссылку на член *m* объекта *o*, компилятор сначала разрешает *o* в лексической иерархии и получает в результате экземпляр некоторого класса *c*. Далее он ищет *m* в таблице для класса *c*, а если не находит, то продолжает поиск в каждой таблице вдоль цепочки суперклассов *c* (в порядке их следования). В итоге он либо находит *m*, либо иерархия заканчивается.

Одна из основных возможностей JIT-компиляторов – снижение накладных расходов, связанных с разрешением имен во время выполнения.

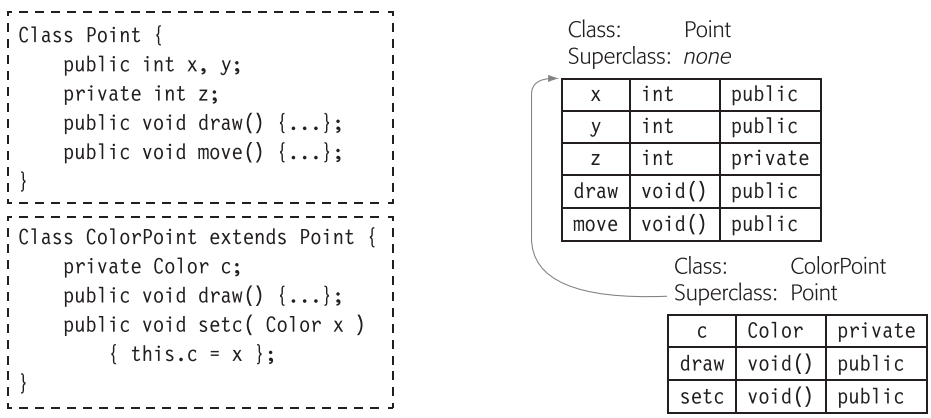
Если структура классов открыта, то компилятору, возможно, придется сгенерировать код, который частично производит разрешение имени во время выполнения, как то имеет место для виртуальных функций в C++. В общем случае простая и часто недорогая ссылка заменяется вызовом более дорогой функции, исполняемой на этапе выполнения, которая разрешает имя и предоставляет надлежащий доступ (для чтения, записи или выполнения).

Построение модели

Обрабатывая определение класса, парсер может (1) внести имя класса в текущую лексическую область видимости и (2) создать новую таблицу для имен,

определенных в этом классе. Поскольку и содержимое класса, и его контекст наследования описываются синтаксически, автор компилятора может использовать синтаксически управляемые действия для построения и заполнения таблицы и включения ее в объемлющую иерархию наследования. Имена членов ищутся в иерархии наследования, а неквалифицированные имена – в лексической иерархии.

Компилятор может использовать механизмы построения таблицы символов, созданные для лексических иерархий, чтобы представить иерархии наследования. На рис. 5.11 показано два определения: класса `Point` и его подкласса `ColorPoint`. Компилятор может включить соответствующие таблицы в путь поиска по иерархии наследования, показанной на рисунке с помощью указателя `SuperClass`. В более запутанных ситуациях, например при наличии лексически вложенных определений классов, просто будут порождаться более сложные пути поиска.



(a) Определения классов

(b) Соответствующие таблицы символов

Рис. 5.11 Таблицы для иерархии наследования

Примеры

Объектно-ориентированные языки различаются терминологией и используемыми объектными моделями.

В C++ структура классов замкнута. По замыслу, имена методов могут быть привязаны к реализациям во время компиляции. C++ включает явный механизм объявления для принудительной привязки во время выполнения – виртуальные функции.

С другой стороны, в JAVA структура классов открыта, хотя плата за изменение структуры классов высока – программа должна вызвать *загрузчик классов*, чтобы импортировать новые определения классов. Компилятор в принципе мог бы разрешать имена, т. е. сопоставлять им реализации, на этапе загрузки программы и выполнять перепривязку после каждого вызова загрузчика классов. Но на практике большинство систем JAVA интерпретируют байт-код и компилируют часто вызываемые методы с помощью JIT-компилятора. Этот

подход обеспечивает высокое качество кода и позднюю привязку. Если загрузчик класса перепишет какое-то определение класса, которое использовалось во время предшествующей JIT-компиляции, то он сможет затребовать перекомпиляцию, сделав недействительным код затронутых методов.

Множественное наследование

Множественное наследование
механизм, позволяющий классу наследовать нескольким непосредственным суперклассам.

Некоторые ООЯ допускают множественное наследование. Такой язык нуждается в синтаксисе, позволяющем программисту указать, что члены *a*, *b* и *c* наследуют определения от суперкласса *x*, тогда как члены *d* и *e* – от суперкласса *y*. Язык также должен как-то разрешать или запрещать бессмысленные ситуации, например когда класс наследует несколько определений одного и того же имени.

Для поддержки множественного наследования компилятору необходима более сложная модель иерархии наследования. Однако построить ее он может, пользуясь теми же строительными блоками: таблицами символов и явными путями поиска. Сложность проявляется в путях поиска.

5.4.3. Видимость

Видимость
Имя *видимо* в точке *p*, если на него можно сослаться в этой точке.

Некоторые языки предоставляют средства управления видимостью имени.

Языки программирования часто предоставляют средства явного управления видимостью, т. е. определения того, в каких частях кода можно определять или использовать некоторое имя. Например, язык C предоставляет ограниченное управление видимостью с помощью ключевого слова `static`. Управление видимостью имеет смысл как в лексических иерархиях, так и в иерархиях наследования.

Ключевое слово C `static` определяет и время жизни, и видимость. Время жизни статической переменной совпадает со временем работы программы, а ее видимость ограничена текущей областью видимости и вложенными в нее. Если объявление расположено вне любой функции, то `static` ограничивает его видимость файлом. (Не будь слова `static`, такое имя было бы видно во всей программе.)

Если слово `static` встречается в объявлении переменной внутри функции, то та его ипостась, которая связана со временем жизни, гарантирует, что значение переменной сохраняется между вызовами этой функции. Ипостась же, связанная с видимостью, никак не проявляется, потому что видимость переменной уже и так ограничена функцией, в которой она объявлена, и вложенными в нее областями видимости.

JAVA предоставляет средства явного управления видимостью с помощью ключевых слов `public`, `private`, `protected` и `default`.

- **public** – метод или член данных, объявленный в разделе `public`, видим в любом месте программы.
- **private** – метод или член данных, объявленный в разделе `private`, виден только в том классе, где объявлен.
- **protected** – метод или член данных, объявленный в разделе `protected`, виден в классе, где объявлен, во всех прочих классах из того же пакета и в любом подклассе, объявленном в другом пакете.
- **default** – метод или член данных, объявленный в разделе `default`, виден в классе, где объявлен, и во всех прочих классах из того же пакета. Если видимость вообще не задана, то предполагается видимость `default`.

Ни `private`, ни `protected` нельзя использовать на верхнем уровне иерархии, потому что они определяют видимость относительно объемлющего класса, а на верхнем уровне никакого объемлющего класса нет.

Строя окружение именования, компилятор должен закодировать атрибуты видимости в модель пространства имен. Типичная реализация включает признак видимости в запись таблицы символов для каждого имени. Компилятор анализирует эти признаки во время поиска в таблице.

Как уже отмечалось, в PYTHON видимость переменной устанавливается с учетом того, является ли определяющее вхождение определением или использованием. (Если это использование, то имя глобально.) Для объектов PYTHON не предлагает никакого механизма управления видимостью их членов данных и кода. Все атрибуты (члены данных) и методы имеют глобальную видимость.

5.4.4. Разрешение имен на этапе компиляции

В процессе трансляции компилятор часто отображает имя лексемы на конкретную сущность, например переменную, объект или процедуру. Для разрешения имени компилятор пользуется таблицей символов, которую построил для представления лексической иерархии и иерархии наследования. Правила языка определяют путь поиска в этих таблицах. Компилятор начинает с самого внутреннего уровня пути поиска и выполняет поиск в каждой таблице, встретившейся на пути, пока либо не найдет имя, либо не потерпит неудачу в самой внешней таблице.

Особенности пути зависят от языка. Если синтаксис имени указывает на то, что это ссылка относительно объекта, то компилятор может начать с таблицы для класса объекта и двигаться вверх по иерархии наследования. Если синтаксис имени указывает на то, что это «обыкновенная» переменная, то компилятор может начать с таблиц для области видимости, в которой встретилась ссылка, и двигаться вверх по лексической иерархии. Если синтаксис языка не позволяет отличить данные-члены объектов от обыкновенных переменных, то компилятор должен построить смешанный путь поиска, который объединяет таблицы способом, моделирующим языковые правила областей видимости.

Компилятор может обслуживать необходимые пути поиска с помощью синтаксически управляемых действий, которые выполняются, когда компилятор входит в области видимости, объявления классов, структур и других агрегатов и выходит из них. Детали, разумеется, сильно зависят от правил компилируемого исходного языка.

ИТОГИ РАЗДЕЛА

Языки программирования предоставляют механизмы управления временем жизни и видимостью имен. Объявления допускают явное задание свойств имени. Помещение объявления в код оказывает прямое влияние на время жизни и видимость, как то определяют правила языка. В объектно-ориентированном языке наследование также влияет на свойства именованной сущности.

Для моделирования этих сложных окружений именования компиляторы применяют два основных инструмента: таблицы символов и пути поиска, которые связывают таблицы друг с другом, образуя иерархию. Компилятор может использовать эти инструменты для построения контекстно-зависимых пространств поиска, моделирующих правила исходного языка.

ВОПРОСЫ

1. Предположим, что компилятор строит отдельную таблицу символов и путь поиска для каждой области видимости. Для простого PASCAL-подобного языка какие действия должен выполнять парсер при входе в каждую область видимости и выходе из нее?
2. При условии что для разрешения имен используется модель таблиц и путей поиска, какова асимптотическая стоимость (a) разрешения локального имени? (b) разрешения нелокального имени? (Можете предполагать, что стоимость поиска в таблице O(1).) Насколько глубоко вложены области видимости в написанных вами программах?

5.5. Информация о типе

Тип
абстрактная категория, определяющая свойства, которыми обладают все представители этого типа.

Из распространенных типов упомянем *целое*, *литеру*, *список* и *функцию*.

Для того чтобы транслировать ссылки в методы доступа, компилятор должен знать, что именно представляет имя. Имя `fee` в исходном языке могло бы относиться к небольшому целому числу или к функции от двух строк литер, которая возвращает число с плавающей точкой, или к объекту класса `fum`. Прежде чем фаза анализа сможет сгенерировать код для манипулирования именем `fee`, она должна знать его фундаментальные свойства, сведенные в *тип*.

Тип – это просто совокупность свойств; все представители типа имеют одни и те же свойства. Например, тип `integer` можно определить как любое целое число i в диапазоне $-2^{63} \le i < 2^{63} - 1$, а тип `red` может быть элементом типа перечисления `colors`, определенного как множество { `red, yellow, blue` }.

Типы могут быть определены правилами; например, объявление структуры в C определяет тип. Тип структуры задает множество объявленных полей

Мы представляем тип структуры как произведение типов составляющих ее полей в порядке их следования.

и их порядок внутри структуры; у каждого поля имеется собственный тип, определяющий его интерпретацию. В языках программирования имеются предопределенные типы, называемые *базовыми*. Большинство языков позволяют программисту конструировать новые типы. Множество типов в данном языке программирования вкупе с правилами, в которых типы используются для определения поведения программы, называется *системой типов*.

Система типов позволяет проектировщикам языка и программистам описывать поведение программы более точно, чем возможно с помощью контекстно-свободной грамматики. Система типов создает второй словарь для описания поведения правильных программ. Рассмотрим, к примеру, выражение JAVA `a + b`. Смысл знака + зависит от типов `a` и `b`. Если `a` и `b` – строки, то оператор + обозначает конкатенацию. Если `a` и `b` – числа, то оператор + обозначает сложение, возможно, с неявным преобразованием к общему типу. Такого рода перегрузка операций требует точной информации о типе.

5.5.1. Применения типов при трансляции

Типы играют важнейшую роль в трансляции, потому что помогают компилятору понять смысл и, следовательно, реализацию исходного кода. Эти знания, более глубокие, чем синтаксис, позволяют компилятору обнаруживать ошибки, которые иначе могли бы проявиться во время выполнения. Во многих случаях компилятор также может сгенерировать более эффективный код, чем было бы возможно без информации о типах.

Совместимость
Говорят, что оператор и его операнды совместимы, если результат применения оператора к этим операндам корректно определен.

Компилятор может использовать информацию о типе, чтобы гарантировать совместимость операторов и операндов, т. е. что оператор корректно определен на типах операндов (например, конкатенация строк может быть не определена на множестве вещественных чисел). В некоторых случаях язык может требовать, чтобы компилятор вставлял код для преобразования несовместимых аргументов к совместимым типам, – этот процесс называется неявным преобразованием. В других случаях (например, при попытке использовать число с плавающей точкой как указатель) определение языка может запрещать такие преобразования; компилятор должен как минимум выдать информативное сообщение об ошибке, чтобы программист понял суть проблемы.

Наличие информации о типе может помочь компилятору транслировать программу в эффективно исполняемый код. Например, в выражении a^x типы a и x определяют, как лучше вычислять выражение. Если x – неотрицательное целое число, то компилятор может сгенерировать последовательность умно-

жений для вычисления a^x. Если же x – вещественное или отрицательное число, то компилятору придется генерировать код, в котором используется более сложная схема вычислений, например разложение в ряд Тейлора. (Более сложная форма может быть реализована обращением к библиотеке.) Аналогично языки, допускающие присваивание целой структуры или целого массива, могут проверять совместимость, чтобы у компилятора была возможность реализовать эти конструкции эффективно.

Сигнатура типа
спецификация типов формальных параметров и возвращаемого значения (или нескольких значений) функции.

Прототип функции
В языке C программист может объявить функции, отсутствующие в файле. Для этого включается объявление, называемое *прототипом функции*.

В более широком масштабе информация о типе играет важную вспомогательную роль в модульном программировании и раздельной компиляции. Модульное программирование оставляет программисту возможность неправильно задать число и типы аргументов функции, реализованной в другом файле или модуле. Если язык требует, чтобы программист указывал *сигнатуру типа* для любой внешней функции (по сути дела, прототип функции в C), то компилятор может сравнить фактические аргументы с сигнатурой типа.

Информация о типе также играет важную роль для сборки мусора (см. раздел 6.6.2). Благодаря ей сборщик мусора узнает о размере каждой сущности в куче и понимает, какие поля объекта являются указателями на другие сущности, которые, возможно, также размещены в куче. Не будь информации о типе, собранной и сохраненной на этапе компиляции, сборщик мусора должен был бы на всякий случай предполагать, что всякое поле может являться указателем, и во время выполнения проверять диапазон и выравнивание, чтобы исключить значения, выходящие за границы допустимого.

Отсутствие информации о типе

Полная информация о типе может быть недоступна вследствие особенностей языка или поздней привязки.

Если информация о типе недоступна в процессе трансляции, то компилятору, возможно, придется генерировать код для проверки типов и выбора кода на этапе выполнения. С каждой сущностью неизвестного типа нужно будет связать тег, в который будет записываться тип во время выполнения. Вместо простого оператора компилятор должен будет генерировать логику ветвления на основе типов операндов, чтобы, с одной стороны, определить тег, а с другой стороны, манипулировать значениями и тегами.

Псевдокод на рис. 5.12 показывает, какой код мог бы сгенерировать компилятор для сложения с проверкой типов и преобразованием во время выполнения. Предполагается наличие трех типов, *SHORT*, *INTEGER* и *LONG INTEGER*. Если операнды имеют одинаковый тип, то выбирается подходящая версия оператора сложения, выполняется арифметическая операция и устанавли-

вается тег. Если типы операндов различны, то вызывается библиотечная функция, которая выполняет полный анализ ситуации, преобразует операнды к нужным типам, складывает преобразованные операнды и возвращает результат и его тег.

```
if (tag(a) = tag(b)) then   // выбрать короткий путь
   switch (tag(a)) into {
      case SHORT:   // использовать сложение коротких целых
         value(c) ← value(a) + value(b)
         tag(c) ← SHORT
         break
      case INTEGER:   // использовать сложение целых
         value(c) ← value(a) + value(b)
         tag(c) ← INTEGER
         break
      case LONG INTEGER:   // использовать сложение длинных целых
         value(c) ← value(a) + value(b)
         tag(c) ← LONG INTEGER
         break
   }
else // выбрать длинный путь
   (c, tag(c)) ← AddMixedTypes(a, tag(a), b, tag(b))
```

Рис. 5.12 Сложение целых с проверкой типов во время выполнения

Конечно, если бы компилятор располагал полной и точной информацией о типе, то мог бы сгенерировать код выполнения операции и необходимых преобразований непосредственно. Тогда теги времени выполнения и соответствующая проверка тегов оказались бы излишними.

5.5.2. Компоненты системы типов

Система типов состоит из четырех основных компонентов: набор базовых, или встроенных, типов, правила построения новых типов из существующих, метод установления эквивалентности двух типов, правила вывода типа выражения на исходном языке.

Базовые типы

Большинство языков включают базовые типы для некоторых (а то и для всех) из следующих видов данных: числа, литеры и булевы величины. Большинство процессоров непосредственно поддерживают эти виды данных. Числа обычно бывают нескольких форматов, например целые и с плавающей точкой, и нескольких размеров, например байт, слово, двойное слово и четверное слово.

Размер «слова» может зависеть от реализации и процессора.

В некоторых языках имеются и дополнительные базовые типы. LISP поддерживает тип рациональных чисел и тип рекурсивного списка. Рациональное

число – это, по сути дела, пара целых чисел, интерпретируемая как их отношение. Список – это либо зарезервированное значение `nil`, либо список, построенный с помощью конструктора `cons`; выражение (`cons first rest`) – это упорядоченный список, в котором `first` – объект, а `rest` – список.

Языки отличаются базовыми типами и операторами, определенными над этими базовыми типами. Например, в C и C++ много разновидностей целых; `long int` и `unsigned long int` имеют одинаковую длину, но разные диапазоны значений. В PYTHON имеется несколько классов строк, которые поддерживают широкий набор операций; напротив, в C типа строки нет, а вместо него программисты используют массив литер. C предоставляет тип указателя, который может содержать произвольный адрес памяти; в JAVA применяется более ограничительная модель ссылочных типов.

Составные и конструируемые типы

Некоторые языки предлагают в качестве базовых типов высокоуровневые абстракции, например словари в PYTHON.

Базовые типы языка программирования являются абстракцией фактических видов данных, поддерживаемых процессором. Однако зачастую базовых типов недостаточно для представления данных из той предметной области, которая интересует программиста, скажем таких абстракций, как графы, деревья, таблицы, записи, объекты, классы, списки, стеки и отображения. Эти высокоуровневые абстракции можно реализовать в виде совокупностей нескольких сущностей, каждая своего типа.

В ООЯ классы можно рассматривать как конструируемые типы. Наследование определяет отношение специализации между типом и подтипом.

Способность конструировать новые типы для представления составных, или агрегатных, объектов – важное свойство многих языков. Типичные конструируемые типы включают массивы, строки, типы перечислений и структуры или записи. Составные типы позволяют программисту организовывать информацию новыми, зависящими от программы способами. Конструируемые типы позволяют языку выражать высокоуровневые операции, например присваивание целой структуры. Они также расширяют возможности компилятора по обнаружению некорректных программ.

Массивы

Массивы – один из самых широкоупотребительных агрегатных типов. Массив группирует в одно целое объекты одного и того же типа и наделяет каждый объект отдельным именем, правда, неявным и вычисляемым, а не задаваемым программистом. Объявление `int a[100][200];` в языке C выделяет память для 100 × 200 = 20 000 целых чисел и позволяет обращаться к ним по имени `a`. Ссылки `a[1][17]` и `a[2][30]` адресуют разные и независимые ячейки памяти. Важное свойство массива состоит в том, что программа может вычислить имя каждого его элемента, пользуясь только числами (или другими упорядоченными дискретными типами) в качестве индексов.

Совместимость массивов
Два массива *a* и *b* совместимы относительно некоторого оператора над массивами, если размерности *a* и *b* имеют смысл для этого оператора.

Например, умножение матриц налагает на массивы иные требования совместимости, нежели сложение матриц.

Поддержка операций с массивами изменяется в широких пределах. Языки FORTRAN 90, PL/I и APL поддерживают присваивание всему массиву или его части. Эти языки также позволяют применять арифметические операции к каждому элементу массива. Для совместимых массивов `x`, `y` и `z` предложение `x = y + z` перезаписывает каждый элемент `x[i,j]` суммой `y[i,j] + z[i,j]`. APL заходит в определении операций над массивами дальше большинства других языков; в нем имеются операторы скалярного произведения, внешнего произведения и нескольких видов свертки. Например, суммирующая свертка `y`, записываемая в виде `x ← +/y`, присваивает `x` скалярную сумму элементов `y`.

Массив можно рассматривать как конструируемый тип, потому что он задается с помощью типа своих элементов. Так, массив целых размера 10×10 имеет тип *двумерного массива целых*. В некоторых языках размерность массива включена в его тип; таким образом, тип массива целых 10×10 отличается от типа массива целых 12×12. При таком подходе операции с массивами, в которых операнды имеют несовместимые размерности, становятся ошибками типизации, а значит, обнаруживаются и диагностируются систематически. В большинстве языков допустимы массивы любого базового типа; в некоторых языках допустимы массивы конструируемых типов, например структур.

Строки

Поддержка строк в разных языках организована по-разному. В PYTHON и PL/I поддерживается несколько видов строк со схожими свойствами, атрибутами и операциями. А в FORTRAN и C строка трактуется просто как вектор литер.

Тип настоящей строки отличается от типа массива в нескольких важных отношениях. Операции, имеющие смысл для строк, например конкатенация, трансляция и вычисление длины, могут не иметь аналогов для массивов. Стандартные операции сравнения можно перегрузить, так что сравнение строк будет работать естественно: `"a" < "boo"` и `"fee" < "fie"`. Реализация похожего сравнения для массивов литер наводит на мысль о применении той же идеи к массивам чисел или структур, для которых никакой аналогии может и не быть. Кроме того, фактическая длина строки может отличаться от размера выделенной для нее памяти, тогда как в большинстве применений массивов используются все выделенные элементы.

Типы перечислений

Многие языки позволяют программисту сконструировать тип, который содержит определенный набор постоянных значений. *Тип перечисления* дает возможность использовать самодокументированные имена небольшого набора констант. Классические примеры – названия дней недели и месяцев. В языке C это можно записать следующим образом:

```
enum WeekDay { Monday, Tuesday, Wednesday,
               Thursday, Friday, Saturday, Sunday};
enum Month   { January, February, March, April,
                May, June, July, August, September,
                October, November, December};
```

Компилятор отображает элементы типа перечисления на разные значения. Элементы типа перечисления упорядочены, поэтому сравнение между элементами одного типа имеет смысл. В примерах выше `Monday < Tuesday` и `June < July`. Сравнение разных типов перечисления бессмысленно – например, сравнение `Tuesday > September` приведет к ошибке типизации. В PASCAL каждый тип перечисления ведет себя так, будто это поддиапазон целых чисел. Например, программист может объявить массив, индексированный элементами типа перечисления.

Структуры и варианты

Структуры, или *записи*, группируют несколько объектов произвольных типов. Элементам структуры обычно даются явные имена. Например, программисту, который реализует дерево разбора на C, могли бы понадобиться узлы с одними или двумя потомками.

```
struct N1 {
  int   Operator;
  int   Value;
  union Node *left;
};
```

```
struct N2 {
  int   Operator;
  int   Value;
  union Node *left;
  union Node *right;
};
```

```
union Node {
  struct N1 one;
  struct N2 two;
};
```

Типом структуры является упорядоченное произведение типов ее элементов. Мы могли описать `N1` и `N2` так:

```
N1: int×int×(Node *)
N2: int×int×(Node *)×(Node *)
```

Эти новые типы должны иметь те же основные свойства, что и базовый тип. В C инкремент указателя на `N1` или приведение указателя к типу `(N1 *)` дает желаемый эффект – поведение, аналогичное тому, что происходит с базовым типом.

В этом примере создан новый тип `Node`, являющийся структурой типа `N1` или `N2`. Таким образом, указатель в узле типа `N1` может ссылаться либо на узел типа `N1`, либо на узел типа `N2`. В PASCAL объединения создаются с помощью вариантных записей. В C используется ключевое слово `union`. Тип `union` – это объединение его компонентов, т. е. `Node` имеет тип $N1 \cup N2$.

Язык и среда выполнения должны выработать общий механизм устранения неоднозначности ссылок. Одно из решений – использовать квалифицированные ссылки, например $p \rightarrow$ `Node.N1.Value` и $p \rightarrow$ `Node.N2.Value`. Вместо этого язык мог бы принять подход PASCAL и потребовать наличия тегов для вариантных записей с явными проверками тегов на этапе выполнения.

Объекты и классы

В объектно-ориентированном языке классы определяют содержимое и форму объектов, а также иерархию наследования, которая используется для разрешения ссылок относительно объектов. Однако на уровне реализации объект выглядит как запись или структура, организация которой описывается определением класса.

Альтернативное представление структур

При классическом представлении каждая структура рассматривается как отдельный тип. Этот подход к структурным типам аналогичен работе с другими агрегатами, например массивами и строками. И кажется естественным. Он проводит различия, полезные для программиста. Например, узел дерева с двумя потомками, вероятно, должен иметь не такой тип, как узел с тремя потомками; скорее всего, они используются в разных ситуациях. При компиляции программы, которая присваивает узел с тремя потомками узлу с двумя потомками, должна диагностироваться ошибка типизации с печатью предупреждения для программиста.

Но с точки зрения системы времени выполнения, рассмотрение каждой структуры как отдельного типа влечет затруднения. Если структуры являются разными типами, то куча содержит множество объектов, принадлежащих произвольному множеству типов. Из-за этого становится трудно рассуждать о программе, которая имеет дело непосредственно с объектами в куче, например о сборщике мусора. Чтобы упростить такие программы, их авторы иногда подходят к структурным типам по-другому.

В этой альтернативной модели все структуры, встречающиеся в программе, рассматриваются как один тип. Отдельные объявления структур создают вариантную форму типа *structure*. Сам же тип *structure* является объединением всех этих вариантов. При таком подходе программа может рассматривать кучу как собрание объектов одного, а не разных типов. Такой взгляд делает код работы с кучей проще для анализа и оптимизации.

Эквивалентность типов

```
struct Tree {
  int value;
  struct Tree *left;
  struct Tree *right;
}

struct Bush {
  int value;
  struct Bush *left;
  struct Bush *right;
}
```

Компилятору необходим механизм для определения того, являются ли два конструируемых типа эквивалентными. (Для базовых типов ответ очевиден.) Рассмотрим объявления структур на языке C, показанные во врезке. Одинаковы ли типы `Tree` и `Bush`? Эквивалентны ли они? Любой язык, включающий конструируемые типы, должен давать однозначный ответ на этот вопрос. Исторически сложилось два подхода.

1. *Эквивалентность по именам* означает, что два типа эквивалентны тогда и только тогда, когда программист выбрал для них одинаковые имена. При этом предполагается, что именование – сознательное действие и что программист вкладывает в имена определенный смысл.

2. *Структурная эквивалентность* означает, что два типа эквивалентны тогда и только тогда, когда они имеют одинаковую структуру. При этом предполагается, что структура важна, а имена – необязательно.

Типы `Tree` и `Bush` эквивалентны структурно, но не эквивалентны по именам.

У обоих подходов есть сторонники и противники. Однако выбор между ними делает проектировщик языка, а не автор компилятора. Автор компилятора должен реализовать подходящее представление типа и подходящую проверку на эквивалентность.

5.5.3. Вывод типа выражения

Компилятор должен каждому выражению и подвыражению назначить какой-то тип. Для простейших выражений, `name` и `num`, тип понятен. Для вычисляемых выражений, содержащих ссылки, компилятор должен вывести тип из комбинации операции и типов ее операндов.

Для этого необходимо определить связь между оператором, типами операндов и типом результата. Концептуально можно рассматривать эту связь как рекурсивную функцию над деревом; на практике же правила варьируются от простых и очевидных до весьма запутанных. Во врезке ниже описаны правила для выражений в языке C++. Поскольку в C++ много базовых типов, правила довольно пространные.

Тип результата выражения зависит от оператора и типов его операндов. Компилятор мог бы назначать типы в процессе обхода дерева разбора снизу вверх. В каждом узле он устанавливал бы тип на основе типа хранящегося в нем оператора и типов сыновних узлов. Альтернативно компилятор мог бы реализовать назначение типов как часть синтаксически управляемого каркаса трансляции.

На рис. 5.13 показан набросок действий, необходимых для назначения типов подвыражениям в синтаксически управляемом каркасе. Предполагается, что функция-тип для оператора α записывается в виде $\mathcal{F}_{\alpha}$. Тогда тип умножения – это просто $\mathcal{F}_{\times}(t_1, t_2)$, где t_1 и t_2 – типы левого и правого операндов ×. Конечно, автор компилятора, скорее всего, передавал бы структурные значения в стеке, так что ссылки `$$`, `$1` и `$3` были бы более сложными.

```
Expr   → Expr + Term      { set_type($$,F+(type($1),type($3))); };
       | Expr - Term      { set_type($$,F-(type($1),type($3))); };
       | Term             { set_type($$,type($1)); };
Term   → Term × Factor    { set_type($$,F×(type($1),type($3))); };
       | Term ÷ Factor    { set_type($$,F÷(type($1),type($3))); };
Factor → ( Expr )         { set_type($$,type($1)); };
       | num              { set_type($$,type($2)); };
       | name             { set_type($$,type(num)); };
                          { set_type($$,type(name)); };
```

Рис. 5.13 Каркас назначения типов подвыражениям

Числовые преобразования в C++

В языке C++, определенном в стандарте ISO 2017, имеется большой и сложный набор правил преобразования [212]. Ниже приведена упрощенная версия правил расширения и преобразования типов числовых значений.

Расширение целых типов: значение литеры или значение нетипизированного перечисления можно расширить до первого целого типа, способного представить все его значения. Целыми типами в порядке от узких к широким являются: `int`, `unsigned int`, `long int`, `unsigned long int`, `long long int` и `unsigned long long int`. (Для типизированного перечисления преобразование допустимо, только если его базовый тип можно преобразовать в целый.)

Расширение типов с плавающей точкой: значение типа `float` можно расширить до типа `double`.

Преобразования целых типов: значение целого типа можно преобразовать в любой другой целый тип, как и значение типа перечисления. Если конечный тип беззнаковый, то результатом является наименьшее целое без знака, конгруэнтное исходному значению. Если конечный тип знаковый, то значение не изменяется, если оно помещается в конечный тип, в противном случае результат *зависит от реализации*.

Преобразования типов с плавающей точкой: значение типа с плавающей точкой можно преобразовать в любой другой тип с плавающей точкой. Если конечный тип способен точно представить исходное значение, то результатом будет это значение. В противном случае может быть выбрано одно из двух соседних значений, а какое именно, зависит от реализации.

Преобразование в булево значение: числовое значение, значение перечисления или значение указателя можно преобразовать в значение типа `bool`. Значение 0, нулевой указатель или нулевой указатель на член преобразуются в `false`; любое другое значение преобразуется в `true`.

Компилятор пытается преобразовать исходное значение в конечный тип, для чего могут потребоваться одновременно расширение и преобразование.

В языке с более сложными правилами вывода компилятор мог бы построить ПП с неполной информацией о типе и выполнить по нему один или несколько проходов для назначения типов подвыражениям.

Роль объявлений

Языки программирования различаются по тому, требуют ли они объявлений. В языке с обязательными объявлениями последние задают конкретный тип для каждой именованной сущности; эти типы, в свою очередь, служат источником информации для вывода типов. В языке без объявлений, например PYTHON или LISP, компилятор должен выводить типы значений из контекста, в котором они встречаются в коде. Например, присваивание `fee ← 'a'` могло бы означать, что `fee` имеет тип, способный содержать одну литеру, тогда как присваивание `fee ← "a"` – что `fee` способен содержать строку литер.

В этом примере предполагается действующее в языке C соглашение о том, что в одиночные кавычки заключается литера, а в двойные – строка.

Языки программирования различаются также тем, в каком месте кода может встречаться объявление. Во многих языках действует правило «объявления перед использованием»: любое имя должно быть объявлено до того, как встретится в исходном коде. Это правило упрощает проверку типов во время трансляции в начальное ПП. Если язык не требует наличия объявления перед использованием, то компилятор вынужден строить начальное ПП, в котором детали типа абстрагированы, а затем выводить типы и с помощью этого абстрактного ПП проверять, что компилятор сможет уточнить операторы и ссылки, подставив правильную информацию о типе.

Выражения смешанного типа

Языки программирования различаются тем, что они ожидают от компилятора в части вставки преобразований типа, когда в коде встречается выражение, содержащее не полностью совместимые друг с другом типы. Например, выражение `a × b` можно определить для случая, когда `a` и `b` оба целые или оба с плавающей точкой, но не тогда, когда `a` целое, а `b` – число с плавающей точкой. Язык может потребовать, чтобы компилятор в этом случае сообщил об ошибке или же чтобы компилятор вставил преобразование. В разделе 7.2.2 обсуждается реализация неявных преобразований.

Например, язык ANSI C++ поддерживает несколько видов целых, различающихся диапазонами представимых чисел. В определении языка есть требование о том, чтобы компилятор вставлял код преобразования одного представления в другое; в этом определении поведение описано с помощью набора правил. Правила определяют преобразования для деления целого на число с плавающей точкой и запрещают деление на строку литер.

Межпроцедурные аспекты вывода типов

Вывод типов выражений зависит от других процедур, образующих исполняемую программу. Даже в простейших системах типов выражения могут содержать вызовы функций. Компилятор должен проверить каждый из таких вызовов и убедиться в совместимости типа каждого фактического аргумента и соответствующего ему формального параметра. Он должен определить тип возвращаемого значения для использования в последующем выводе.

Чтобы проанализировать и понять вызовы процедур, компилятору нужна *сигнатура типа* каждой функции. Например, в стандартной библиотеке C функция `strlen` вычисляет длину строки литер. Ее прототип имеет вид:

```
unsigned int strlen(const char *s);
```

Этот прототип говорит, что `strlen` принимает аргумент типа `char *`. Атрибут `const` означает, что `strlen` не изменяет `s`. Функция возвращает неотрицательное целое. Сигнатуру типа можно записать в виде:

*strlen : const char * → unsigned int*

и читается она так: «`strlen` – функция, которая принимает константную строку литер и возвращает целое без знака».

Другой пример: функция `filter` в SCHEME имеет такую сигнатуру типа:

filter: (α → boolean) × list of α → list of α

`filter` возвращает список, содержащий все элементы входного списка, для которых входная функция возвращает `true`.

То есть `filter` – функция двух аргументов. Первый должен быть функцией, которая отображает некий тип *α* в *boolean*, что записывается в виде *(α → boolean)*, а второй – списком, все элементы которого имеют тип *α*. Получив аргументы таких типов, `filter` возвращает список элементов типа *α*. Функция `filter` дает пример *параметрического полиморфизма*: тип ее результата является функцией от типов аргументов.

Для правильного вывода типов компилятору нужна сигнатура типа каждой функции. Получить ее он может несколькими способами. Компилятор может потребовать, чтобы для компиляции была предъявлена вся программа целиком, исключив тем самым раздельную компиляцию. Или он может потребовать задания сигнатуры типа для каждой функции, что обычно достигается с помощью обязательных прототипов функций. Компилятор может также отложить проверку типов до этапа компоновки или выполнения, когда такая информация будет доступна. Наконец, автор компилятора может включить компилятор в систему разработки программ, которая собирает необходимую информацию. Каждый из этих подходов встречается в реальных системах.

ИТОГИ РАЗДЕЛА

Тип представляет набор свойств, общих для всех значений этого типа. Система типов назначает тип каждому значению в программе. Языки программирования используют типы для установления правильного или неправильного поведения. Хорошая система типов может повысить выразительность языка, способствовать обнаружению тонких ошибок и позволить компилятору избежать проверок типов на этапе выполнения.

Система типов состоит из набора базовых типов, правил построения новых типов из существующих, метода определения эквивалентности двух типов и правил вывода типа выражения. Понятия базовых типов, конструируемых типов и эквивалентности типов должны быть знакомы по большинству языков высокого уровня.

ВОПРОСЫ

1. Каковы базовые типы в вашем любимом языке программирования? Существует ли в нем механизм построения агрегатных типов? Предоставляет ли он механизм создания процедуры, принимающей переменное число аргументов, как то умеет делать функция `printf` из стандартной библиотеки ввода-вывода языка C?
2. Типобезопасность в местах вызова процедур часто основана на использовании прототипов, в которых объявляются аргументы и возвращаемые значения процедуры. Предложите эскиз механизма, который мог бы проверять корректность таких прототипов функций.

Указание: может потребоваться взаимодействие с компоновщиком или системой времени выполнения.

5.6. Размещение в памяти

Имея модель пространства имен и информацию о типе каждой именованной сущности, компилятор может определить размещение данных в памяти. Процесс состоит из двух шагов. Сначала компилятор должен назначить каждой сущности логическую область данных. Это решение зависит от времени жизни сущности и ее видимости. Затем для каждой логической области данных компилятор назначает каждой помещенной в эту область сущности смещение от начала.

5.6.1. Классы памяти и области данных

Компилятор может классифицировать значения, нуждающиеся в памяти, по их времени жизни. Большинство языков программирования позволяют программисту создавать значения, принадлежащие как минимум следующим *классам памяти*: *автоматический*, *статический* и *нерегулярный*. Компилятор сопоставляет имени переменной область памяти на основе ее времени жизни, класса памяти и видимости (см. раздел 4.7.3).

Автоматические переменные

Время жизни автоматической переменной *a* совпадает со временем жизни объемлющей ее области видимости. Поэтому ее можно хранить в области локальных данных, связанной с областью видимости. Например, если *a* объявлена в процедуре *p*, то компилятор может хранить *a* в области локальных данных *p*. (Если область видимости содержится внутри *p*, то компилятор может зарезервировать место для этой области внутри области локальных данных *p*.) Если переменная *a* локальная, скалярная и однозначная, то компилятор может решить, что ее нужно хранить в регистре (см. раздел 4.7.2).

Запись активации
область памяти, зарезервированная для хранения управляющей информации и области локальных данных, связанной с вызовом процедуры.

Для управления выполнением процедуры *p* компилятор должен позаботиться о том, чтобы с каждым вызовом *p* был связан небольшой блок памяти, где будет храниться управляющая информация, необходимая для вызова и возврата. Эта *запись активации* (ЗА) также будет содержать аргументы, переданные *p*. Записи активации, в принципе и на практике, создаются, когда поток управления входит в процедуру, и уничтожаются, когда он покидает процедуру.

Указатель на запись активации
На этапе выполнения код хранит указатель на текущую ЗА. Указатель на запись активации (УЗА) почти всегда находится в регистре, чтобы ускорить доступ.

Компилятор может поместить область локальных данных *p* в ее ЗА. При каждом вызове *p* создается новая ЗА, а вместе с ней и новая область локальных данных. Такая организация гарантирует, что время жизни области локальных данных совпадает со

временем жизни ЗА. При этом корректно обрабатываются рекурсивные вызовы; для каждого вызова создается новая область локальных данных. Помещение области локальных данных в ЗА обеспечивает эффективный доступ к локальным переменным с помощью указателя на запись активации (УЗА). В большинстве реализаций область локальных данных располагается в начале или в конце ЗА процедуры (см. раздел 6.3.1).

Статические переменные

Время жизни статической переменной *s* простирается от момента первого определения *s* в исполняемой программе и до момента последнего использования значения *s*. Промежуток между первым определением и последним использованием *s* может быть очень коротким или совпадать с полным временем выполнения программы. Атрибут `static` обычно означает, что время жизни совпадает со всем временем выполнения – от начала до конца.

Языки программирования поддерживают статические переменные с помощью различных ограничений видимости. Глобальная переменная является статической; ее область видимости охватывает несколько не вложенных друг в друга процедур. Видимость статической переменной, объявленной внутри процедуры, совпадает с процедурой (включая все вложенные области видимости); переменная сохраняет свое значение между несколькими вызовами процедуры, как и глобальная переменная. В языке C слово `static` используется для создания файловой области видимости; значение существует на протяжении всего времени выполнения программы, но видимо только процедурам, определенным в том же файле.

Компиляторы создают различные области данных для статических переменных. В принципе, программа могла бы реализовать отдельные области данных для каждой статической переменной, а могла бы вместо этого объединить их все в одну область. Автор компилятора должен решить, как статические переменные будут отображаться на отдельные области данных. Простой подход – создать по одной области статических данных на каждый файл и положиться на встроенный в компилятор механизм разрешения имен для поддержки ограничений видимости.

Компиляторы обычно используют конструкции языка ассемблера для создания и инициализации областей статических данных, поэтому выделение, инициализация и освобождение памяти на этапе выполнения ничего не стоят. Компилятор должен создать области глобальных данных таким образом, чтобы системный компоновщик отображал все ссылки на данное глобальное имя в один и тот же адрес памяти – в этом и состоит смысл «глобальной» видимости.

Нерегулярные сущности

Время жизни некоторых значений управляется программой в том смысле, что код явно выделяет для них память. (Освобождение памяти может производиться явно или неявно.) Ключевое отличие заключается в том, что

Если для некоторого значения память в куче выделяется ровно один раз, то либо программист, либо компилятор могут преобразовать его в статическое.

Куча
область памяти, отведенная для нерегулярных сущностей и управляемая с помощью библиотеки времени выполнения.

выделение и освобождение памяти происходят в моменты, никак не связанные со временем жизни конкретной процедуры, и на протяжении одного выполнения программы это может происходить много раз.

Библиотека поддержки компилятора должна предоставлять механизм выделения и освобождения памяти для подобных нерегулярных сущностей. Обычно для таких целей используется *куча времени выполнения*. Управление кучей может быть явным – посредством вызовов таких функций, как `malloc` и `free` в LINUX. Куча также может быть *управляемой*, когда освобождение памяти производится неявно с применением таких приемов, как сборка мусора или подсчет ссылок.

Хотя память для самих сущностей фактически может находиться в куче, исходный код обычно требует, чтобы имя содержало ссылку или начинало цепочку ссылок. Таким образом, связанный список может состоять из автоматической локальной переменной, скажем `root`, которая содержит указатель на первый элемент списка. Для `root` необходимо отвести место в регистре или в области локальных данных, тогда как элементы списка могли бы размещаться в куче.

Временные значения

Оптимизация может продлить время жизни временного значения. Если код повторно вычисляет произведение `b × c`, то компилятор мог бы сохранить его значение, вместо того чтобы вычислять его дважды (см. раздел 8.4.1).

В процессе выполнения программа вычисляет много значений, которые никогда не хранятся в именованных областях памяти. Например, вычисляя выражение `a - b × c`, откомпилированный код вычисляет значение `b × c`, но нет никаких семантических причин это значение возвращать. Поскольку такие временные значения не имеют имен, программист не может использовать их повторно. Время их жизни коротко.

Если временное значение помещается в регистр, то компилятор мог бы там и сохранить его. Но некоторые временные значения в регистр не помещаются. А бывает и так, что длина значения неизвестна. Например, если `d` и `e` – строки неизвестной длины, а `+` – оператор конкатенации, то одна из схем вычисления `length(d + e)` подразумевает создание временной строки `d + e`, также неизвестной длины.

Компилятор может поместить большие значения известной или ограниченной длины в область локальных данных. Если же ограничить значение сверху нельзя, то компилятору, возможно, придется сгенерировать код, который выделяет память во время выполнения для размещения значения в куче.

5.6.2. Размещение в виртуальном адресном пространстве

Виртуальное адресное пространство
Во многих системах у каждого процесса имеется адресное пространство, изолированное от других процессов. Эти адресные пространства виртуальны в том смысле, что не привязаны к физической памяти (см. рис. 5.15).

Компилятор должен спланировать, как код будет использовать память во время выполнения. В большинстве систем каждая программа работает в отдельном *виртуальном адресном пространстве*, т. е. выполняется в своем собственном защищенном диапазоне адресов. Операционная система совместно с оборудованием отображает это виртуальное адресное пространство на физические адреса способом, прозрачным для программы; компилятору интересны только виртуальные адреса.

Организация виртуального адресного пространства – результат соглашения между операционной системой, оборудованием и компилятором. Хотя мелкие детали зависят от реализации, в большинстве систем организация такая, как показано на рис. 5.14. Адресное пространство разделено на четыре области.

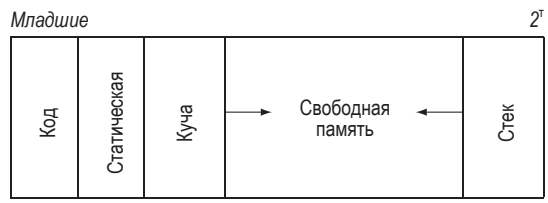


Рис. 5.14 Организация виртуального адресного пространства

Код. В одном конце адресного пространства компилятор размещает исполняемый код. Как правило, размер откомпилированного кода известен и редко изменяется во время выполнения. Если же его размер все-таки изменяется, то новый код размещается в блоке памяти, выделенном в куче.

Статическая. В этой области хранятся статически определенные сущности, к которым относятся глобальные и статические переменные. Размер статической области может быть определен на *этапе компоновки*, когда весь код и данные объединяются для построения исполняемого образа.

Куча. Куча представляет собой область переменного размера, память из которой выделяется явно под управлением программы. Динамически создаваемые сущности, например структуры данных или объекты (в ООЯ) переменного размера, обычно размещаются в куче. Освобождение памяти может быть неявным, с применением сборщика мусора или механизма подсчета ссылок, или явным, посредством вызова библиотечной функции.

В некоторых случаях память для записей активации должна выделяться из кучи (см. раздел 6.3.1).

Стек. Чаще всего вызовы процедур обслуживаются согласно дисциплине *последним пришел, первым ушел*. То есть код вызывает процедуру, а эта процедура возвращает управление. При таком подходе записи активации можно размещать в стеке, что упрощает выделение, освобождение и повторное использование памяти. Стек помещается в конце памяти, противоположном куче, так что все свободное место оказывается между ними.

Стек и куча растут навстречу друг другу. Такая организация позволяет эффективно использовать свободное место между ними.

С точки зрения компилятора, это виртуальное адресное пространство представляет собой целостную и законченную картину. Однако в современных вычислительных системах одновременно исполняется много программ, поочередно занимая процессор. Операционная система отображает несколько виртуальных адресных пространств в одно физическое, поддерживаемое процессором. На рис. 5.15 показана эта более общая картина. Каждая программа изолирована в своем личном виртуальном адресном пространстве и может вести себя так, будто вся машина в ее распоряжении.

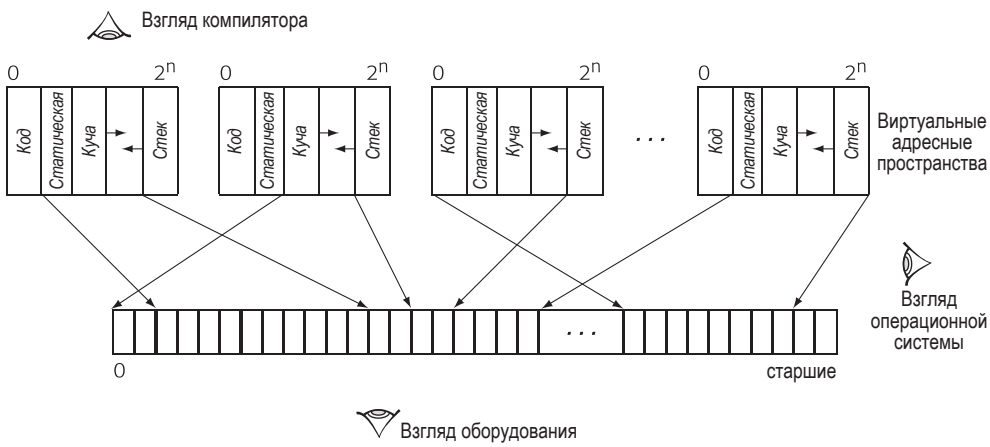


Рис. 5.15 Различные взгляды на адресное пространство

Страница
основная единица выделения памяти в виртуальном адресном пространстве.

Операционная система отображает виртуальные страницы в физические страничные рамки.

Одно виртуальное адресное пространство может занимать разрозненные страницы в физической памяти; например, адреса 100 000 и 200 000 в виртуальном адресном пространстве программы необязательно отстоят друг от друга на 100 000 байт в физической памяти. На самом деле физический адрес, ассоциированный с виртуальным адресом 100 000, может быть больше физического адреса, ассоциированного с виртуальным адресом 200 000. Отображение виртуальных адресов в физические организуется совместными усилиями операционной системы и оборудования. И почти во всех отношениях оно выходит за пределы сферы интересов компилятора.

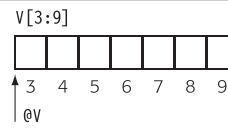
5.6.3. Назначение памяти

Имея множество переменных в конкретной области данных, компилятор должен назначить каждой место в памяти. Если автор компилятора намеревается по максимуму использовать регистры, то компилятор должен сначала найти все однозначные значения, помещающиеся в регистр, и назначить им уникальные виртуальные регистры (см. раздел 4.7.2). Затем он назначает каждому неоднозначному значению смещение от начала области данных. В разделе 5.6.5 описан метод размещения областей данных, минимизирующий влияние аппаратных ограничений на выравнивание.

Внутреннее размещение массивов

Хотя массивы были добавлены в FORTRAN для моделирования матриц при числовых расчетах, у них есть и много других применений.

В большинстве языков программирования имеются массивы – агрегатные структуры определенной размерности, все элементы которых имеют один и тот же тип. В процессе размещения данных в памяти компилятор должен знать, куда он поместит каждый массив. Он также должен понимать, в какой момент устанавливается размер массива и как вычисляются его требования к памяти. Ответы на эти вопросы зависят в том числе от схемы, используемой для размещения элементов массива в памяти.



Размещение вектора в памяти

Компилятор может разместить одномерный массив, или *вектор*, в множестве соседних ячеек памяти. Если диапазон индексов простирается от младшего *low* до старшего *high*, то для размещения вектора понадобится (*high* – *low* + 1) × *w* соседних байтов памяти, где *w* – ширина элемента в байтах. Адрес `V[i]` равен `@V` + (*i* – *low*) × *w*, где `@V` – адрес первого элемента `V`.

В разделе 7.3.2 обсуждается вычисление адресов для каждого из этих размещений.

При двух и более измерениях язык должен определить способ размещения массива в памяти. На рис. 5.16 показано три используемых на практике варианта. На рисунке (a) мы видим концептуальное представление массива 3×4.

В порядке *по строкам* массив размещается как последовательность строк (рис. 5.16(b)). Такой порядок применяется во многих языках. В порядке по *столбцам* массив размещается как последовательность столбцов (рис. 5.16(d)). Такой порядок применяется в FORTRAN. Если массив состоит из *c* столбцов и *r* строк, а каждый элемент занимает *w* байт, то оба этих размещения требуют *c* × *r* × *w* байт непрерывной памяти.

Последний вариант – разместить массив как последовательность векторов указателей, как показано на рис. 5.16(c)). Такая схема используется в JAVA. Здесь последнее измерение массива размещается в соседних ячейках,

а остальные измерения представлены векторами указателей. Для массива с c столбцами и r строками требуется $c \times r \times w$ байт для данных плюс $r \times p$ байт для указателей, где w – размер элемента массива, а p – размер указателя. Отдельные строки и столбец указателей не обязаны занимать непрерывную область памяти.

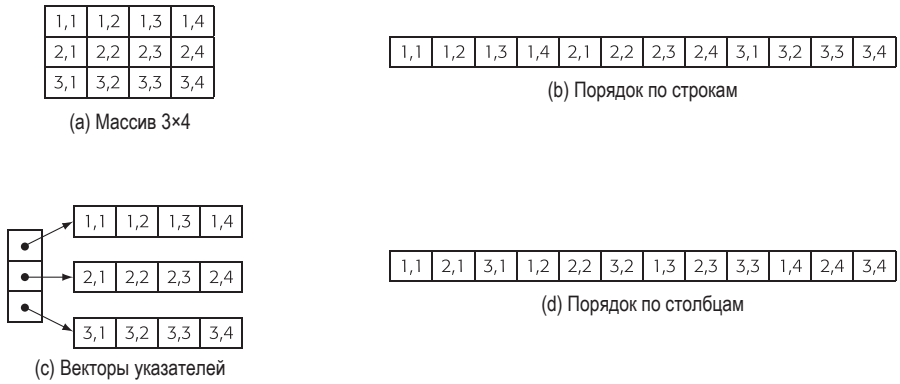


Рис. 5.16 Варианты размещения двумерного массива в памяти

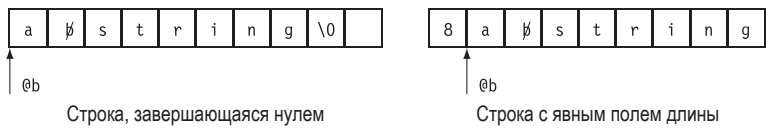
Внутренняя организация строк

Операции со строками обсуждаются в разделе 7.6.

Большинство языков программирования поддерживают строки в той или иной форме. Наиболее употребительны строки литер, но встречаются и строки элементов других типов. Представление строк несколько сложнее, чем вектора, потому что на этапе выполнения строковая переменная может принимать строковые значения разной длины. Поэтому в представлении строки должно храниться ее текущее содержимое и его длина. Кроме того, может храниться максимальная длина строки, которую может содержать данное представление.

Глиф b/ представляет пробел.

Два распространенных представления – строка, завершающаяся нулем, и строка с полем длины. В строке, завершающейся нулем, используется вектор элементов со специальным маркером конца строки. Такое представление впервые было введено в языке C, за которым потянулись и другие.



В представлении строки справа поле длины хранится в отдельном поле. Требования к памяти в этих двух представлениях немного различаются; в строке с завершающим нулем хранится один дополнительный элемент,

а в строке с явной длиной нужно целочисленное поле, достаточно большое для хранения максимальной длины строки.

Но настоящее различие между этими представлениями – стоимость вычисления длины строки. В случае строки с завершающим нулем сложность этой операции составляет O(n), где n – длина строки, а в случае строки с явной длиной она составляет O(1). Это различие переносится и на другие операции, которым нужно знать длину строки, в частности на конкатенацию. Оно играет критически важную роль для проверки диапазона (см. раздел 7.3.3).

Внутренняя организация структур

Компилятор должен также размещать в памяти структуры и объекты. В большинстве языков содержимое объявления структуры считается новой областью видимости. Программист может использовать любые имена для полей, а правила областей видимости гарантируют их корректную интерпретацию. Для каждого поля в объявлении структуры выделяется память; компилятор должен сопоставить полю смещение от начала структуры.

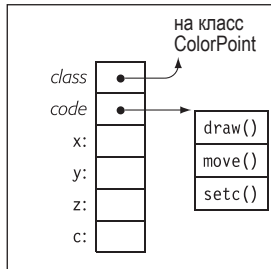
Языки программирования различаются по тому, считают ли они, что текст объявления структуры определяет ее организацию, или нет. В пользу обеих точек зрения существуют убедительные аргументы. Если объявление диктует размещение в памяти, то компилятор назначает полям смещения в том порядке, в каком поля объявлены. Если же размещением в памяти управляет компилятор, то он может переупорядочивать поля с целью уменьшить непроизводительные потери памяти.

Языки системного программирования часто считают описанное в объявлении размещение единственно верным, чтобы программа могла реализовать интерфейс с оборудованием, например через блок управления устройством, структура которого определена производителем и не подлежит изменению.

Внутренняя организация записей объектов

В объектно-ориентированном языке у каждого объекта имеется собственная запись объекта (ЗО). Поскольку время жизни объекта нерегулярно, ЗО обычно находятся в куче. ЗО содержит данные-члены, определенные в классе объекта, а также указатель на его класс и во многих реализациях вектор методов класса. В случае наследования ЗО должна также включать данные-члены, унаследованные от его суперклассов, и средства доступа к методам суперклассов.

Во врезке показана организация ЗО для экземпляра класса `ColorPoint` на рис. 5.11. В ЗО отведена память для каждого члена данных объекта, а также указателей на ЗО класса `ColorPoint` и на вектор видимых методов `ColorPoint`.



Запись объекта для экземпляра класса ColorPoint

Основная трудность при организации объекта проистекает из того факта, что методы суперкласса должны работать при вызове от имени объектов подкласса. Чтобы обеспечить эту возможность, смещения данных-членов суперкласса в объекте подкласса должны быть такими же, как в суперклассе. В случае одиночного наследования это достигается с помощью стратегии *префиксной организации*. Объект подкласса размещается в памяти так, что подобъект, соответствующий суперклассу, находится в начале, т. е. является префиксом. Смещения данных-членов предков в цепочке наследования остаются согласованными, а данные-члены текущего класса добавляются в конец ЗО.

Мы используем термины «вектор методов» и «вектор кода» как синонимы.

Чтобы уменьшить потребление памяти, в большинстве реализаций вектор методов хранится в ЗО класса, а не копируется в ЗО каждого объекта. На рис. 5.17 показаны ЗО двух экземпляров класса `ColorPoint`, а также ЗО этого класса. Связывание ЗО объектов `CPOne` и `CPTwo` непосредственно с вектором методов класса `ColorPoint` уменьшает требования к памяти без каких-либо накладных расходов. Конечно, смещения элементов векторов методов должны быть согласованы с иерархией наследования; как и раньше, для одиночного наследования пригодна префиксная организация.

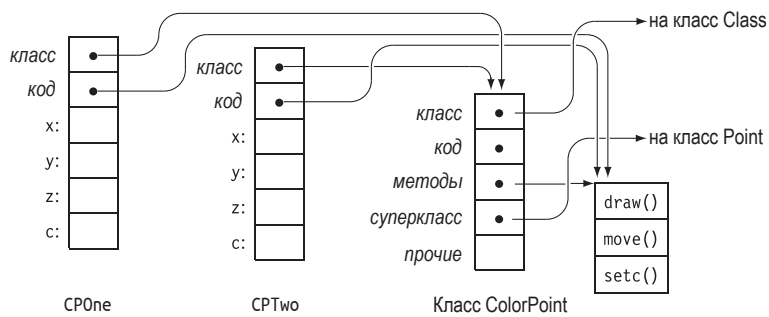


Рис. 5.17 Несколько экземпляров класса `ColorPoint`

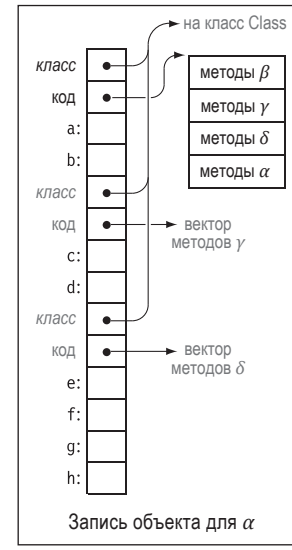
Детали имеют значения

В конструировании компиляторов детали имеют значение. В качестве примера рассмотрим два класса, α и его подкласс β. При планировании размещения записей объектов β в памяти включаются ли закрытые (`private`) члены α? Поскольку они закрыты, объект класса β не может обращаться к ним напрямую.

Объекту класса β понадобятся закрытые члены α, если α предоставляет открытые методы, которые читают или записывают закрытые члены. Аналогично, если размещение ЗО в памяти изменяется, то закрытые члены могут понадобиться, чтобы открытые члены имели правильные смещения в ЗО класса β (даже если нет никакого механизма чтения их значений).

Для упрощения поиска ЗО может содержать полностью инициализированный вектор кода с указателями на методы класса и суперклассов.

Организация записи объекта в случае множественного наследования



Множественное наследование усложняет организацию ЗО. В откомпилированном коде метода суперкласса смещение зависит от размещения в памяти ЗО этого суперкласса. Различные промежуточные суперклассы могут назначать конфликтующие смещения своим членам. Чтобы устранить конфликты, компилятор должен принять несколько более хитрую схему: использовать различные указатели на ЗО с методами из разных суперклассов.

Рассмотрим класс α, наследующий нескольким суперклассам: β, γ и δ. Чтобы разместить в памяти ЗО объекта класса α, реализация должна сначала определиться с порядком суперклассов α; допустим, выбран порядок β, γ, δ. Тогда она размещает ЗО класса α, так чтобы вся ЗО класса β, включая указатели на класс и вектор методов, была префиксом α. После этого она размещает ЗО класса γ, а за ней ЗО класса δ. В конце добавляются данные-члены α. Вектор методов строится путем дописывания друг за другом унаследованных методов в порядке следования классов, а в конце оказываются методы α. Эта организация показана во врезке; видно, что указатели на классы и векторы методов β и γ находятся в середине ЗО.

На рисунке предполагается, что в каждом классе определено два члена данных: в β – `a` и `b`, в γ – `c` и `d`, в δ – `e` и `f`, в α – `g` и `h`. Вектор кода для α указывает на вектор, содержащий все методы, определенные в α или унаследованные им.

На этапе выполнения метод из класса β найдет все данные-члены на местах с ожидаемыми смещениями – такими же, как в объекте класса β. Аналогично метод из класса α найдет свои данные-члены на местах со смещениями, которые были известны во время его компиляции.

Однако для членов γ или δ смещения данных-членов неверны. Компилятор должен скорректировать указатель на ЗО, так чтобы он указывал в нужное место ЗО. Во многих системах это достигается с помощью *функции-батута*, которая просто увеличивает указатель на ЗО, затем вызывает метод, а после возврата из метода уменьшает указатель на ЗО и возвращает управление.

> Методы, откомпилированные для β, γ или δ, не видят члены, определенные в α. Таким образом, код может безнаказанно корректировать указатель на ЗО.

5.6.4. Место назначения памяти в процессе трансляции

Обдумывая трансляцию языка, автор компилятора стоит перед выбором. Можно спроектировать компилятор, так что он будет делать максимально много на синтаксически управляемой стадии, или построить начальное ПП и поручить последующим проходам по ПП завершить трансляцию. Важную роль в этом выборе играет время, когда принимаются решения о размещении в памяти.

Автор компилятора может использовать действие в середине продукции вида

Body → *Decls Execs*

где *Decls* выводит объявления, а *Execs* – исполняемые предложения.

1. Некоторые языки требуют, чтобы все переменные были объявлены до первого исполняемого предложения. Компилятор может собрать всю информацию о типах и символах в процессе обработки объявлений. И прежде чем приступить к первому исполняемому предложению, может спланировать размещение в памяти, что позволит сгенерировать конкретный код для ссылок.
2. Если язык требует объявлений, но не настаивает на определенном порядке, то компилятор может строить таблицу символов в процессе разбора и генерировать ПП с абстрактными ссылками. После разбора он может произвести вывод типов, а потом размещение в памяти. Затем он может уточнить ПП и сделать ссылки более конкретными.
3. Если язык не требует объявлений, то компилятор вынужден строить ПП с абстрактными ссылками. Затем компилятор может выполнить более сложный (возможно, итеративный) вывод типов по ПП и далее – размещение в памяти. И наконец, он может уточнить ПП и сделать ссылки более конкретными.

Выбор подхода зависит от правил исходного языка и предпочтений автора компилятора. Многопроходный подход может упростить код самого компилятора.

5.6.5. Ограничения выравнивания и дополнение

Ограничение выравнивания
Большинство процессоров налагают ограничения на выравнивание значений, зависящие от типа. Например, восьмибайтовое целое, возможно, должно начинаться по адресу, кратному 8.

Архитектура набора команд налагает ограничения на выравнивание значений. (В этом обсуждении мы будем предполагать, что байт состоит из восьми бит и что слово содержит четыре байта.) Для каждого поддерживаемого оборудованием типа ISA может ограничивать множество адресов, по которым может храниться значение этого типа. Например, 32-разрядное число с плавающей точкой, возможно, должно начинаться на границе слова, т. е. по адресу, кратному 4. Аналогично 64-разрядное целое, возможно, должно начинаться на границе двойного слова, т. е. по адресу, кратному 8.

У компилятора есть два механизма для соблюдения ограничений выравнивания. Во-первых, он может контролировать выравнивание в начале каждой области данных. В большинстве языков ассемблера имеются директивы, диктующие выравнивание на границу двойного или четверного слова, которые ставятся в начале области данных. Такие псевдооперации гарантируют, что каждая область данных начинается по адресу с нужным выравниванием.

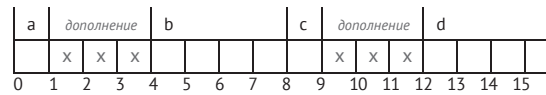
Во-вторых, компилятор управляет размещением внутри области данных, т. е. назначает смещения каждому значению, хранящемуся в этой области. И таким образом он гарантирует, что каждое значение имеет правильное смещение. Например, значение, нуждающееся в выравнивании на границу двойного слова, должно иметь смещение, кратное восьми.

Рассмотрим переменную *x*, хранящуюся со смещением *offset* от начала области данных *base*. Если адрес *base* выровнен на границу четверного слова, то *base* mod 16 = 0. Если *offset* mod 8 = 0, то адрес *x*, равный *base* + *offset*, выровнен на границу двойного слова, т. е. (*base* + *offset*) mod 8 = 0.

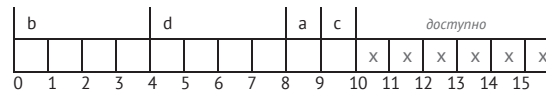
Производя размещение в области данных, компилятор должен удовлетворить всем ограничениям выравнивания. Чтобы добиться надлежащего выравнивания, он может вставлять пустое место между значениями. На рис. 5.18(a) показаны длины и ограничения для простого примера с четырьмя переменными. На рисунке (b) показано размещение, получающееся, когда компилятор назначает переменным смещения в алфавитном порядке. Использовано 16 байт, из них на дополнение потрачено 6 байт. На рисунке (c) показано альтернативное размещение, в котором использовано десять байт и нет никакого дополнения. В обоих случаях сколько-то места может быть потеряно перед началом следующей сущности.

Имя	Байтов	Ограничение
a	1	@a mod 1 = 0
b	4	@b mod 4 = 0
c	1	@c mod 1 = 0
d	4	@d mod 4 = 0

(a) Переменные и их выравнивание



(b) = Размещение с непроизводительным расходованием памяти



(c) Улучшенное размещение

Рис. 5.18 Проблемы выравнивания при назначении смещения от начала области данных

Чтобы создать размещение на рисунке (c), компилятор может построить список имен в области данных, а затем отсортировать его по ограничениям выравнивания – от наибольшей до наименьшей границы выравнивания. Далее он может назначать именам смещения в порядке сортировки. Если для выравнивания очередного имени на нужную границу необходимо вставить байты дополнения, то он может заполнить этот промежуток именами с меньшими требованиями к выравниванию из конца списка.

ИТОГИ РАЗДЕЛА

Для каждой сущности времени выполнения компилятор должен решить, где она будет размещаться в памяти и когда эту память нужно будет выделить. Свое решение компилятор основывает на времени жизни и видимости сущности. Каждому имени сопоставляется один из нескольких классов памяти. Для объектов с предсказуемым временем жизни эти решения определяет класс памяти.

Сущности с непредсказуемым временем жизни компилятор обычно размещает в куче. Память для них выделяется явно, а ссылки на сущности в куче обычно производятся косвенно, через переменные с регулярным временем жизни.

ВОПРОСЫ

1. В языке C файл может содержать переменные, статические на уровне файла или процедуры. Должен ли компилятор создавать отдельные области данных для этих двух классов видимости?
2. Рассмотрите короткий фрагмент C-кода, представленный в рамке. В нем именуются три значения: `a`, `b` и `*b`. Какие из этих значений неоднозначны? А какие однозначны?

```
void fee() {
    int a, *b;
    ...
    b = &a;
    ...
}
```

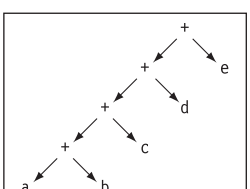
5.7. Дополнительные темы

Эта глава посвящена механизму синтаксически управляемой трансляции и его использованию при построении компилятора. В качестве примеров обсуждалась трансляция выражения и предложений `if-then-else`, модели окружения именования в исходной программе, простой подход к проверке типов и размещение данных в памяти. В этом разделе мы обсудим еще три вопроса.

В первом подразделе рассматривается связь между направлением рекурсии в грамматике и ассоциативностью. Во втором – связь между дизайном языка и выводом и проверкой типов. И в последнем приводится краткий обзор связи между смещениями от начала кеша и производительностью.

5.7.1. Структура грамматики и ассоциативность

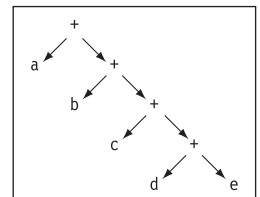
В главе 3 мы встречали леворекурсивный и праворекурсивный варианты грамматики выражений, а также обсуждали преобразование для исключения левой рекурсии. При этом было отмечено, что преобразование сохраняет ассоциативность. В этом разделе исследуется связь между рекурсией, ассоциативностью, структурой ПП и глубиной стека разбора. Рассмотрим следующие простые грамматики для сложения имен.



Леворекурсивное АСД

Expr → *Expr* + `name`
| `name`
Леворекурсивная грамматика

Expr → `name` + *Expr*
| `name`
Праворекурсивная грамматика



Праворекурсивное АСД

Для входного выражения `a + b + c + d + e` эти две грамматики приводят к существенно различным АСД, показанным во врезке. И если значения находятся на границе допустимого диапазона, то эти деревья могут дать различные результаты.

Вычисление в прямом порядке АСД для леворекурсивной грамматики дает `(((a + b) + c) + d) + e`, а для праворекурсивной грамматики – `(((d + e) + c) + b) + a`. В случае сложения, являющегося коммутативной и ассоциативной операцией, эти результаты будут численно отличаться, только если слагаемые очень малы или очень велики.

При использовании генератора LL(1)-парсеров, когда левая рекурсия неприемлема, автор компилятора может добиться левой ассоциативности, написав сначала леворекурсивную грамматику, а затем преобразовав ее в праворекурсивную. В случае генератора LR(1)-парсеров автор компилятора может выбрать левую или правую рекурсию в зависимости от обстоятельств.

Глубина стека

В общем случае левая рекурсия может уменьшить потребную глубину стека. Рассмотрим, что происходит, когда LR(1)-парсер обрабатывает выражение `a + b + c + d + e` с помощью каждой из двух показанных выше грамматик.

1. *Леворекурсивная грамматика.* В этом случае `a` переносится в стек и сразу сворачивается *Expr*. Затем `+` и `b` переносятся в стек и *Expr* `+ b` сворачивается в *Expr*. Далее эти действия продолжаются: `+` и `name` переносятся в стек, и левый контекст сворачивается в *Expr*. В процессе обработки строки максимальная глубина стека достигала трех, а средняя была равна 1.8.
2. *Праворекурсивная грамматика.* В этом случае сначала все токены переносятся в стек: `(a, +, b, +, c, +, d, +, e)`. Затем `e` сворачивается в *Expr* по второму правилу. Далее выполняется последовательность четырех сверток в соответствии с первой продукцией: `d` + *Expr* `a` *Expr*, `c` + *Expr* в *Expr*, `b` + *Expr* в *Expr* и `a` + *Expr* в *Expr*. В процессе обработки строки максимальная глубина стека достигала девяти, а средняя была равна 4.8.

Праворекурсивная грамматика требует больше места в стеке; максимальная глубина ограничена только длиной выражения. С другой стороны, в случае леворекурсивной грамматики максимальная глубина стека зависит от грамматики, а не от входного потока.

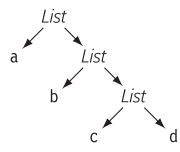
Построение списков

Те же проблемы возникают в случае списков элементов, например списка предложений в блоке. Автор компилятора может использовать в грамматике левую или правую рекурсию.

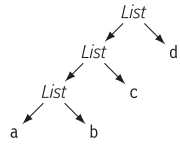
```
List → List ; stmt
     | stmt
```
Леворекурсивная грамматика

```
List → stmt ; List
     | stmt
```
Праворекурсивная грамматика

В случае леворекурсивной грамматики используемое в стеке место ограничено, а в случае праворекурсивной оно пропорционально длине списка. Если список короткий, то место в стеке – не проблема. Но для длинных списков – скажем, когда блок содержит сотни или тысячи предложений – разница может оказаться весьма существенной. Поэтому автору компилятора лучше остановиться на леворекурсивной грамматике.



Праворекурсивное АСД



Леворекурсивное АСД

Проблема при таком подходе возникает, когда компилятор строит структуру данных для представления списка. Рассмотрим простое абстрактное синтаксическое дерево для списка с четырьмя элементами: (a b c d). АСД для праворекурсивной грамматики отражает наше интуитивное понимание списка предложений: a – первый элемент, d – последний. Дерево для леворекурсивной грамматики представляет ту же самую информацию: предложения упорядочены слева направо. Но порядок вложенности интуитивно не столь очевиден, как для праворекурсивной версии АСД. Код обхода списка также становится менее очевидным.

Во многих случаях автор компилятора хотел бы использовать леворекурсивную грамматику из-за ограниченного места в стеке, но при этом построить АСД, которое естественно возникает из праворекурсивной грамматики. Решение состоит в том, чтобы построить конструктор списка, который будет добавлять элементы в конец списка. Прямолинейная реализация этой идеи – обходить список при каждой свертке, но тогда время работы конструктора было бы равно $O(n^2)$, где n – длина списка.

Располагая правильным набором конструкторов списка, автор компилятора может организовать построение праворекурсивного АСД из леворекурсивной грамматики. Рассмотрим следующую синтаксически управляемую схему:

```
List  → List'          { $$ ← RemoveListHeader() ; }

List' → List' ; stmt   { AddToEnd($1, $2) ; }
      | stmt           { AddToEnd( MakeListHeader(), $1) ; }
```

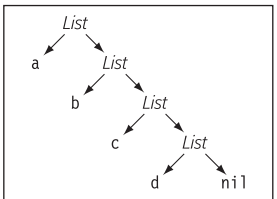
Мы разработали эту схему для парсера ILOC, написанного на bison. Первоначальная праворекурсивная версия переполняла стек разбора, когда обработка входа требовала более 64 000 операций.

В схеме используется три вспомогательные функции.

- MakeListHeader() строит заголовочный узел, содержащий указатели на начало и конец списка. Она возвращает указатель на созданный узел.

- `RemoveListHeader(x)` принимает заголовочный узел *x*. Она возвращает указатель на начало списка и уничтожает заголовочный узел.
- `AddToEnd(x, y)` принимает заголовочный узел *x* и элемент *y*. Она создает новый узел списка *List* и делает *y* его левым потомком, а `nil` – правым потомком. Затем она использует указатель на конец списка, хранящийся в *x*, чтобы добавить новый узел *List* в конец списка. После чего возвращает *x*.

Все три функции можно реализовать так, что время работы будет составлять O(1). Совместно они строят заголовочный узел и список и выполняют операции над ними. Продукция *List* → *List'* создает в вычислении точку, в которой можно уничтожить заголовочный узел. Построенное АСД немного отличается от приведенного ранее; как показано во врезке, в нем правым потомком последнего узла *List* всегда является `nil`.



АСД, построенное каркасом

5.7.2. Более трудные проблемы вывода типов

Сильно типизированные языки со статической проверкой типов могут помочь программисту писать корректные программы, поскольку обнаруживают большие классы ошибок. Но те же средства, которые обнаруживают ошибки, могут улучшить способность компилятора генерировать эффективный код благодаря (1) устранению проверок во время выполнения или (2) выявлению ситуаций, в которых компилятор может специализировать код некоторых конструкций с целью исключить случаи, невозможные во время выполнения. Именно этими преимуществами отчасти объясняется растущая роль систем типов в языках программирования.

Однако в наших примерах делаются предположения, выполняющиеся не во всех языках программирования. Например, мы предполагали, что переменные и процедуры объявляются – программист записывает краткую и обязывающую спецификацию каждого имени. Изменение этих предположений может радикально изменить природу как самой проблемы проверки типов, так и стратегий, применяемых компилятором для реализации языка.

В некоторых языках программирования объявления либо опускаются вовсе, либо трактуются как факультативная информация. В программах на PYTHON и SCHEME объявлений переменных нет вообще. В SMALLTALK классы объявляются, но класс объекта определяется, только когда программа создает экземпляр этого класса. Языки, поддерживающие раздельную компиляцию – независимую компиляцию процедур с последующей компоновкой для создания исполняемой программы, – могут не требовать объявлений для независимо компилируемых процедур.

В отсутствие объявлений проверка типов осложняется, потому что компилятор должен полагаться на контекстуальные ключи, чтобы определить тип каждого имени. Например, если компилятор видит ссылку на элемент

массива a[i], то такое использование может наложить ограничения на тип i. Возможно, язык допускает только целые индексы или же только типы, которые можно преобразовать в целое.

Правила типизации формулируются в определении языка. Конкретные детали этих правил определяют, насколько трудно вывести тип каждой переменной. А это, в свою очередь, напрямую влияет на стратегии, доступные компилятору при реализации языка.

Совместимое с типом использование и постоянные типы функций

Рассмотрим не поддерживающий объявлений язык, который требует непротиворечивого использования переменных и функций. Компилятор может назначить общий тип каждому имени, а затем сузить этот тип, проанализировав, как имя используется в контексте. Например, из предложения a ← b × 3.14159 следует, что a и b – числа и что a должно иметь тип, допускающий хранение в нем десятичного числа. Если b встречается также в контекстах, где ожидается целое, например в ссылке на элемент массива c[b], то компилятор должен выбрать между десятичным числом (исходя из b × 3.14159) и целым (исходя из c[b]). При любом выборе в одном из случаев потребуется преобразование.

Если функция возвращает значение известного и постоянного типа – т. е. функция *fee* всегда возвращает один и тот же тип, – то компилятор может разрешить проблему вывода типа с помощью итеративного алгоритма поиска неподвижной точки, определенного над решеткой типов.

Совместимое с типом использование и неизвестные типы функций

Map может также обрабатывать функции с несколькими аргументами. Для этого она принимает несколько аргументов-списков и рассматривает их как списки аргументов, по порядку.

Если тип функции изменяется в зависимости от ее аргументов, то проблема вывода типа становится более сложной. Эта ситуация возникает, например, в языке SCHEME, где библиотечная функция *map* принимает в качестве аргументов функцию и список. Она возвращает результат применения аргумента-функции к каждому элементу списка. То есть если аргумент-функция преобразует значение типа α в значение типа β, то *map* принимает список значений типа α и возвращает список значений типа β. Такая сигнатура типа записывается в виде:

$$map: (\alpha \rightarrow \beta) \times \text{list of } \alpha \rightarrow \text{list of } \beta$$

Поскольку тип возвращаемого *map* значения зависит от типов ее аргументов (это свойство называется параметрическим полиморфизмом), правила

вывода должны включать уравнения над пространством типов. (Если возвращаемый тип известен и постоянен, то функции возвращают значения из пространства типов.) При таком добавлении простого итеративного подхода к выводу типов на основе поиска неподвижной точки становится недостаточно.

Классический подход к проверке типов в этих более сложных системах опирается на унификацию, хотя при продуманном дизайне системы типов и представлений типов возможны более простые или более эффективные методы.

Динамические изменения типа

Если тип переменной может изменяться во время выполнения, то для обнаружения места, где именно тип изменяется, и для вывода типов могут понадобиться другие стратегии. В принципе, компилятор может переименовывать переменные, так что каждому месту определения будет соответствовать уникальное имя. Затем он может вывести тип имени из контекста, предоставленного операцией, которая определяет имя.

Для успешного вывода типов такая система должна была бы обрабатывать места в коде, где разные определения обязаны сливаться в силу схождения различных путей управления, как обстоит дело с φ-функциями в форме с единственным статическим присваиванием (см. разделы 4.6.2 и 9.3). Если язык включает параметрический полиморфизм, то механизм вывода типов должен обрабатывать и его тоже.

Классический подход к языку с динамически изменяющимися типами – обращение к интерпретации. С этой проблемой сталкиваются LISP, SCHEME, SMALLTALK и APL. Стандартная практика реализации этих языков подразумевает интерпретацию операторов, аннотирование данных их типами и проверку типов во время выполнения.

В APL выражение `a × b` может означать перемножение целых чисел при первом выполнении и перемножение многомерных массивов чисел с плавающей точкой – в следующий раз. Эта особенность породила целый пласт исследований по исключению и перемещению проверок. Лучшие APL-системы сумели избежать многих проверок, кажущихся необходимыми в наивном интерпретаторе.

5.7.3. Относительные смещения и производительность кеша

У широкого применения кеш-памяти есть тонкие последствия для размещения переменных в памяти. Если два значения используются в коде рядом, то компилятору стоило бы позаботиться о том, чтобы они могли оказаться в кеше одновременно. Добиться этого можно двумя способами. В идеале оба значения располагаются в одном блоке кеша, так что всегда загружаются

в кеш вместе. Если поместить их в один блок кеша не удается, то компилятор мог бы постараться сделать так, чтобы обе переменные отображались в разные строки кеша. Для этого он может контролировать расстояние между их адресами.

Если мы говорим всего о двух переменных, то контроль расстояния между ними кажется не слишком сложным делом. Но если рассматривать все активные переменные разом, то задача их организации оптимальным для кеширования способом становится NP-полной. Большинство переменных взаимодействуют со многими другими переменными; это создает запутанную сеть взаимосвязей, распутать которую компилятору может оказаться не под силу. Если мы рассмотрим цикл, в котором используется несколько больших массивов, то проблема гарантии взаимного невмешательства оказывается еще труднее. Если компилятор способен обнаружить связь между различными ссылками на массив в цикле, то он может добавить промежутки между массивами, чтобы повысить вероятность того, что ссылки попадают в разные строки кеша и не мешают друг другу.

Введение в кеш-память

Один из методов, который архитекторы применяют, чтобы сгладить разрыв между быстродействием процессора и памяти, – использование *кеш-памяти*. Кеш – это небольшая быстрая память, расположенная между процессором и основной памятью. Кеш разбит на несколько *фреймов* одинакового размера. У каждого фрейма имеется *тег*, в котором хранится достаточно информации об адресе в основной памяти, чтобы идентифицировать содержимое фрейма.

Оборудование автоматически отображает адреса памяти на фреймы кеша. В простейшем случае кеша с прямым отображением адрес в кеше вычисляется как адрес в основной памяти по модулю размера кеша. Тем самым память разбивается на линейное множество блоков, размер каждого из которых равен размеру фрейма кеша. *Строкой* называется блок памяти, отображаемый на фрейм. В каждый момент времени каждый фрейм кеша содержит копию данных из какого-то блока. В поле тега находится адрес в памяти, откуда взяты эти данные.

При каждом чтении из памяти оборудование сначала проверяет, находится ли запрошенное слово во фрейме кеша. Если да, то запрошенные байты сразу возвращаются процессору. В противном случае (1) блок, находящийся в данный момент во фрейме, вытесняется, (2) запрошенный блок выбирается из памяти в кеш и (3) запрошенные байты возвращаются процессору.

В некоторых кешах применяется более сложное отображение. В наборно-ассоциативном кеше используется несколько фреймов на одну строку кеша, обычно два или четыре. В полностью ассоциативном кеше любой блок может быть загружен в любой фрейм. В обеих схемах применяется ассоциативный поиск по тегам, чтобы определить, находится ли блок в кеше. В ассоциативных кешах используется некоторая политика, определяющая, какой блок вытеснять. Наиболее распространенные политики – случайное вытеснение и вытеснение по давности использования.

На практике эффективное быстродействие памяти определяется, исходя из пропускной способности памяти, длины блока кеша, отношения быстродействия кеша к быстродействию памяти и отношения количества попаданий в кеш к количеству непопаданий. С точки зрения компилятора, первые три фактора фиксированы. Все

усилия компилятора по повышению производительности памяти направлены на увеличение частоты попаданий в кеш. В некоторых архитектурах имеются команды, позволяющие программе давать указания кешу насчет того, какие конкретно блоки следует загружать (*предвыборка*) и когда их следует вытеснять (*сброс*).

Как мы уже видели, отображение виртуального адресного пространства программы на физическое адресное пространство не обязано сохранять расстояние между переменными. Доводя эту мысль до логического завершения, читатель должен задаться вопросом, как компилятор может гарантировать что-нибудь по поводу относительных смещений, больших размера страницы виртуальной памяти. Процессорный кеш может использовать в тегах либо виртуальные, либо физические адреса. Виртуально адресуемый кеш сохраняет расстояние между значениями в виртуальном пространстве; компилятор может обеспечить отсутствие интерференции между большими объектами. В случае физически адресуемого кеша расстояние между двумя адресами в разных страницах определяется с помощью карты страницы (если только не выполняется условие размер кеша ⩽ размер страницы). Таким образом, решения компилятора по поводу размещения в памяти что-то значат только тогда, когда речь идет об одной странице. В такой ситуации компилятор должен постараться, чтобы объекты, к которым программа обращается вместе, находились на одной странице и, если возможно, даже в одной строке кеша.

5.8. Итоги и перспективы

Истинная работа компилятора – трансляция: отображение конструкций исходного языка на операции целевой машины. Фаза анализа строит начальную модель программы: промежуточное представление и набор дополнительных структур. В этой главе была рассмотрена синтаксически управляемая трансляция – механизм, позволяющий автору компилятора записывать действия, которые должны быть выполнены, когда фаза анализа распознает конкретные синтаксические конструкции. Автор компилятора связывает эти действия с продукциями грамматики: компилятор выполняет их, когда распознает продукцию.

Разработка формальных методов позволила автоматизировать многие аспекты построения сканеров и парсеров. При трансляции большинство компиляторов полагаются на ситуативные приемы синтаксически управляемой трансляции. И хотя исследователи разработали много формальных методов, например атрибутные грамматические системы, они не нашли широкого применения. Синтаксически управляемые методы по природе своей ситуативны; чтобы

Мы подозреваем, что атрибутные грамматические системы не смогли покорить аудиторию из-за отсутствия широкодоступной, хорошо реализованной и простой в использовании системы.

`yacc` и `bison` выиграли схватку не потому, что особенно элегантны, а потому, что распространялись в составе UNIX и работали.

эффективно применять их, необходим практический опыт. В этой главе мы поделились таким опытом.

Для выполнения трансляции компилятор должен построить базу знаний, более глубоких, чем синтаксические. Он должен использовать языковую систему типов для вывода типа каждого значения, вычисляемого программой, и воспользоваться этой информацией для обнаружения ошибок и автоматических преобразований типов. Наконец, компилятор должен вычислить размещение в памяти того кода, который он видит; это размещение должно быть согласовано и совместимо с результатами других компиляций связанного кода.

Замечания к этой главе

Материал, изложенный в этой главе, представляет собой свод знаний, почерпнутых из практик, которые были введены в обиход в конце 1950-х – начале 1960-х годов.

Идеи, лежащие в основе синтаксически управляемой трансляции, всегда были частью разработки реальных парсеров. Айронс, описавший ранний компилятор ALGOL-60, ясно указывает на необходимость действий парсера, отдельных от описания его синтаксиса [214]; он формулирует основные идеи синтаксически управляемой трансляции. Те же базовые идеи, несомненно, использовались в современных ему парсерах с операторным предшествованием.

Конкретная нотация, применяемая для описания синтаксически управляемых действий, была введена Джонсоном в системе `yacc` [216]. Эта нотация перешла во многие более поздние системы, включая генератор парсеров bison из проекта Gnu.

Системы типов являлись неотъемлемой частью языков программирования еще со времен первого компилятора FORTRAN. Самые первые системы типов отражали ресурсы физической машины, но вскоре в системах типов для таких языков, как ALGOL 68 и SIMULA-67, появились более глубокие уровни абстракции. Теория систем типов активно разрабатывалась на протяжении нескольких десятилетий, что породило серию языков, в которых были воплощены важные принципы. К ним относятся RUSSELL [49] (параметрический полиморфизм), CLU [256] (абстрактные типы данных), SMALLTALK [172] (подтипизация с помощью наследования) и ML [274] (тщательный и всесторонний подход к обращению с типами как с полноценными объектами). Карделли написал великолепный обзор систем типов [76]. В сообществе APL родилась серия классических статей, в которых рассматривались методы исключения проверок во время выполнения [1, 38, 273, 361].

Большая часть материала по размещению в памяти разрабатывалась как часть спецификаций языка программирования. Порядок хранения массивов по столбцам появился в ранних версиях FORTRAN [27, 28] и был закреплен в стандарте FORTRAN 66. Порядок хранения по строкам используется с 1950-х годов.

Упражнения

К разделу 5.3

1. Рассмотрите проблему добавления синтаксически управляемых действий в генератор LL(1)-парсеров. Как бы вы изменили заготовку LL(1)-парсера, чтобы включить в нее определенные пользователем действия для продукций?
2. Рассмотрите следующую простую грамматику, описывающую язык строк над словами `one`, `two` и `three`.

```
Goal  → List
List  → List Word
      | Word
Word  → one
      | two
      | three
```

   У каждого слова есть вес: `one` = 1, `two` = 2, `three` = 3.
   Напишите набор правил синтаксически управляемой трансляции, который вычисляет две характеристики входного предложения: сумму весов всех слов и сколько раз каждое слово встречается (его частоту).
   В момент свертки символа *Goal* парсер должен напечатать сумму и частоты.

К разделу 5.4

3. Правило определения перед использованием требует, чтобы каждая переменная, используемая в процедуре, была объявлена раньше, чем она встречается в тексте. Напишите набросок простой синтаксически управляемой схемы трансляции, проверяющей, что процедура удовлетворяет этому правилу. (**Указание:** используйте отображение.)
   Становится ли эта задача проще, если язык требует, чтобы все предложения объявлений предшествовали исполняемым предложениям?
4. Для программы, показанной на рис. 5.9, составьте полный набор таблиц символов и покажите пути поиска, которые компилятор должен построить для процедур `Main`, `Fee`, `Fie`, `Foe` и `Fum`.
5. Встретив определение структуры, компилятор должен рассматривать все, что внутри, как новую область видимости. (Имена элементов структуры маскируют такие же имена в объемлющей области видимости.)
   a. Если компилятор представляет эти области видимости с помощью независимых таблиц, то как он мог бы связать их в путь поиска?
   b. Может ли компилятор использовать квалифицированные имена (например, `x.next`) для представления этих имен?
6. Компилятор должен хранить в ПП программы информацию, позволяющую находить имена в таблице символов. Автору компилятора доступны следующие возможности: указатели на исходные строки литер, указатели на записи таблицы символов и пара, содержащая указатель на таблицу и смещение от ее начала.

Каковы плюсы и минусы каждого из этих представлений имени? Как бы вы стали представлять имена в компиляторе, который проектируете с нуля?

```
print "enter a number:";
n = input();
if n < 0:
     v = "string";
else:
     v = 23;
print v;
```

(a) Оригинальная версия программы на PYTHON

```
print "enter a number:";
n = input();
if n < 0:
     v = "string";
     print v;
else:
     v = 23;
     print v;
```

(a) Модифицированная версия программы на PYTHON

Рис. 5.19 Код к упражнению 7

К разделу 5.5

7. На рис. 5.19 показаны две версии программы на PYTHON. Язык PYTHON динамически типизированный, поэтому тип v определяется во время выполнения на основе присваиваний v. Вы разрабатываете инструмент, который пытается выполнить статическую проверку программ на PYTHON.
   a. Какой тип имеет v в предложении `print` на рисунке (a)?
   b. Какой тип имеет v в каждом предложении `print` на рисунке (b)?
   c. Будут ли различаться сигнатуры типа, выведенные вашим инструментом для функции `print` в обеих версиях кода?
   d. Какие проблемы возникают при попытке трансляции каждой из этих программ на C?

8. Предположим, что в вашем компиляторе используется схема, объединяющая показанное на рис. 5.5 и 5.13. Дополните эту схему, так чтобы она правильно выполняла преобразование между целыми и числами с плавающей точкой на основе типов каждого подвыражения. В качестве конкретного ПП возьмите язык ILOC, описанный в приложении A.

К разделу 5.6

9. Компилятор должен сопоставить каждой переменной смещение от начала области данных. Предположим, что алгоритм получает на входе список переменных, их длин и ограничений выравнивания, например:

   ⟨a, 4, 4⟩, ⟨b, 3, 1⟩, ⟨c, 8, 8⟩, ⟨d, 4, 4⟩, ⟨e, 1, 4⟩, ⟨f, 8, 16⟩, ⟨g, 1, 1⟩.

   На выходе алгоритм должен породить список переменных и их смещений от начала области данных. Цель алгоритма – минимизировать непроизводительные потери места в памяти.
   a. Напишите краткий алгоритм, который выполняет размещение в памяти, минимизируя непроизводительные потери места.
   b. Примените этот алгоритм к списку выше и двум другим спискам, которые вы должны спроектировать для демонстрации проблем, возникающих при размещении в памяти.
   c. Какова сложность вашего алгоритма?

10. Пусть дан массив целых чисел размерностью `A[0:99,0:89,0:109]`. Сколько слов памяти необходимо для представления `A` по строкам и с помощью вектора указателей? Можете предполагать, что размер целых и указателей одинаков.

11. Для каждой из следующих категорий переменных укажите, где компилятор должен отвести им память. Возможные ответы: регистр, запись активации, область статических данных (с различной видимостью) и куча.
- **a.** Скалярная неоднозначная локальная переменная.
- **b.** Скалярная однозначная локальная переменная.
- **c.** Глобальная переменная.
- **d.** Динамически выделяемая глобальная переменная.
- **e.** Формальный параметр процедуры.
- **f.** Сгенерированное компилятором временное значение.
- **g.** Локальная переменная, размер которой определяется во время выполнения.

К разделу 5.7

12. Для леворекурсивной и праворекурсивной грамматик сложения, приведенных в разделе 5.7.1, покажите все состояния стека, возникающие при разборе выражения `a + b + c + d + e + f`. Вычислите максимальную и среднюю глубину стека для каждой грамматики. Каково асимптотическое значение средней глубины стека для каждой грамматики?

Глава 6

Реализация процедур

Аннотация

Процедуры играют очень важную роль в разработке программных систем. Они предоставляют абстракции для управления потоком и именования. Они являются одним из основных средств сокрытия информации. Они служат строительным блоком, на котором системы возводят интерфейсы. Они являются одной из главных форм абстрагирования в Algol-подобных языках, а объектно-ориентированные языки опираются на процедуры при реализации методов.

В этой главе подробно рассматривается реализация процедур и их вызовов, с точки зрения автора компилятора. Особое внимание уделяется сходствам и различиям реализации в Algol-подобных и объектно-ориентированных языках. В разделе «Дополнительные темы» представлено краткое введение в алгоритмы, используемые для управления кучей времени выполнения.

Ключевые слова

Вызовы процедур, передача параметров, соглашения о связях

6.1. Введение

Процедура – одна из центральных абстракций в большинстве языков программирования. Процедуры создают управляемую среду выполнения, у каждой процедуры есть собственная закрытая память. Процедуры помогают определять интерфейсы между компонентами системы; взаимодействия компонентов обычно структурируются с помощью вызовов процедур. Наконец, процедуры – базовая единица работы большинства компиляторов. Типичный компилятор обрабатывает набор процедур и генерирует для них код. Код, порожденный в результате такой раздельной компиляции, должен компоноваться с кодом, откомпилированным в другое время, в исполняемую программу.

Процедуры играют важную роль в раздельной компиляции, благодаря которой разработчики могут создавать большие программные системы. Если

бы компилятору был нужен весь текст программы для каждой компиляции, то с большой системой было бы невозможно работать. Только представьте, что требуется перекомпилировать несколько миллионов строк кода после внесения каждого изменения. Поэтому процедуры столь же важны для проектирования и конструирования систем, сколь для проектирования языков и реализации компиляторов. В этой главе мы рассмотрим, как компиляторы реализуют процедуры и методы.

Концептуальная дорожная карта

Для трансляции программы с исходного языка в исполняемый код компилятор должен отобразить все конструкции исходного языка в операции и структуры данных целевого процессора. Компилятору нужны стратегии для всех абстракций, поддерживаемых исходным языком. В состав стратегии входят алгоритмы и структуры данных, включаемые в исполняемый код.

Для реализации этих стратегий автор компилятора должен расставить по местам вычисления и структуры данных, применяемые на этапе компиляции, механизмы передачи информации от этапа компиляции этапу выполнения, а также код в исполняемом файле, который реализует часть стратегии, относящуюся к этапу выполнения. В совокупности все эти элементы порождают желаемое поведение во время выполнения программы.

В этой главе описываются приемы, используемые авторами компиляторов для реализации процедур и их вызовов. Точнее, рассматривается реализация управления, именования и интерфейса вызовов. Эти абстракции инкапсулируют многие средства, благодаря которым с языками программирования можно работать и использовать их для построения крупномасштабных систем.

Несколько слов о времени

В этой главе рассматриваются проблемы, возникающие на этапах компиляции и выполнения. Компилятор планирует желаемое поведение на этапе выполнения. Он генерирует код для реализации этого поведения – обычно стандартные механизмы вызова и возврата. Чтобы понять, как работают вызовы, читатель должен внимательно отнестись к взаимодействию между планированием на этапе компиляции и поведением на этапе выполнения.

Например, компилятор должен спланировать потребности каждой процедуры в памяти. Анализируя код процедуры, компилятор строит карту памяти, включающую тип и размер каждого вычисляемого в процедуре значения. Каждому значению сопоставляется место в памяти на этапе выполнения. С помощью этой карты компилятор генерирует код, который на этапе выполнения будет выделять, инициализировать и освобождать память

Вызывающая сторона
Процедура, которая инициирует вызов другой процедуры.

Вызываемая сторона
Процедура, вызываемая из другой процедуры.

для этих значений. Код, реализующий эти решения, распределен между вызывающей и вызываемой сторонами.

Аналогично компилятор определяет, какие действия следует предпринять в каждом месте вызова, чтобы сохранить и защитить окружение вызывающей процедуры; затем он генерирует код, проводящий эти решения в жизнь на этапе выполнения. К ним относятся сохранение и восстановление регистров, вычисления и связывание параметров, а также обслуживание структур данных, обеспечивающих правильный и надлежащий доступ к значениям в объемлющих областях видимости. И снова компилятор анализирует, планирует и генерирует код. А во время выполнения этот код обеспечивает желаемое поведение.

Обзор

Процедуры – это строительные блоки программ. Они создают известную и контролируемую среду выполнения. Процедура выполняется, когда ее вызывает другая процедура. В момент вызова создается и инициализируется память, локальная для процедуры. Она защищает окружение вызывающей стороны, устанавливает окружение вызываемой стороны и создает связи между этими окружениями, характер которых определяется вызовом и языком. Вызываемая сторона может вернуть вызывающей значение, и в таком случае процедура называется *функцией*. Интерфейс между процедурами позволяет программисту разрабатывать и тестировать части программы изолированно друг от друга; разделение на процедуры обеспечивает определенную изоляцию от проблем в других процедурах.

Процедуры мостят путь, следуя которым, программисты разрабатывают программное обеспечение, а компиляторы транслируют программы. Три важнейшие абстракции, предоставляемые процедурами, позволяют создавать нетривиальные программы.

1. *Вызов*. Вызовы служат средством упорядоченной передачи управления между процедурами. Механизм вызова определяет стандартный способ вызова процедуры и передачи набора аргументов, или параметров, из пространства имен вызывающей стороны в пространство имен вызываемой. После завершения работы вызываемой стороны управление возвращается вызывающей стороне в точке, находящейся сразу после вызова. Большинство языков позволяют процедуре возвращать вызывающей стороне одно или несколько значений.

Соглашение о связях
соглашение между компилятором и операционной системой, определяющее действия, предпринимаемые для вызова процедуры или функции.

Языки и их реализации определяют стандартный набор действий, необходимых для вызова процедуры, который иногда называют *последовательностью вызова*. Стандартизация последовательностей вызова в форме *соглашения о связях* позволяет программисту вызывать код, написанный и откомпилированный другими людьми и другими компиляторами в другое время.

Фактический параметр
Значение или переменная, переданная в качестве параметра в точке вызова, называется фактическим параметром вызова.

Формальный параметр
Имя, объявленное в качестве параметра процедуры *p*, называется *формальным параметром p*.

2. *Пространство имен.* В большинстве языков вызов процедуры создает новое защищенное пространство имен для вызываемой стороны (см. раздел 5.4). Программист может объявлять новые имена, например переменные и метки, внутри процедуры, не заботясь об объемлющем контексте. Эти локальные объявления маскируют ранее определенные одноименные элементы. Программист может использовать формальные параметры процедуры для передачи значений и переменных из пространства имен вызывающей стороны в пространство имен вызываемой стороны. Поскольку процедура имеет известное и отдельное пространство имен, она может работать правильно и одинаково, будучи вызвана из разных контекстов.

   Выполнение вызова создает пространство имен вызываемой стороны. При вызове необходимо выделить память для объектов, объявленных вызываемой стороной. Это выделение должно быть автоматическим и эффективным.
3. *Внешний интерфейс.* Процедуры обеспечивают интерфейс между частями большой программной системы. Соглашение о связях определяет правила, которые отображают имена на значения и адреса в памяти, сохраняют среду выполнения вызывающей стороны и передают управление между вызывающей и вызываемой сторонами.

   Соглашение о связях описывает стандартный способ вызова кода, написанного другими людьми и на других языках. Единообразные последовательности вызова позволяют строить большие программные системы с использованием библиотек и системных вызовов. Не будь соглашения о связях, программисту и компилятору пришлось бы разбираться в коде вызываемой стороны при каждом вызове процедуры.

Таким образом, процедура во многих отношениях является фундаментальной абстракцией, лежащей в основе Algol-подобных языков. Это тщательно продуманный интерфейс, созданный совместно компилятором, оборудованием и операционной системой. Процедуры создают именованные переменные и отображают их на виртуальные адреса, а операционная система и оборудование отображают виртуальные адреса в физические. Процедуры устанавливают правила видимости и адресуемости имен, а оборудование загружает и сохраняет значения по адресам, вычисленным откомпилированным кодом. Процедуры дают возможность разбивать большие программные системы на компоненты; компоновщики и загрузчики связывают эти компоненты в исполняемые файлы; оборудование исполняет код, увеличивая счетчик команд и выполняя условные и безусловные переходы.

Реализация соглашения о связях требует координации между компилятором и объемлющей системой, а также между отдельными компиляциями.

Компилятор должен позаботиться о том, чтобы сгенерированный код использовал интерфейсы, предоставляемые операционной системой для ввода и вывода, для управления выделением и освобождением памяти и для взаимодействия с другими процессами. Компилятор должен организовать память для каждой процедуры и включить эти знания в сгенерированный код. Поскольку разные компоненты программы могут компилироваться в разное время, разными компиляторами и отдельно друг от друга, все эти взаимодействия должны быть стандартизированы, а соглашения должны применяться единообразно.

Дело осложняет тот факт, что пользователь хочет, чтобы оптимизатор специализировал сгенерированный код с учетом контекста времени выполнения, сделав его максимально быстрым (см. главы 8–10).

В этой главе нас будут интересовать процедура как абстракция и механизмы, используемые компилятором для организации этой абстракции управления, ее пространства имен и интерфейса с внешним миром.

6.2. Предварительные сведения

В Algol-подобных языках для процедур определена простая дисциплина вызова и возврата. Вызов передает управление из точки вызова на вызывающей стороне в начало вызываемой стороны; когда вызываемая сторона завершает работу, управление возвращается в точку, расположенную сразу после вызова. Если вызываемая сторона, в свою очередь, вызывает другие процедуры, то они следуют тому же протоколу вызова и возврата. На рис. 6.1 показан граф вызовов и *история выполнения* для PASCAL-программы на рис. 5.9.

Активация
Вызов процедуры *активирует* ее, поэтому мы называем экземпляр ее выполнения *активацией*.

Граф вызовов на рисунке (b) показывает все имеющиеся в программе вызовы. При выполнении `Main` возможны два разных вызова `Fee`: один из `Foe`, другой из `Fum`. История выполнения показывает, что оба вызова действительно имели место. Каждый вызов создает отдельный экземпляр, или *активацию* `Fee`. К моменту вызова `Fum` первый экземпляр `Fee` уже не активен. Он был создан вызовом из `Foe` (событие 3 в истории выполнения) и уничтожен после возврата управления в `Foe` (событие 4). Когда `Fum` вызывает `Fee` (событие 6), создается новая активация `Fee`. Возврат из `Fee` в `Fum` завершает эту активацию и делает ее состояние недоступным.

Отклонение
Если вычисление не завершается нормально, говорят, что оно *отклоняется*.

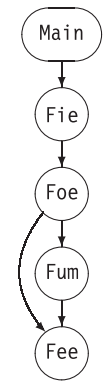
Адрес возврата
Когда p вызывает q, адрес в p, с которого вычисление должно продолжиться после возврата из q, называется *адресом возврата*.

Когда программа выполняет присваивание `x:=1` в первом вызове `Fee`, активны `Fee`, `Foe`, `Fie` и `Main`. Все они лежат на пути из `Main` в `Fee` в графе вызовов. Аналогично в момент второго вызова `Fee` активны `Fee`, `Fum`, `Foe`, `Fie` и `Main`; они лежат на другом пути из `Main` в `Fee`. Механизм вызова и воз-

врата гарантирует, что в любой точке выполнения активации процедуры образуют некоторый путь в графе вызовов, начинающийся в корне.

```
program Main(input, output);
   var x,y,z: integer;
   procedure Fee;
      var x: integer;
      begin { Fee }
         x := 1;
         y := x * 2 + 1
      end;
   procedure Fie;
      var y: real;
      procedure Foe;
         var z: real;
            procedure Fum;
               var y: real;
               begin { Fum }
                  x := 1.25 * z;
                  Fee;
                  writeln('x = ',x)
               end;
         begin { Foe }
            z := 1;
            Fee;
            Fum
         end;
      begin { Fie }
         Foe;
         writeln('x = ',x)
      end;
   begin { Main }
      x := 0;
      Fie
   end.
```

(a) Пример PASCAL-программы



(b) Граф вызовов

1. Main calls Fie
2. Fie calls Foe
3. Foe calls Fee
4. Fee returns to Foe
5. Foe calls Fum
6. Fum calls Fee
7. Fee returns to Fum
8. Fum returns to Foe
9. Foe returns to Fie
10. Fie returns to Main

(c) История выполнения

Рис. 6.1 Нерекурсивная PASCAL-программа и история ее выполнения

Код вызовов и возвратов, генерируемый компилятором, должен сохранять достаточно информации для корректного продолжения работы после возврата. Когда `Foe` вызывает `Fum`, код должен запомнить состояние `Foe`, включая адрес в `Foe`, куда `Fum` должна вернуть управление, – ее *адрес возврата*. Если `Fum` не отклоняется, то управление должно быть возвращено в `Foe`, и выполнение `Foe` должно быть продолжено. Механизм вызовов сохраняет достаточный объем состояния, чтобы все так и было.

Поведение вызова и возврата в Algol-подобных языках можно смоделировать с помощью стека, работающего по принципу «последним пришел, первым ушел». Поэтому реализации среды выполнения во многих языках программирования используют стек для хранения состояния всех активных

процедур. Когда `Fie` вызывает `Foe`, она помещает в стек новое состояние для `Foe`; выполнение `Foe` начинается с этого состояния. Когда `Foe` возвращается, она выталкивает это состояние из стека, открывая для `Fie` то состояние, в котором она находилась в момент вызова.

Механизм стека поддерживает рекурсию. По сути дела, он разворачивает циклический путь в графе вызовов и создает отдельную активацию и состояние для каждого вызова процедуры. При условии что рекурсия завершается, этот путь будет конечным, и стек корректно отражает поведение программы.

Чтобы добавить обсуждению конкретики, рассмотрим следующее вычисление факториала, написанное на языке SCHEME:

```
(define (fact k)
    (cond
        [(<= k 1) 1]
        [else (* (fact (sub1 k)) k)]
    ))
```

Вызов `(fact 5)` генерирует последовательность рекурсивных вызовов: `(fact 5)` вызывает `(fact 4)` вызывает `(fact 3)` вызывает `(fact 2)` вызывает `(fact 1)`. В этой точке предложение `cond` выполняет ветвь для `(<= k 1)`, завершая рекурсию. Теперь рекурсия раскручивается в обратном порядке. Вызов `(fact 1)` возвращает значение 1 `(fact 2)`. Он, в свою очередь, возвращает значение 2 `(fact 3)`, который возвращает 6 `(fact 4)`. Наконец, `(fact 4)` возвращает 24 `(fact 5)`, где 24 умножается на 5 и возвращается результат 120. Рекурсивная программа демонстрирует поведение вида «последним пришел, первым ушел», так что стек корректно отслеживает все адреса возврата.

Вызовы процедур в объектно-ориентированных языках

С точки зрения вызовов процедур, объектно-ориентированные языки (ООЯ) похожи на Algol-подобные. Основное различие между вызовами процедур в тех и других – механизм именования вызываемой стороны и ее нахождения во время выполнения.

В ООЯ вызов производится от имени некоторого объекта. Компилятор знает имена получателя и метода. Отсюда он должен определить класс получателя и сопоставить имени метода правильную реализацию. Если компилятор может разрешить связь между именем и реализацией на этапе компиляции, то генерируемый код будет похож на стандартный вызов в Algol-подобном языке. Но если эту связь невозможно разрешить до этапа выполнения, то компилятор должен сгенерировать код, который будет использовать для разрешения структуры данных, доступные во время выполнения.

Для поддержки разрешения имени на этапе выполнения компилятор должен сгенерировать код, который конструирует структуры данных, необходимые для моделирования иерархии наследования. Простой пример показан на рис. 5.17.

Более сложный межпроцедурный поток выполнения

Замыкание
процедура и контекст выполнения, определяющий ее свободные переменные.

В соответствии с идеями лямбда-исчисления некоторые языки программирования позволяют программе инкапсулировать процедуру и контекст ее выполнения в объект, называемый *замыканием*. При вызове замыкания процедура выполняется в инкапсулированном контексте. Простого стека недостаточно для реализации этой функциональности. Вместо этого управляющая информация должна быть сохранена в более общей структуре, способной представить процедуру и ее контекст. Аналогичные проблемы возникают, если язык допускает ссылки на локальные переменные, живущие дольше, чем активация процедуры.

ИТОГИ РАЗДЕЛА

В Algol-подобных языках процедуры активируются вызовом и завершаются возвратом. Компилятор должен позаботиться о том, чтобы при вызове было сохранено состояние вызывающей стороны, а при возврате оно было найдено и восстановлено. Использование стека для хранения состояний активации процедур отражает поведение «последним пришел, первым ушел», характерное для механизма вызова и возврата.

ООЯ и языки, поддерживающие замыкание, строятся на той же базовой инфраструктуре, но порождают более сложный поток управления. В ООЯ применяются схемы именования, ориентированные на данные, для отображения имен на реализации методов. Замыкания разрывают связь между временем жизни области видимости и создавшим ее вызовом; в этих схемах требуется более сложная реализация.

ВОПРОСЫ

1. Многие языки программирования включают прямую передачу управления – предложение `goto`. Сопоставьте вызов процедуры и `goto`.
2. Во многих языках используются библиотеки заранее откомпилированного кода. Что необходимо компилятору от библиотечного кода, чтобы гарантировать правильное поведение? Как ваш любимый язык предоставляет это компилятору?

6.3. Поддержка именования на этапе выполнения

Общая цель состоит в том, чтобы преобразовать имя времени компиляции в виртуальный адрес времени выполнения.

Чтобы поддержать пространства имен, существующие в программах на исходном языке, компилятор должен уметь отображать заданную ссылку в ее виртуальный адрес времени выполнения. Этот механизм состоит из двух частей: на этапе компиляции имя преобразуется в некоторую сущность времени выполнения, а на этапе выполнения предоставляется метод для нахождения этой сущности и организации доступа к ней.

Описание этих механизмов разнесено по нескольким главам. В разделе 5.4 описываются механизмы времени выполнения, с помощью которых имя исходного языка преобразуется в сущность времени выполнения. В разделе 5.6 обсуждается, как компилятор определяет, где будет находиться каждая сущность в памяти компьютера. В настоящем разделе рассматриваются структуры времени выполнения, которые компилятор должен включить в код, чтобы обеспечить адресуемость и сохранить значения. Наконец, в разделе 7.3 описывается, как компилятор использует эти механизмы, чтобы сгенерировать код, умеющий вычислять виртуальный адрес времени выполнения.

Поддержка разрешения и трансляция имен на этапе выполнения сосредоточены в двух основных местах. Структуры данных создаются, заполняются и уничтожаются как часть реализации вызовов процедур. В разделе 6.3.1 описываются механизмы, которые обычно применяются для поддержки вложенных лексических областей видимости в Algol-подобных языках. В разделе 6.3.2 обсуждаются дополнительные структуры времени выполнения, необходимые для поддержки объектно-ориентированных языков (ООЯ). Все остальное происходит, когда компилятор генерирует код отдельных ссылок, следуя планам, изложенным в разделе 7.3.

6.3.1. Поддержка времени выполнения в Algol-подобных языках

Запись активации
область памяти, предназначенная для хранения управляющей информации и данных, связанных с одним экземпляром одной процедуры.

Чтобы реализовать тесно связанные абстракции вызовов процедур и пространств имен с лексической областью видимости, компилятор и среда выполнения должны подготовить ряд структур данных. Главная структура, участвующая в управлении и именовании, – это запись активации (ЗА), блок памяти, ассоциированный с одной активацией процедуры. Каждый вызов процедуры порождает новую ЗА.

- Компилятор должен генерировать такой код, чтобы вызывающая сторона могла сохранить адрес возврата для вызываемой. Адрес возврата помещается в ЗА вызываемой стороны.
- Компилятор должен отобразить фактические параметры в месте вызова в формальные параметры, по которым они доступны вызываемой стороне. Эта информация о параметрах помещается в ЗА вызываемой стороны.
- Компилятор должен отвести место для локальных переменных, объявленных вызываемой стороной. Для значений, время жизни которых совпадает с вызовом процедуры, это место отводится в ЗА вызываемой стороны.
- Компилятор должен предоставить вызываемой стороне другую информацию, необходимую для того, чтобы она могла связаться с вызывающим контекстом и взаимодействовать с другими процедурами. Эти данные помещаются в ЗА вызываемой стороны.

Каждый вызов создает новую ЗА. Если активно несколько экземпляров процедуры, то у каждой имеется своя ЗА. Таким образом, рекурсия порождает несколько ЗА, в каждой из которых хранится локальное состояние отдельной активации процедуры.

На рис. 6.2 показано, как могло бы быть размещено содержимое ЗА. *Указатель записи активации* (УЗА) указывает на фиксированное место внутри ЗА. Различные поля ЗА имеют положительные и отрицательные смещения относительно УЗА. ЗА, показанные на рис. 6.2, имеют следующие поля.

Указатель записи активации
Компилятор помещает указатель на ЗА, называемый *указателем записи активации*, в специальный регистр.

Мы будем обозначать этот регистр r_{arp}.

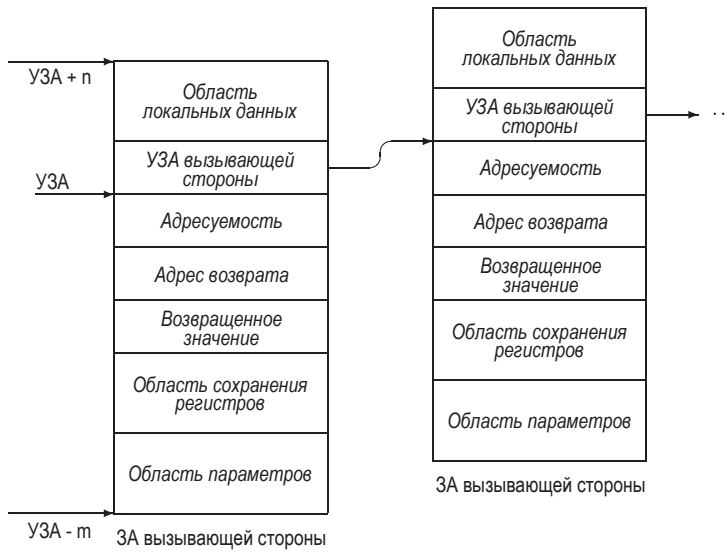


Рис. 6.2 Типичные записи активации

- В области параметров хранятся фактические параметры в месте вызова в том порядке, в котором они были перечислены при вызове.
- Под область сохранения регистров отведено достаточно места для хранения значений регистров, которые должны быть сохранены при вызовах процедур.
- Поле возвращенного значения служит для передачи данных от вызываемой стороны вызывающей, если в этом есть необходимость.
- В поле адреса возврата хранится адрес, на который должно быть передано управление после завершения вызываемой стороны.
- В поле адресуемости хранится информация, позволяющая вызываемой стороне обращаться к локальным переменным в объемлющих лексических областях видимости.

Объемлющая лексическая область видимости может быть процедурой, отличной от вызывающей стороны.

- В поле УЗА на вызываемой стороне хранится УЗА вызывающей стороны, чтобы вызываемая сторона могла восстановить окружение вызывающей стороны при возврате.
- В области локальных данных хранятся переменные, объявленные в локальной области видимости вызываемой стороны, которые та не может разместить в регистрах, а также другие значения, которые компилятор должен сохранить.

Ради эффективности часть информации, показанной в ЗА на рис. 6.2, может храниться в специальных регистрах.

Локальная память

В ЗА процедуры *q* хранятся как локальные данные, так и управляющая информация для этого экземпляра *q*. При каждом вызове *q* порождается новая ЗА. Доступ к полям ЗА производится через УЗА. Таким образом, УЗА играет роль описателя контекста и состояния активации. Во время выполнения код обращается к ЗА настолько часто, что большинство компиляторов отводят физический регистр для хранения УЗА активной процедуры. В ILOC мы называем этот специальный регистр $_{rarp}$.

УЗА указывает на вполне определенное место ЗА. Средняя часть ЗА имеет статическую структуру – длины всех полей фиксированы и известны. Поэтому откомпилированный код может обращаться к этим полям по их смещениям от УЗА. Оба конца ЗА зарезервированы для областей памяти, размеры которых зависят от деталей конкретной процедуры; с одной стороны хранятся параметры, а с другой – локальные данные.

Резервирование места для локальных данных

Распределяя память, компилятор отводит в области локальных данных место для автоматических локальных переменных, которые нельзя разместить в регистрах (см. раздел 5.6.3). Назначая память таким переменным, он должен запомнить в таблице символов текущий лексический уровень и смещение значения от УЗА. Эта пара играет решающую роль при доступе к значению на этапе выполнения (см. раздел 7.3.1).

Входная программа может содержать одну или несколько переменных, размер которых неизвестен на этапе компиляции. Например, код может вычислять размер массива или читать его размер из внешнего источника. В таких случаях компилятор может отвести в области локальных данных место для указателя либо на фактические данные, либо на вектор предварительной информации о фактических данных (см. раздел 7.3.2).

Затем компилятор может выделить место для хранения данных где-то еще. Он мог бы поместить данные в кучу; если ЗА находятся в стеке, то данные можно было бы поместить на вершину стека. В любом случае компилятор может сохранить указатель на выделенную им память в ЗА или в подходящем элементе вектора предварительной информации. Тогда статическая координата будет вести на позицию в ЗА, и код для доступа к массиву будет начинать работу с этого указателя.

Инициализация переменных

Большинство ассемблеров предоставляют псевдооперацию для выделения и инициализации статической памяти.

Если исходный язык позволяет программисту задавать начальные значения переменных, то компилятор должен позаботиться о том, чтобы инициализация имела место. Для статической переменной данные можно вставить непосредственно в нужные позиции области статических данных.

Переменные, хранящиеся в ЗА, должны быть инициализированы во время выполнения. Поскольку процедуру можно вызывать несколько раз, компилятор должен генерировать операции, которые производят инициализацию при каждом вызове. По существу, инициализация сводится к присваиваниям, которые выполняются до первого предложения процедуры при каждом вызове.

Место для хранения значений регистров

Когда p вызывает q, кто-то из них должен сохранить значения регистров, которые будут необходимы p после вызова. Возможно, придется сохранить значения всех регистров, но иногда может быть достаточно и подмножества. После возврата в p сохраненные значения должны быть восстановлены. Время жизни сохраненных регистров совпадает со временем жизни вызываемой стороны. Таким образом, компилятор может хранить их как в ЗА вызываемой стороны, так и в ЗА вызывающей стороны.

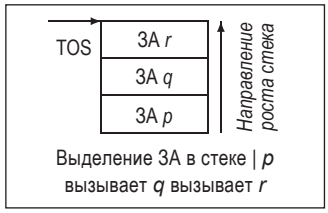
Если вызывающая сторона сохраняет регистр до передачи управления вызываемой стороны, то этот регистр будет храниться в ее собственной ЗА. И наоборот, если вызываемая сторона сохраняет регистр, прежде чем начать работу, то значение будет храниться в ее ЗА. Таким образом, в ЗА должно хватать места для хранения полного набора регистров, сохраняемых вызывающей или вызываемой стороной. Поскольку вызывающая сторона сохраняет регистры перед вызовом и восстанавливает после вызова, она может повторно использовать одно и то же место для хранения регистров при каждом вызове.

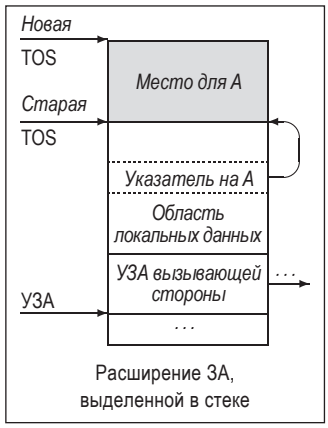
Выделение памяти для записи активации

Когда процедура p вызывает процедуру q, код вызова должен выделить и инициализировать память для записи активации q. Поскольку процедуры вызываются часто, автор компилятора должен постараться свести стоимость этого выделения к минимуму. В общем случае у автора компилятора есть три способа выделить память для ЗА: в стеке, в куче и статически.

Выделение записей активации в стеке

Во многих случаях содержимое ЗА представляет интерес только на время одной активации процедуры. Если локальные переменные не могут жить дольше создавшей их процедуры, а активация процедуры не может жить доль-





ше вызвавшей ее стороны, то вызовы и возвраты сбалансированы и подчиняются дисциплине «последним пришел, первым ушел» (LIFO). Вызов q из p рано или поздно возвращается, а любой возврат, случившийся между вызовом q из p и возвратом в p из q, должен быть результатом вызова (прямого или косвенного), выполненного внутри q.

При таких ограничениях ЗА также обслуживаются в порядке LIFO, и для выделения можно использовать простой стек. В PASCAL, C и JAVA ЗА обычно реализуются в стеке, как показано на рисунке слева.

Выделение ЗА в стеке дает несколько преимуществ. Выделение и освобождение памяти обходится дешево – код просто выполняет арифметические действия с указателем на вершину стека (TOS). Чтобы расширить ЗА на вершине стека, код должен проверить стек на переполнение, а затем изменить указатель на TOS. Эта схема позволяет процедуре расширять собственную ЗА для размещения локальных сущностей с динамически изменяющимся размером. Эта ситуация изображена во врезке. Здесь компилятор оставил место в области локальных данных для указателя на A, а затем расширил ЗА, чтобы выделить место для A.

В случае размещения в стеке задачу создания ЗА можно распределить между вызывающей и вызываемой сторонами. Вызывающая сторона создает те части ЗА, размер которых ей известен – всё от области параметров до УЗА вызывающей стороны. Затем вызываемая сторона расширяет собственную ЗА, отводя место для области локальных данных.

Размещение записей активации в куче

Если язык программирования позволяет активации процедуры существовать дольше вызывающей стороны, то схема применения стека с целью выделения памяти для ЗА не работает. Аналогично, если процедура может вернуть объект, например замыкание, включающий, явно или неявно, ссылки на ее локальные переменные, то выделять место в стеке нельзя, потому что останутся висячие указатели. В таких ситуациях компилятор может размещать ЗА в куче (см. раздел 6.6). В реализациях SCHEME и ML обычно используются ЗА в куче.

Хороший распределитель памяти может свести стоимость выделения в куче к минимуму. Большинство реализаций ЗА в куче не позволяют объекту расширяться после того, как память выделена. Этот факт осложняет схему размещения ЗА, т. к. из него следует одно из двух.

1. Вызывающая сторона должна знать размер области локальных данных вызываемой стороны. Компилятор может вычислить этот размер и со-

хранить его в глобальном пуле констант. При каждом вызове вызывающая сторона извлекает это значение и включает его в вычисление размера области локальных данных.

2. Вызываемая сторона должна выделять память для области локальных данных отдельно. Этот вариант, хотя и проще, может потребовать отдельного регистра для хранения указателя на область локальных данных.

В случае ЗА, размещаемых в куче, для объектов переменного размера можно выделять отдельную память в куче. Если эта память нуждается в явном освобождении, то код возврата из процедуры должен освободить ЗА и все ее расширения переменного размера.

ЗА, размещаемые в куче, хорошо работают для замыканий. Замыкание хранит указатель на запись создавшей ее активации. Цепочка полей УЗА вызывающей стороны будет удерживать остальное состояние доступным до тех пор, пока существует замыкание. Когда замыкание уничтожается, все части состояния, остававшиеся живыми только из-за замыкания, тоже умирают; схема неявного освобождения позаботится об их повторном замыкании.

Статическое выделение памяти для записей активации

Листовая процедура
процедура, не содержащая ни одного вызова.

Если процедура *q* не вызывает никаких других процедур, то для *q* не может быть нескольких активных вызовов. В этом случае мы называем *q листовой процедурой*, потому что она завершает путь в графе возможных вызовов процедур. Компилятор может статически выделить память для ЗА листовых процедур, чтобы избежать накладных расходов на размещение ЗА. Если соглашение о вызове подразумевает сохранение регистров только вызывающей стороной, то в ЗА *q* не нужно предусматривать место для сохраненных регистров.

Если язык не допускает замыканий, то компилятор может сделать лучше, чем одна статическая ЗА на каждую листовую процедуру. В любой точке выполнения активна может быть только одна листовая процедура. (Чтобы были активны две, первая должна была вызвать какую-то другую процедуру, но тогда она не была бы листовой.) Таким образом, компилятор может выделить одну статическую ЗА для использования всеми листовыми процедурами. Статическая ЗА должна быть достаточно большой, чтобы вместить данные любой листовой процедуры. Статические переменные, объявленные в любой из листовых процедур, можно разместить в этой единственной ЗА, которая устраняет необходимость в отдельных статических ЗА для каждой листовой процедуры.

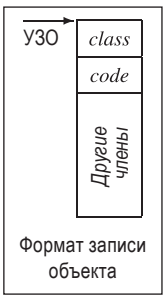
Объединение записей активации

Если компилятор обнаруживает набор процедур, которые всегда вызываются в одной и той же последовательности, то он может объединить их ЗА. Например, если вызов *q* из *p* всегда приводит к вызовам *r* и *s*, то компилятор может счесть выгодным выделить ЗА для *q*, *r* и *s* одновременно. Объединение ЗА помогает уменьшить затраты на выделение памяти; достигаемая экономия

напрямую зависит от величины этих затрат. На практике такая оптимизация ограничена раздельной компиляцией и использованием параметров-функций. То и другое мешает компилятору определить, какие именно вызовы реально происходят во время выполнения.

6.3.2. Поддержка объектно-ориентированных языков на этапе выполнения

Как Algol-подобные языки нуждаются в структурах времени выполнения, таких как ЗА, для поддержки лексических пространств имен, так объектно-ориентированным языкам нужны аналогичные структуры для поддержки лексической иерархии и иерархии классов. Некоторые структуры идентичны используемым в Algol-подобных языках. Например, управляющая информация для методов, а также память для локальных имен методов находится в ЗА. Другие структуры предназначены для удовлетворения потребностей, специфических для ООЯ. Так, время жизни объектов необязательно совпадает со временем жизни какого-то конкретного метода активации, поэтому их состояние нельзя сохранить ни в какой ЗА. Следовательно, каждому объекту необходима своя собственная запись объекта (ЗО) для хранения его состояния. ЗО классов порождают иерархию наследования, они играют важную роль в трансляции и выполнении.



Формат записи объекта

Каждая ЗО начинается членом *class*, содержащим класс объекта, а за ним следует член *code*, указывающий на вектор методов класса, как показано во врезке. Все прочие члены, зависящие от класса, расположены после *code*. Чтобы найти ЗО, код хранит указатель на ЗО (УЗО), который ведет на позицию ЗО с фиксированным смещением от ее начала; на рисунке предполагается, что эта позиция совпадает с началом ЗО.

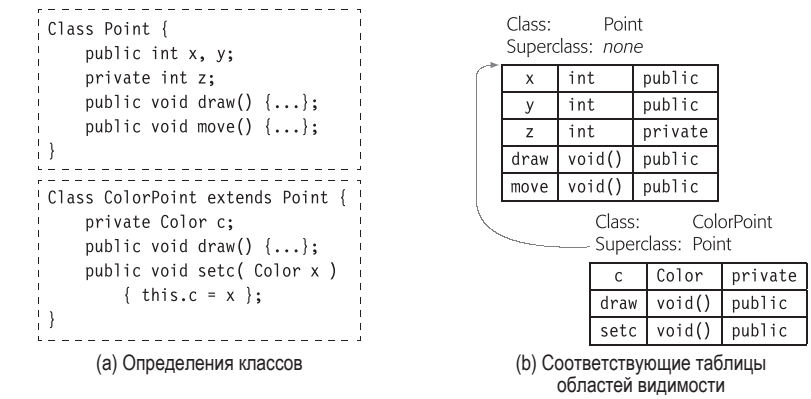
Объем поддержки на этапе выполнения, необходимый ООЯ, сильно зависит от особенностей ООЯ. Чтобы объяснить диапазон возможностей, мы начнем с одиночного наследования и открытой структуры классов, обратившись к примеру из раздела 5.4.2. Затем мы рассмотрим, какие последствия для поиска методов влечет замкнутая структура классов.

На рис. 6.3(a) и (b) повторен пример из главы 5. На рисунке (c) показаны структуры времени выполнения, которые могут быть результатом создания объектов `Point` и `ColorPoint`. У классов `Point`, `ColorPoint` и объектов `aPoint` и `aColorPoint` имеются собственные ЗО. ЗО для `class` не поместилась на странице.

На рисунке показаны полные векторы методов в каждом классе в соответствии с префиксной организацией.

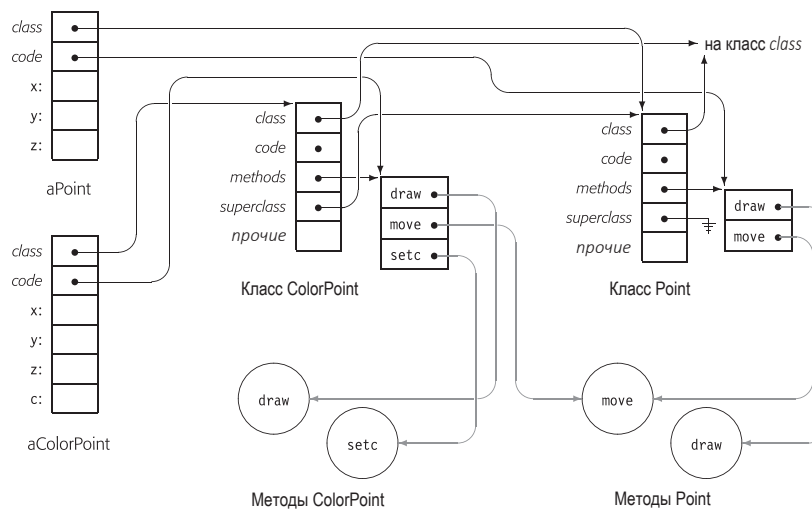
ЗО для `aPoint` проста. Она содержит указатель на класс `aPoint`, указатель на вектор методов `aPoint` (в классе `Point`) и место для данных-членов `Point`. ЗО для `aColorPoint` столь же проста. Она содержит указатель

на класс `aColorPoint`, указатель на вектор методов `aPoint` (в классе `ColorPoint`) и место для данных-членов `ColorPoint`. Данные-члены упорядочены в соответствии со схемой префиксной организации для одиночного наследования, которая обсуждалась в разделе 5.6.3. Следовательно, смещения унаследованных данных-членов `x`, `y` и `z`, такие же, как в экземпляре `Point`, а дополнительные данные-члены, объявленные в классе `ColorPoint`, расположены под ними.



(a) Определения классов

(b) Соответствующие таблицы областей видимости



(c) Записи объектов для `Point`, `ColorPoint`, `aPoint` и `aColorPoint`

Рис. 6.3 Объект: размещение в памяти, связи и наследование

ЗО для класса имеет похожую структуру. В каждой ЗО поле `class` содержит указатель на класс. Указатели `code`, на рисунке не показанные, вели бы на вектор методов в классе `class`. В ЗО классов показаны указатели на вектор методов и суперкласс. Безусловно, ЗО классов должны были бы содержать дополнительные члены конкретного класса.

Выделение памяти для записей объектов

Память для ЗО выделяется явно в момент создания объекта и освобождается, когда объект становится недостижим. Большинство ЗО размещаются в куче, потому что время жизни объекта обычно не коррелировано с какой-то одной активацией некоторого метода. Если время жизни объекта ограничено активацией какого-то метода, то его ЗО можно хранить в ЗА, созданной для этого метода.

В процессе анализа может выясниться, что размещенную в куче ЗО можно было бы вместо этого разместить в ЗА какого-то метода. Это преобразование может уменьшить затраты на выделение памяти и управление кучей.

Вызов метода

Как компилятор генерирует код для вызова метода, например `draw`? Методы всегда вызываются от имени какого-то объекта-получателя, скажем экземпляра `aColorPoint`. Чтобы вызов считался допустимым, `aColorPoint` должен быть виден в точке вызова, дабы компилятор мог найти `aColorPoint` в таблице символов. Сначала компилятор производит поиск в лексической иерархии метода, затем в иерархии классов и, наконец, в глобальной области видимости. Этот поиск дает достаточно информации, чтобы компилятор мог сгенерировать код получения УЗО `aColorPoint`.

Сгенерировав код получения УЗО, компилятор может найти и прочитать указатель на вектор методов, расположенный по фиксированному смещению от УЗО. А указатель на `draw` расположен по фиксированному смещению от начала вектора методов. (На рисунке это смещение равно нулю.) Компилятор может использовать адрес `draw` для стандартного вызова процедуры с одним дополнением: в вызове `draw` передается УЗО `aColorPoint` в качестве неявного первого параметра, привязанного к зависящему от языка имени, например `this` или `self`.

Благодаря продуманному построению структур данных времени выполнения эта стратегия работает. Поле `code` в ЗО `aColorPoint` указывает на вектор методов `ColorPoint`, а не `Point`. Следовательно, вызов `draw` от имени `aColorPoint` приводит к вызову `ColorPoint.draw`. Вызов `draw` от имени `aPoint` привел бы к вызову `Point.draw`. Наконец, приведение `aColorPoint` к типу `Point` привело бы к использованию вектора методов `Point` и вызову `Point.draw`.

В примере на рис. 6.3 каждый класс показан со своим полным вектором методов. Так, вектор методов `ColorPoint` содержит указатели на `ColorPoint.draw`, `Point.move` и `ColorPoint.setc`. Вектор – плод разрешения этих имен с помощью иерархии наследования. Описанная схема достигает желаемого результата – от имени объекта класса `x` вызывается реализация метода, видимая внутри класса `x`.

В качестве альтернативы компилятор может представлять только локальные методы `ColorPoint` в его векторе методов класса. Имени метода ставилась бы в соответствие координата, сообщающая компилятору, как далеко по цепочке суперклассов он должен подняться, чтобы найти указатель на код.

Тогда компилятор сгенерировал бы код перемещения по цепочке суперклассов до нужного уровня и там прочитал бы указатель на код. Полные векторы методов экономят время работы за счет небольшого увеличения потребной памяти.

Статическая и динамическая диспетчеризация

Диспетчеризация
Процесс вызова метода часто называют *диспетчеризацией*; этот термин заимствован из модели передачи сообщений в ООЯ, применяемой, например, в SMALLTALK.

До сих пор мы предполагали, что каждый вызов метода требует поиска его реализации с использованием ЗО получателя. Но в языке с замкнутой структурой классов компилятор может разрешить имя метода, т. е. сопоставить ему конкретную реализацию, еще на этапе компиляции и, стало быть, сгенерировать прямой вызов. Например, в C++ компилятор может сам найти конкретную реализацию любого метода, если только он не объявлен *виртуальным*, т. е. если программист не указал, что хочет искать реализацию относительно класса получателя.

В случае виртуального метода механизм диспетчеризации находит реализацию с помощью подходящего вектора методов. Компилятор генерирует код прохода по пути от ЗО к вектору методов класса и далее к указателю на код; этот процесс часто называют *динамической диспетчеризацией*. Но если компилятор C++ может доказать, что для некоторого вызова виртуального метода известен инвариантный класс получателя, то вправе сгенерировать прямой вызов, иногда это называют *статической диспетчеризацией*.

В языках с открытой структурой классов динамическая диспетчеризация может оказаться необходимой. Если структура классов может изменяться во время выполнения, то компилятор не способен разрешать имена методов, сопоставляя им реализации, а должен отложить этот процесс до этапа выполнения. Способов решения данной проблемы несколько: пересчет векторов методов при изменении иерархии классов, перекомпиляция классов JIT-компилятором, разрешение имен во время выполнения и поиск в иерархии классов во время выполнения.

- Если иерархия классов изменяется редко, то реализация может просто перестраивать векторы методов затронутых классов после каждого изменения. При такой схеме система времени выполнения должна обойти иерархию классов и найти в ней реализации методов во время построения векторов методов подклассов. Изменение структуры класса делает недействительным любой JIT-компилированный код относящихся к нему методов (см. главу 14).
- Если иерархия классов изменяется часто, то реализация может хранить неполные векторы методов в каждом классе, т. е. в векторе методов класса хранятся только указатели на методы, локальные в этом классе. В этой схеме вызов метода суперкласса инициирует поиск во время выполнения первого метода с таким именем в иерархии классов.

Кеши методов

Чтобы поддержать открытую иерархию классов, компилятору, возможно, придется создать ключ поиска для каждого имени метода и завести отображение пар ⟨*класс*, *ключ*⟩ на реализации, допускающее поиск на этапе выполнения. При такой схеме диспетчеризация включает поиск по ключу в таблицах иерархии классов.

Чтобы ускорить поиск метода в этой ситуации, система времени выполнения может реализовать *кеш методов* – программный аналог процессорного кеша данных. Каждая запись кеша методов состоит из ключа, класса и указателя на код. Динамическая диспетчеризация начинается с поиска в кеше методов; если она находит пару ⟨*класс*, *ключ*⟩, то возвращает указатель на код. Если же поиск завершается неудачно, то механизм диспетчеризации продолжает поиск по цепочке суперклассов класса получателя. Найдя метод, он кеширует результат и возвращает указатель на код.

Конечно, создание новой записи может потребовать вытеснения какой-то существующей. Для этого можно применить стандартные политики вытеснения, например по давности использования или циклическую. В большом кеше можно сохранить больше информации, но потребуется больше памяти и увеличится время поиска.

Чтобы уловить локальность типов в индивидуальных вызовах, в некоторых системах используется *встроенный кеш методов* – кеш из одной записи, находящийся в каждой точке вызова. В кеше хранится класс получателя и указатель на код для последнего выполнения в этой точке. Если текущий класс получателя совпадает с кешированным, то используется кешированный указатель на код. В противном случае выполняется полный поиск, и его результат сохраняется в кеше.

После любого изменения иерархии классов необходимо сделать недействительными все затронутые им кеши методов. Реализация могла бы очистить кеш методов и встроенные кеши; по мере выполнения кода они снова заполнятся. Альтернативное решение, которое хорошо работает для встроенных кешей, – сгенерировать новые теги для всех затронутых классов. Старые теги не совпадут, а новые поиски найдут новые методы. Устаревшие записи рано или поздно будут вытеснены.

Ключи можно было бы хранить в таблице, локальной для класса, или же вектор методов мог бы состоять из пар ⟨*ключ, адрес*⟩.

В любом случае языковой среде выполнения понадобятся таблицы имен методов – либо исходных, либо в виде ключей поиска, сгенерированных на их основе. Концептуально каждому классу необходим маленький словарь. В процессе разрешения имени производится поиск имени метода в иерархии, напоминающий поиск в цепочке таблиц символов, описанный в разделе 4.5.1.

В реализациях ООЯ применяются две основные стратегии уменьшения стоимости динамической диспетчеризации. Код анализируется с целью доказать, что в данном месте вызова всегда используется объект-получатель одного и того же известного класса, и если это так, то динамическую диспетчеризацию можно заменить статической. Для тех вызовов, где класс получателя изменяется или неизвестен, реализация может кешировать результаты поиска, чтобы ускорить последующие вызовы. При такой схеме механизм диспетчеризации сначала просматривает кеш методов, а затем производит поиск в иерархии классов. Если класс получателя и имя метода найдены в кеше, то используется кешированный указатель на код метода.

ИТОГИ РАЗДЕЛА

В Algol-подобных языках обычно используются лексические области видимости, когда пространства имен вложены, и новые экземпляры имени маскируют старые. При каждом вызове создается запись активации для вызываемой стороны, которая включает область локальных данных последней. В объектно-ориентированных языках в пространство имен добавляется иерархия наследования, основанная на определениях классов. Такое пространство имен с двойной иерархией ведет к более сложным взаимодействиям между именами и более сложным реализациям.

Для обоих стилей именования необходимы структуры времени выполнения, чтобы правильно реализовать иерархию имен. В случае Algol-подобных языков структура пространства имен улавливается записями активации, которые предоставляют место для большинства значений и сохраняют состояние, необходимое для правильного выполнения. В ООЯ записи активации для методов по-прежнему улавливают лексическую сторону пространства имен и состояние выполнения. Однако реализации также необходима иерархия записей объектов, улавливающая аспект пространства имен, связанный с наследованием.

ВОПРОСЫ

1. В языке C функции `setjmp` и `longjmp` предлагают механизм межпроцедурной передачи управления. `setjmp` строит структуру, инкапсулирующую состояние среды выполнения; вызов `longjmp` с передачей этой среды восстанавливает состояние и позволяет продолжить выполнение, как будто последний вызов `setjmp` только что вернул управление. Какую информацию должна сохранять `setjmp`? Как реализация `setjmp` изменяется в зависимости от того, размещены ЗА в стеке или в куче?
2. Рассмотрите пример на рис. 6.3. Если программист приводит `aColorPoint` к классу `Point`, то какие действия должен предпринять сгенерированный код, чтобы это приведение состоялось и программа вела себя правильно? (Напомним, что приведение имеет локальный эффект, так что код не должен изменять ЗО получателя навсегда.)

6.4. Передача значений между процедурами

Главная идея процедур – абстрагирование. Программист абстрагирует часто выполняемые операции с небольшим множеством имен, или формальных параметров, и инкапсулирует эти операции в процедуре. Чтобы воспользоваться процедурой, программист вызывает ее с надлежащей *привязкой* значений, или фактических параметров, к формальным параметрам. Вызываемая сторона выполняется, используя формальные параметры для доступа к значениям, переданным в качестве фактических параметров. По желанию программиста процедура может вернуть результат.

6.4.1. Передача параметров

Связывание параметров отображает фактические параметры в месте вызова на формальные параметры вызываемой стороны. Это позволяет программисту писать процедуру, не зная, из каких контекстов она может быть вызвана. Программист может вызывать процедуру из разных контекстов, не раскры-

вая детали ее работы вызывающей стороне. Следовательно, связывание параметров играет важную роль для нашей способности писать абстрактный, модульный код.

В большинстве современных языков программирования используется одно из двух соглашений об отображении фактических параметров в формальные: *передача по значению* и *передача по ссылке*. Различие между ними проще всего объяснить, описав, как они реализованы.

Передача по значению

Передача по значению
соглашение, при котором вызывающая сторона вычисляет фактические параметры и передает их значения вызываемой стороне.

Изменения переданных по значению параметров, произведенные вызываемой стороной, не видны вызывающей стороне.

Рассмотрим следующий С-код – процедуру `fee` и несколько мест, из которых она вызывается:

```
int fee(int x, int y) {
   x = 2 * x;
   y = x + y;
   return y;
}
```

```
c = fee(2,3);
a = 2;
b = 3;
c = fee(a,b);
a = 2;
b = 3;
c = fee(a,a);
```

В случае передачи по значению, как в С, вызывающая сторона копирует значение фактического параметра в место, соответствующее формальному параметру, – либо в регистр, либо в позицию параметра в ЗА вызываемой стороны. На это значение ссылается только одно имя – формального параметра. Его значение является начальным условием, определенным в результате вычисления фактического параметра в момент вызова. Если вызываемая сторона изменит это значение, то изменение будет видно вызываемой стороне, но для вызывающей останется невидимым.

Три показанных выше вызова порождают следующие результаты в предположении передачи параметров по значению:

Передача по значению	a		b		Возвращенное значение
	вход	выход	вход	выход	
`fee(2,3)`	–	–	–	–	7
`fee(a,b)`	2	2	3	3	7
`fee(a,a)`	2	2	3	3	6

В случае передачи по значению связывание простое и интуитивно понятное.

Один из вариантов передачи по значению называется *передачей по значению с модификацией* (*call-by-value-result*). При такой схеме значения формальных параметров копируются в соответствующие фактические параметры в момент возврата из вызываемой стороны. Язык программирования ADA включает такой механизм передачи параметров. Он применяется и в некоторых реализациях FORTRAN 77.

Передача по ссылке

Передача по ссылке
соглашение, при котором компилятор передает вызываемой стороне адрес фактического параметра.

Если фактический параметр является переменной, то изменение значения формального параметра отражается и на значении фактического.

В случае передачи параметра *по ссылке* вызывающая сторона передает не значение, а адрес. Если фактический параметр размещен в памяти, то передается его адрес в памяти. Если фактический параметр является выражением, то вызывающая сторона вычисляет выражение, сохраняет его значение в своей области локальных данных и передает адрес этого места. Значения, находящиеся в регистрах, и константы обрабатываются так же, как выражения. Внутри вызываемой стороны все ссылки на формальные параметры требуют дополнительного уровня косвенности.

Передача по ссылке отличается от передачи по значению в двух важных отношениях. Во-первых, если вызывающая сторона передает по ссылке переменную *x* в фактическом параметре, привязанном к переменной *y* вызываемой стороны, то любое изменение *y* отражается также на *x*. Во-вторых, если вызываемая сторона может обращаться к *x* напрямую, то внутри нее оказываются доступны два имени, что может приводить к поведению, противоречащему интуиции. Когда вызываемая сторона располагает двумя именами одного и того же места в памяти, говорят, что имена являются *псевдонимами*.

Рассмотрим предыдущий пример, переписанный на PL/I, когда параметры передаются по ссылке.

```
fee: procedure (x,y)                    c = fee(2,3);
         returns fixed binary;          a = 2;
     declare x, y fixed binary;         b = 3;
     x = 2 * x;                         c = fee(a,b);
     y = x + y;                         a = 2;
     return y;                          b = 3;
     end fee;                           c = fee(a,a);
```

Псевдоним
Два имени, ссылающихся на одно и то же место в памяти, называются *псевдонимами*.
В примере вызов `fee(a,a)` делает `x` и `y` внутри `fee` псевдонимами.

В случае передачи параметров по ссылке результаты получаются иными. Первый вызов не вызывает сомнений. Во втором вызове `a` и `b` переопределяются, и эти изменения видны вызывающей стороне. В третьем вызове `x` и `y` ссылаются на одно и то же место в памяти, а значит, на одно и то же значение. Такое наличие *псевдонима* изменяет поведение `fee`. При первом присваивании `a` и `b` получают значение 4. При втором присваивании `a` и `b` получают значение 8, и `fee` возвращает 8. Вызов же `fee(2,2)` с литеральными константами вернул бы 6.

Передача по ссылке	a		b		Возвращенное значение
	вход	выход	вход	выход	
`fee(2,3)`	–	–	–	–	7
`fee(a,b)`	2	4	3	7	7
`fee(a,a)`	2	8	3	3	8

Место для параметров

Размер представления параметров влияет на стоимость вызова процедуры. Скалярные значения, такие как переменные и указатели, хранятся в регистрах или в области параметров ЗА вызываемой стороны. В случае передачи по значению хранится само значение параметра, а в случае передачи по ссылке – адрес параметра. В любом случае накладные расходы в расчете на один параметр невелики – как правило, одна операция сохранения.

Большие значения, такие как массивы, записи или структуры, при передаче по значению создают проблемы. Если язык требует, чтобы большие значения копировались, то затраты на копирование в область параметров вызываемой стороны заметно увеличивают стоимость вызова процедуры. (В этом случае программист, скорее всего, захочет смоделировать передачу по ссылке и будет передавать не саму сущность, а указатель на нее.) Некоторые языки позволяют реализации передавать такие сущности по ссылке. Другие включают средства, позволяющие программисту указать, что передача конкретного параметра по ссылке безопасна и приемлема, например добавление атрибута параметра `const` в прототип функции в C сообщает компилятору, что параметр не модифицируется при вызове функции.

6.4.2. Возврат значений

В случае передачи параметров по значению некоторые соглашения о связи постулируют, что регистр, в котором передается первый параметр, используется также для возвращаемого значения.

Чтобы вернуть значение из функции, компилятор должен отвести место для возвращаемого значения. Поскольку возвращенное значение, по определению, используется после возврата, ему нужно место вне ЗА вызываемой стороны. Если автор компилятора может гарантировать, что размер возвращаемого значения фиксирован и невелик, то его можно сохранить либо в ЗА вызывающей стороны, либо в специальном регистре.

Во всех приведенных выше рисунках в ЗА было предусмотрено место для возвращенного значения. Чтобы воспользоваться им, вызывающая сторона выделяет память для возвращенного значения в своей ЗА и сохраняет указатель на эту память в точно известной позиции ЗА. Вызываемая сторона может прочитать указатель из этой позиции (воспользовавшись копией УЗА, присутствующей в ЗА вызываемой стороны). Имея этот указатель, вызываемая сторона может получить доступ к памяти в ЗА вызывающей

стороны, отведенной для возвращенного значения. При условии что обе стороны согласовали размер и тип возвращенного значения, такой подход работает.

Передача параметра по имени

В ALGOL была введена *передача параметра по имени*. При таком связывании ссылка на формальный параметр ведет себя так, будто фактический параметр был текстуально подставлен в нужное место с приличествующим случаю переименованием. Это простое правило может порождать сложное поведение. Рассмотрим следующий короткий пример на языке ALGOL-60:

```
begin comment Пример простого массива;
    procedure zero(Arr, i, j, u1, u2);
        integer Arr, i, j, u1, u2;
        begin;
            for i := 1 step 1 until u1 do
                for j := 1 step 1 until u2 do
                    Arr := 0;
            end;

    integer array Work[1:100,1:200];
    integer p, q, x, y;
    x := 100;
    y := 200;
    zero(Work[p,q], p, q, x, y);
end
```

Вызов `zero` обнуляет все элементы массива `Work`. Чтобы убедиться в этом, перепишем `zero`, подставив текст фактических параметров.

Хотя определить передачу параметров по имени просто, реализовать и понять ее трудно. В общем случае компилятор должен для каждого формального параметра сгенерировать функцию, которая вычисляет фактический параметр и возвращает указатель. Такие функции называются *переходниками* (thunk). Генерировать правильные переходники трудно, а вычислять переходник для доступа к каждому параметру дорого. В конечном итоге недостатки перевесили теоретические преимущества передачи параметров по имени.

В языке программирования R передача по имени применяется для создания ленивых параметров. Вызов создает и передает переходники, или *обещания*. Код вычисляет переходник, когда встречает первую ссылку на параметр, и сохраняет результат для последующего использования.

Если вызывающая сторона не знает размер возвращенного значения, то место для него может выделить вызываемая сторона, предположительно в куче. В таком случае вызываемая сторона выделяет память, сохраняет в ней возвращаемое значение, а указатель на эту память помещает в место, отведенное для возвращенного значения в ЗА вызывающей стороны. Выделенная в куче память должна быть освобождена – либо явно вызывающей стороной, либо неявно сборщиком мусора.

Если возвращаемое значение мало – такого же размера, как место для возвращенного значения в ЗА вызывающей стороны, или меньше, – то компилятор может устранить косвенность. Тогда вызываемая сторона сохраняет значение прямо в месте, отведенном для возвращенного значения в ЗА вызывающей стороны. Разумеется, при этом требуется, чтобы компилятор обрабатывал значение согласованно на вызывающей и вызываемой сторонах. По счастью, сигнатура типа процедуры может гарантировать передачу информации, необходимой для компиляции обеих сторон.

6.4.3. Обеспечение адресуемости нелокальных переменных

Базовый адрес
Адрес начала области данных часто называют ее *базовым адресом*.

Как часть соглашения о связях компилятор должен гарантировать, что каждая процедура может сгенерировать адрес любой переменной, на которую ей необходимо сослаться. В общем случае вычисление адреса состоит из двух частей: нахождение *базового адреса* нужной области данных для области видимости, содержащей значение, и нахождение смещения от начала этой области данных. В задаче нахождения базового адреса можно выделить три случая: области данных со статическими базовыми адресами, области данных, адрес которых неизвестен до этапа выполнения, и объекты, память для которых выделена в куче.

Переменные со статическими базовыми адресами

Обычно компиляторы организуют память так, что области глобальных и статических данных имеют статические адреса. Чтобы получить адрес переменной в одной из этих областей, компилятор генерирует код, который помещает базовый адрес области данных в регистр и прибавляет к нему смещение переменной. Эти вычисления настолько частые, что в большинстве ISA имеется специальный режим адресации, позволяющий выполнять их эффективно. Например, в ILOC имеются режимы «регистр + непосредственное значение» (`loadAI`) и «регистр + регистр» (`loadAO`).

Чтобы сгенерировать значение статического базового адреса на этапе выполнения, компилятор присоединяет к области данных символическую метку уровня ассемблера. В зависимости от набора команд целевой машины эту метку можно использовать в операции непосредственной загрузки или для инициализации позиции в пуле констант; в последнем случае ее можно переместить в регистр стандартной операцией загрузки.

Декорирование имени
процесс построения уникальной строки из имени в исходной программе.

Компилятор конструирует метку базового адреса с помощью *декорирования* имени. Обычно он добавляет к оригинальному имени префикс, суффикс или то и другое сразу,

включая литеры, допустимые в языке ассемблера, а не в исходном языке. Например, результатом декорирования имени глобальной переменной `fee` может быть метка `&fee.`; затем эта метка присоединяется к ассемблерной псевдооперации, которая резервирует место для `fee`. Чтобы переместить адрес в регистр, компилятор должен сгенерировать операцию типа `loadI &fee.` ⇒ r_i. Последующие операции могут использовать r_i для доступа к месту в памяти, где хранится `fee`. Метка становится перемещаемым символом для ассемблера и загрузчика, который преобразует ее в виртуальный адрес времени выполнения.

Если значение `&fee.` слишком длинное для операции непосредственной загрузки, то компилятору придется сохранить адрес в известном месте и загружать его оттуда.

Глобальные переменные можно помечать по отдельности или большими группами. Например, язык FORTRAN позволяет собрать глобальные переменные в общие блоки. Типичный компилятор FORTRAN ставит одну метку на каждый общий блок. Он назначает смещение каждой переменной в каждом общем блоке и генерирует операции `load` и `store` относительно метки общего блока. Если область данных больше, чем смещение, допустимое в операции «регистр + смещение», то может оказаться разумно присвоить разные метки частям области данных.

Аналогично компилятор может объединить все статические переменные в одной области видимости в одну область данных. Это уменьшает вероятность неожиданного конфликта имен; такие конфликты обнаруживаются во время компоновки или загрузки и могут неприятно удивить программиста. Чтобы их избежать, компилятор может положить в основу метки глобально видимое имя, ассоциированное с областью видимости. Эта стратегия уменьшает количество базовых адресов, используемых одновременно, а стало быть, и спрос на регистры. Использование слишком большого числа регистров для хранения базовых адресов может негативно повлиять на общую производительность программы.

Локальные переменные текущей процедуры

Доступ к локальной переменной текущей процедуры не вызывает трудностей. Хотя ее виртуальный адрес на этапе компиляции неизвестен, базовым адресом является просто адрес текущей ЗА, хранящийся в УЗА. Далее, компилятор знает смещение переменной от начала области локальных данных в этой ЗА. (Компилятор сам назначил это смещение в процессе организации памяти и протащил его дальше в статической координате переменной.) Таким образом, компилятор может сгенерировать код, который прибавляет смещение переменной к УЗА и тем самым порождает ее виртуальный адрес. Почти все процессоры предоставляют режим адресации для поддержки операций вида база+смещение. В ILOC это команды `loadAI`, `loadAO`, `storeAI` и `storeAO`.

В некоторых случаях значение не хранится в месте с постоянным смещением от УЗА. Оно может храниться в регистре, и тогда операции загрузки

и сохранения не нужны. Если размер переменной непредсказуем или может изменяться, то компилятор будет сохранять ее в области, зарезервированной для объектов переменного размера, – в конце области локальных данных или в куче. В этом случае компилятор может зарезервировать в ЗА место для указателя на фактическое местонахождение переменной, получить доступ к этому указателю с помощью операции УЗА + смещение, а затем использовать его для доступа к переменной.

Локальные переменные других процедур

Чтобы можно было получить доступ к локальной переменной в какой-то из объемлющих областей видимости, компилятор должен построить структуры времени выполнения, которые поддерживают поиск областей локальных данных, ассоциированных с такими областями видимости. Он также должен генерировать код, использующий эти структуры для нахождения необходимых виртуальных адресов.

Например, предположим, что процедура `fee` на лексическом уровне *m* ссылается на переменную `x`, определенную на лексическом уровне *n* в процедуре `fie`, $n < m$. Парсер может сконструировать статическую координату `x`, $\langle n, o \rangle$, где *o* – смещение `x` относительно начала ЗА процедуры `fie`. Чтобы преобразовать $\langle n, o \rangle$ в адрес времени выполнения, компилятор генерирует код, в котором *n* используется для нахождения УЗА последней созданной процедуры уровня *n*, а затем к этому УЗА прибавляется *o* для вычисления виртуального адреса `x` времени выполнения.

Конечно, этот план зависит от доступности во время выполнения структур данных, которые могут найти УЗА уровня *n*. Компилятор должен генерировать код для построения и обслуживания таких структур. Два самых распространенных метода называются «связи доступа» и «глобальный дисплей». Они рассматриваются в следующих подразделах.

Связи доступа

Интуитивно связи доступа просты. Компилятор заботится о том, чтобы в каждой ЗА был указатель, называемый *связью доступа*, или *статической связью*, на ЗА непосредственно объемлющей области видимости. Связи доступа образуют цепочку, включающую всех лексических предков текущей процедуры, как показано на рис. 6.4. Следовательно, любая локальная переменная любой процедуры, которая видна текущей процедуре, хранится в какой-то ЗА, принадлежащей цепочке связей доступа, начинающейся текущей процедурой.

Чтобы получить доступ к значению $\langle n, o \rangle$ в процедуре уровня *m*, компилятор генерирует код обхода цепочки связей и нахождения УЗА уровня *n*. Затем он генерирует команду загрузки, использующую этот УЗА уровня *n* и *o*. Для конкретики рассмотрим программу, представленную на рис. 6.4. Предположим, что *m* равно 2 и что связь доступа хранится в поле со смещением –4 от УЗА. В следующей таблице показаны набор из трех статических координат

и ILOC-код, который компилятор мог бы сгенерировать для них. Каждая последовательность оставляет результат в r_2.

Координата	Код
⟨2, 24⟩	loadAI r_{arp}, 24 ⇒ r_2
⟨1, 12⟩	loadAI r_{arp}, -4 ⇒ r_1
	loadAI r_1, 12 ⇒ r_2
⟨0, 16⟩	loadAI r_{arp}, -4 ⇒ r_1
	loadAI r_1, -4 ⇒ r_1
	loadAI r_1, 16 ⇒ r_2



Рис. 6.4 Использование связей доступа

Поскольку компилятор располагает статическими координатами любой ссылки, он может вычислить статическое расстояние (m–n), которое говорит, сколько нужно сгенерировать команд загрузки для следования вдоль цепочки. Стоимость вычисления адреса пропорциональна этому статическому расстоянию. Если лексическая вложенность в программе неглубокая, то различие в стоимости доступа к двум переменным на разных уровнях будет сравнительно небольшим.

Для обслуживания связей доступа компилятор должен добавить в каждый вызов процедуры код, который находит надлежащий УЗА и сохраняет его в качестве связи доступа вызываемой стороны. Если вызывающая сторона расположена на уровне p, а вызываемая – на уровне q, то возможны три случая. Если $q = p + 1$, то вызываемая сторона вложена в вызывающую, и вызываемая сторона может использовать УЗА вызывающей в качестве своей связи доступа. Если $q = p$, то связь доступа вызываемой стороны совпадает со связью доступа вызывающей. Наконец, если $q < p$, то связь доступа вызываемой стороны является связью доступа уровня $q - 1$ для вызывающей. (Если q

равен нулю, то связь доступа содержит null.) Компилятор может сгенерировать последовательность $p - q + 1$ операций загрузки, чтобы найти этот УЗА и сохранить его в качестве связи доступа вызываемой стороны.

Глобальный дисплей

В этой схеме компилятор выделяет память для одного глобального массива, называемого *дисплеем*, в котором будет храниться УЗА последней активации процедуры на каждом лексическом уровне. Ссылки на локальные переменные других процедур становятся косвенными ссылками через дисплей. Для доступа к переменной со статической координатой $\langle n, o \rangle$ компилятор сначала извлекает из дисплея УЗА уровня n. Затем он использует этот УЗА совместно со смещением o операции загрузки или сохранения по адресу база + смещение. В ссылках на локальные переменные текущей процедуры можно по-прежнему напрямую использовать УЗА из r_{arp}.

На рис. 6.5 показано состояние программы с рис. 6.4, представленное с помощью глобального дисплея, а не связей доступа. В дисплее имеется одна запись на каждый лексический уровень программы. Память для самого дисплея можно выделить статически и сопоставить ей известную декорированную метку. Неиспользуемые записи дисплея содержат недопустимый указатель.

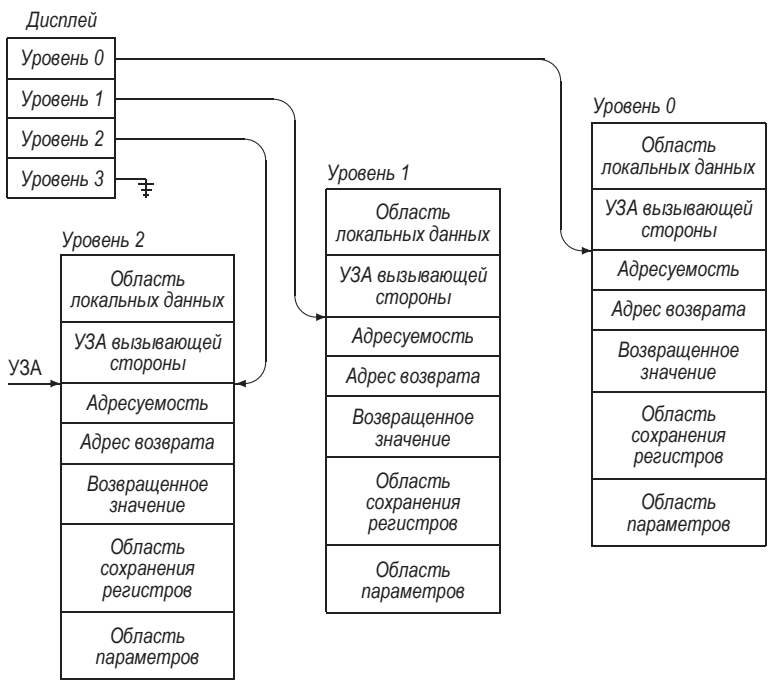


Рис. 6.5 Использование глобального дисплея

Используя те же статические координаты, что в обсуждении связей доступа, компилятор может сгенерировать следующий код для реализации на

основе дисплея. Предположим, что текущая процедура находится на лексическом уровне 2, а метка `_&disp_`: дает адрес дисплея.

Координата	Код
⟨2, 24⟩	loadAI r_{arp}, 24 ⇒ r_2
⟨1, 12⟩	loadI _&disp_ ⇒ r_1
	loadAI r_1, 4 ⇒ r_1
	loadAI r_1, 12 ⇒ r_2
⟨0, 16⟩	loadI _&disp_ ⇒ r_1
	loadAI r_1, 0 ⇒ r_1
	loadAI r_1, 16 ⇒ r_2

В случае дисплея стоимость нелокального доступа фиксирована. В случае связей доступа компилятор генерирует последовательность $m–n$ операций загрузки; в случае дисплея он использует $n×l$ в качестве смещения от начала дисплея, где l – длина указателя (в примере – 4). Локальный доступ все равно дешевле нелокального, но в случае дисплея штраф за нелокальный доступ постоянный.

Разумеется, компилятор должен вставить в нужные места код для обслуживания дисплея. Таким образом, когда процедура p на уровне n вызывает процедуру q на уровне n+1, УЗА p становится записью дисплея для уровня n. (При выполнении p эта запись не используется.) Самый простой способ обслуживания дисплея – поручить p обновление записи уровня n: в момент, когда управление попадает в p, заместить эту запись своим собственным УЗА, а при выходе из p восстановить прежнюю запись.

p может сохранить старую запись дисплея уровня n в поле адресуемости ЗА p.

Некоторых обновлений дисплея компилятор может избежать. Если процедура p не вызывает вложенные в нее процедуры, то ее собственный УЗА не может использоваться для нелокального доступа, и p необязательно обновлять дисплей. Это наблюдение позволяет устранить многие лишние обновления.

Чтобы устранить все лишние обновления, компилятору нужно было бы знать, на какие нелокальные переменные ссылается каждая процедура.

ИТОГИ РАЗДЕЛА

Способность передавать данные в процедуры и получать обратно результаты – то, что делает процедуры полезными. Для этого используются два разных механизма: связывание параметров и прямой доступ к именам в объемлющих областях видимости. Языки специфицируют различные механизмы передачи параметров, которые компилятор должен реализовать. Что до нелокального доступа, компилятор должен спланировать и обслуживать структуры времени доступа, которые указывают на нужные области данных.

Наиболее запутанным аспектом этого материала является различие между действиями на этапе компиляции, такими как вычисление статического расстояния между координатами парсером, и действиями на этапе выполнения, такими как код прохода по цепочке связей доступа. Действия на этапе компиляции выполняются самим компилятором. Для остальных действий он генерирует код, исполняемый на этапе выполнения.

ВОПРОСЫ

1. В ранних реализациях FORTRAN была странная ошибка. Коротенькая программа, показанная во врезке, печатала значение 16. Какие действия компилятора привели к этому результату? А что он должен был сделать? (В FORTRAN используется передача параметров по ссылке.)
2. Одна из проблем реализации замыканий – необходимость сохранять релевантные ЗА после создания замыкания. Какую роль могут сыграть связи доступа или дисплей в этом процессе? Является ли одна техника лучше другой в этой ситуации?

```
subroutine change(n)
  integer n
  n = n * 2
end

program test
  call change(2)
  print *, 2 * 2
end
```

6.5. Стандартные соглашения о связях

Связь с процедурой – это контракт между компилятором, операционной системой и целевой машиной, в котором четко распределена ответственность за именование, выделение ресурсов, адресуемость и защиту. Соглашение о связях гарантирует интероперабельность процедур, оттранслированных компилятором, с кодом из других источников, в т. ч. системных и прикладных библиотек, и с кодом, написанным на других языках программирования. Обычно все компиляторы для данной комбинации целевой машины и операционной системы используют одно и то же соглашение о связях в той мере, в какой это возможно.

Соглашение о связях изолирует процедуру от различных окружений, встречающихся в местах ее вызова. Предположим, что процедура *p* имеет целый параметр *x*. Различные вызовы *p* могли бы связать *x* с локальной переменной, хранящейся в кадре стека вызывающей стороной, с глобальной переменной, с элементом статического массива или с результатом вычисления выражения, например *y* + 2. Поскольку соглашение о связях описывает, как вычислять фактический параметр и как сохранять его значение, а также как обращаться к *x* на вызываемой стороне, компилятор может сгенерировать для вызываемой стороны код, игнорирующий различия между окружениями в разных местах вызова. При условии что все процедуры соблюдают соглашение о связях, будет организована передача значений, обещанная спецификацией исходного языка.

Соглашение о связях по необходимости машинно-зависимо. Например, оно неявно зависит от таких вещей, как количество доступных регистров целевой машины и механизм выполнения вызова и возврата.

На рис. 6.6 показано, как сочетаются части стандартного соглашения о связях с процедурой. У каждой процедуры имеются *пролог* и *эпилог*. В каждом месте вызова присутствуют *предварительная* и *заключительная* последовательности команд.

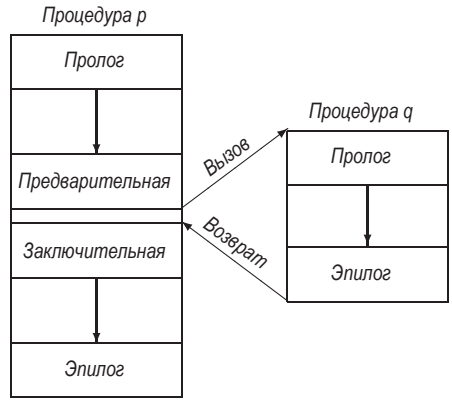


Рис. 6.6 Стандартное соглашение о связях

Предварительная последовательность

Предварительная последовательность начинает процесс сохранения окружения вызывающей стороны и создания окружения вызываемой стороны. Выделяется место для ЗА вызываемой стороны. Вычисляются фактические параметры, и их значения или адреса сохраняются в отведенном месте ЗА вызываемой стороны. Адрес заключительной последовательности сохраняется в качестве адреса возврата. УЗА вызывающей стороны сохраняется в ЗА вызываемой стороны. Регистры, сохраняемые вызывающей стороной, сохраняются в области сохранения регистров в ее ЗА. Также может производиться обслуживание структур данных адресуемости: связей доступа или дисплея.

> Если передаваемый по ссылке параметр в месте вызова находится в регистре, то предварительная последовательность должна будет сохранить его в ЗА вызывающей стороны, чтобы можно было взять его адрес.

Для активации вызываемой стороны предварительная последовательность записывает УЗА вызываемой стороны в регистр, выделенный для хранения текущего УЗА, и выполняет безусловный переход на код пролога вызываемой стороны.

Пролог

Пролог завершает процесс сохранения окружения вызывающей стороны и создания окружения вызываемой стороны. Сохраняются все регистры, сохраняемые вызываемой стороной, которые последняя будет использовать. Возможно, придется выделить место для области локальных данных вызываемой стороны. Вероятно, придется загрузить базовые адреса областей статических или глобальных данных, на которые ссылается вызываемая сторона. Может быть, также придется произвести обслуживание структур данных адресуемости: связей доступа или дисплея. Наконец, производится инициализация локальных переменных. После этого начинается выполнение основного кода вызываемой стороны.

Эпилог

Эпилог начинает процесс демонтажа окружения вызываемой стороны и восстановления окружения вызывающей стороны. Если процедура возвращает значение, то эпилог может сохранить его по адресу, заданному вызывающей стороной. Он также может обслуживать структуры данных адресуемости: связи доступа или дисплей. Если память для области локальных данных выделялась отдельно от ЗА, то эпилог освобождает эту память. Наконец, он загружает адрес возврата, восстанавливает УЗА вызывающей стороны и производит безусловный переход на адрес возврата.

Заключительная последовательность

Заключительная последовательность завершает процесс восстановления окружения вызывающей стороны. Она освобождает ЗА вызываемой стороны и копирует переданные по ссылке фактические параметры, которые должны быть возвращены, в регистры. Она восстанавливает регистры, сохраненные вызывающей стороной на предварительной стадии. И наконец, возобновляет выполнение вызывающей стороны с точки после вызова.

Эта схема дает общие рекомендации по разработке соглашений о связях. Многие задачи могут быть реализованы как вызываемой, так и вызывающей стороной. Вообще говоря, перемещение работы в пролог и эпилог приводит к более компактному коду. Предварительная и заключительная последовательности генерируются для каждого вызова, тогда как пролог и эпилог один раз для каждой процедуры. Если процедуры вызываются в среднем больше одного раза, прологов и эпилогов будет меньше, чем предварительных и заключительных последовательностей.

Многие значения, показанные на диаграммах ЗА, можно передать вызывающей стороне в регистрах. Очевидными кандидатами являются адрес возврата, адрес поля для возврата значения и УЗА вызывающей стороны. Многие соглашения предусматривают передачу первых *k* фактических параметров в регистрах; типичное значение *k* равно 3 или 4. Если передается более *k* параметров, то остальные можно разместить в ЗА вызываемой стороны.

Сохранение регистров

Значение *переживает* вызов, если оно было создано до вызова и может быть использовано после вызова.

В какой-то точке последовательности вызова значения всех регистров, которые, как ожидает вызывающая сторона, переживают вызов, должны быть сохранены в памяти. Сохранение может выполнить либо вызывающая, либо вызываемая сторона, у обоих решений есть плюсы и минусы. Если регистры сохраняет вызывающая сторона, то она может не сохранять те, которые заведомо не переживут вызов; в результате можно сохранять и восстанавливать меньше значений. С другой стороны, если регистры сохраняет вызываемая сторона, то она может не сохранять те, которые не использует;

и снова дополнительные знания могут уменьшить количество операций сохранений и восстановления.

В общем случае компилятор может использовать знания о компилируемой процедуре, чтобы оптимизировать сохранение регистров. Для любого разделения труда между вызывающей и вызываемой сторонами можно построить программы, которые будут работать хорошо и плохо. В большинстве современных систем выбрана золотая середина – часть набора регистров сохраняется вызывающей стороной, другая часть – вызываемой. На практике это, похоже, работает неплохо. Такой подход побуждает компилятор помещать долгоживущие значения в регистры, сохраняемые вызываемой стороной, где они будут сохраняться, только если вызываемая сторона их действительно использует. А короткоживущие значения компилятор помещает в регистры, сохраняемые вызывающей стороной, которая может решить, что сохранять их при вызове нет нужды.

Еще о времени

В типичной системе соглашение о связях обсуждается между авторами компиляторов и разработчиками операционной системы на ранней стадии. Поэтому такие вопросы, как разделение регистров на сохраняемые вызывающей и вызываемой сторонами, решаются на этапе проектирования. Компилятор должен генерировать пролог и эпилог для каждой процедуры, а также предварительную и заключительную последовательности в каждом месте вызова. Этот код исполняется во время выполнения. Компилятор не может знать адрес возврата, который должен сохранить в ЗА вызываемой стороны. (Да и сам адрес этой ЗА он в общем случае не знает.) Однако он может предусмотреть механизм, который будет генерировать адрес возврата на этапе компоновки (воспользовавшись перемещаемой меткой языка ассемблера) или на этапе выполнения (используя некое смещение от значения в регистре счетчика команд), и сохранять его в специальном поле в ЗА вызываемой стороны.

Аналогично в системе, где для адресуемости локальных переменных из других процедур используется дисплей, компилятор не может знать адреса дисплея или ЗА во время выполнения. Тем не менее он генерирует код для обслуживания дисплея. Соответствующий механизм нуждается в двух элементах информации: лексический уровень вложенности текущей процедуры и адрес глобального дисплея. Первый известен на этапе компиляции, последний можно определить на этапе компоновки, воспользовавшись перемещаемой ассемблерной меткой. Следовательно, пролог может просто загрузить текущую запись дисплея для известного уровня процедуры (командой `loadAO` относительно адреса дисплея) и сохранить ее в ЗА (командой `storeAO` относительно УЗА). Наконец, он сохраняет адрес новой ЗА в месте дисплея, отведенном для лексического уровня данной процедуры.

Выделение памяти для записи активации

В общем случае и вызывающая, и вызываемая сторона должна иметь доступ к ЗА вызываемой стороны. К сожалению, вызывающая сторона, вообще говоря, не может знать размер ЗА вызываемой стороны (если только компилятор и компоновщик не договорятся о том, что компоновщик помещает вычисленные значения в каждое место вызова).

Если память для ЗА выделяется в стеке, то возможна золотая середина. Поскольку выделение памяти сводится к увеличению указателя стека, вызывающая сторона может начать создание ЗА вызываемой стороны, сдвинув вершину стека и сохранив значения в надлежащих местах. Когда управление попадет вызываемой стороне, та сможет расширить частично построенную ЗА, сдвинув вершину стека еще, чтобы отвести место для локальных данных. Позже заключительная последовательность может восстановить положение вершины стека и тем самым освободить память одним махом.

Если память для ЗА выделяется в куче, то не всегда возможно расширить ЗА инкрементно. В таком случае у автора компилятора есть три варианта.

1. Компилятор может передать значения, которые должен сохранить в ЗА вызываемой стороны, в регистрах или в собственной ЗА вызывающей стороны. Затем пролог может выделить память надлежащего размера для ЗА и сохранить в ней переданные значения. Значения параметров могут находиться в ЗА вызывающей стороны. Для доступа к этим параметрам можно было бы использовать копию УЗА вызывающей стороны, сохраненную в ЗА вызываемой стороны.
2. Автор компилятора может разделить ЗА на несколько частей: в одной хранится управляющая информация и информация о параметрах, сгенерированная вызывающей стороной, а остальные содержат память, которая необходима вызываемой стороне, но размер которой неизвестен вызывающей стороне. Поскольку в общем случае вызывающая сторона не знает, сколько места отвести под область локальных данных, она может позволить вызываемой стороне самостоятельно выделить память под свои локальные данные. Вызываемая сторона могла бы отвести специальный регистр для хранения базового адреса этой области локальных данных.
3. Компилятор мог бы использовать схему, при которой каждая процедура создает инициализированную статическую константу с помощью декорированной метки. Компилятор может сохранить размер области локальных данных процедуры *fee* в метке, полученной на основе строки «*fee*». При любом вызове *fee* эту константу можно загрузить и использовать ее для определения размера области локальных данных.

Выделение памяти для ЗА в куче повышает стоимость вызова процедуры. Но продуманная реализация может снизить эти затраты.

Управление дисплеями и связями доступа

Какой бы механизм – связи доступа или дисплей – ни выбрал автор компилятора, необходима определенная работа по его обслуживанию в последовательности вызова. В случае дисплея пролог обновляет запись дисплея для своего уровня, а эпилог восстанавливает ее. Если процедура не вызывает более глубоко вложенных процедур, то этот шаг можно опустить. В случае связей доступа предварительная последовательность должна найти подходящую первую связь доступа для вызываемой стороны. Объем работы зависит

от разности лексических уровней вызывающей и вызываемой сторон. При условии что вызываемая сторона известна на этапе компиляции, обе схемы достаточно эффективны. Если же вызываемая сторона неизвестна (например, если это параметр-функция), то компилятору придется сгенерировать специальный код для выполнения необходимых шагов.

ИТОГИ РАЗДЕЛА

Соглашение о связях – это контракт между компилятором, операционной системой и оборудованием. Он описывает передачу управления между процедурами, сохранение состояния вызывающей стороны, создание состояния вызываемой стороны и правила передачи значений между ними.

Стандартные соглашения о связях позволяют собирать исполняемые программы из исходного кода, написанного разными людьми в разное время. Они гарантируют, что все процедуры будут работать правильно и безопасно. Те же самые соглашения позволяют приложениям вызывать системные и библиотечные функции. Хотя детали соглашений о связях зависят от системы, основные принципы похожи для большинства целевых машин, операционных систем и компиляторов.

ВОПРОСЫ

1. Какую роль соглашения о связях играют при построении программ, написанных на разных языках? Что должен знать компилятор, чтобы сгенерировать код для вызова процедуры, написанной на другом языке?
2. Если в месте вызова компилятор знает, что вызываемая сторона не содержит ни одного вызова процедуры, то какие шаги последовательности вызова он может опустить? Существуют ли в ЗА поля, которые вызываемой стороне не понадобятся?

6.6. Дополнительные темы

Компилятор должен организовать выделение памяти для определенных пользователем данных и для вспомогательных структур времени выполнения. Часто время жизни этих структур плохо сочетается с дисциплиной обслуживания стека «последним пришел, первым ушел». В таких случаях реализация языка выделяет место в куче – области памяти, предназначенной для таких объектов и управляемой функциями из библиотеки времени выполнения. Компилятор должен также организовать хранение других объектов, время жизни которых не связано с потоком управления, таких, например, как большинство объектов в JAVA-программе.

Мы предполагаем простой интерфейс с кучей, а именно функции `allocate(size)` и `free(address)`. Функция `allocate` принимает целый аргумент `size` и возвращает адрес блока памяти в куче, содержащего не менее `size` байт. Функция `free` принимает адрес предварительно выделенного блока памяти и возвращает его в пул свободной памяти. Критические проблемы, возникающие при проектировании алгоритмов явного управления кучей, – эффективность `allocate` и `free` и фрагментация пула свободной памяти на мелкие блоки.

В этом разделе в общих чертах описываются алгоритмы, участвующие в выделении и освобождении памяти в куче. В разделе 6.6.1 описывается алгоритм выделения «первый подходящий», после чего эта идея обобщается для разработки распределителей с несколькими пулами. В разделе 6.6.2 рассматриваются неявные схемы освобождения и возврата памяти в пул, устраняющие необходимость явно вызывать `free` для освобождения памяти в куче.

6.6.1. Явное управление кучей

В большинстве реализаций языков имеется система времени выполнения, предоставляющая вспомогательные функции коду, сгенерированному компилятором. В их число обычно входят функции для управления кучей. Поддержка управления кучей может быть языково-зависимой, как в интерпретаторе SCHEME или в виртуальной машине JAVA, или являться частью операционной системы, как в случае реализаций `malloc` и `free`, отвечающих стандарту POSIX.

Для реализации `allocate` и `free` предложено много разных алгоритмов. Большинство из них основано на общих стратегиях и предположениях. В этом разделе рассматривается простая стратегия *выделения первого подходящего*, которая обнажает много проблем, а затем показано, как подобная стратегия используется для реализации современного распределителя памяти.

Выделение первого подходящего блока

Основная цель распределителя типа «первый подходящий» – обеспечить скорость работы `allocate` и `free`. Скорость важнее эффективности использования памяти. Каждый блок содержит скрытое поле размера, обычно находящееся перед адресом, возвращенным `allocate`; см. рис. 6.7(a). Распределитель поддерживает список доступных блоков, называемый *списком свободных*. Блоки, находящиеся в списке свободных, имеют дополнительные поля, показанные на рис. 6.7(b). Каждый свободный блок хранит указатель на следующий блок в списке свободных (в последнем блоке он равен `null`) и указатель на сам блок в его последнем слове. Сразу после инициализации куча содержит один большой блок свободной памяти.

Для уменьшения фрагментации распределитель может задать минимальный размер блока, s. Блок расщепляется на два, только если размеры обоих результирующих блоков больше или равны s.

Вызов `allocate(`k`)` приводит к следующей последовательности событий. Функция `allocate` проходит по списку свободных, пока не найдет блок размера, большего или равного k плюс одно слово для поля `size`. Обозначим найденный блок b_i. Этот блок удаляется из списка свободных. Если b_i больше, чем необходимо, то `allocate` создает новый блок, помещая в него избыток памяти в конце b_i, обновляет поля b_i и нового блока, после чего помещает новый блок в список свободных. Наконец, `allocate` возвращает указатель на смещение внутри b_i.

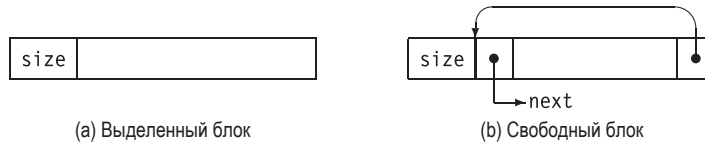


Рис. 6.7 Блоки в распределителе типа «первый подходящий»

Не найдя достаточно большого блока, `allocate` пытается расширить кучу. Если это получается, то она возвращает блок нужного размера из вновь выделенной части кучи. Если расширить кучу не удалось, то `allocate` сообщает об ошибке.

Чтобы освободить блок, программа вызывает `free`, передав ей адрес блока, b_j. Простейшая реализация `free` добавляет b_j в начало списка свободных и возвращается. Это быстро, но, к сожалению, такой распределитель со временем фрагментирует память на малые блоки.

Чтобы преодолеть этот недостаток, распределитель может использовать указатель в конце освобожденного блока для объединения соседних свободных блоков в один. Функция `free` загружает слово, предшествующее полю размера b_j, которое является концевым указателем блока, непосредственно предшествующего b_j в памяти. Если это слово содержит допустимый указатель, который указывает на заголовок собственного блока (того, для которого указатель `next`, следующий за полем размера, указывает на начало b_j), значит, и b_j, и его предшественник свободны. Функция `free` может объединить их, увеличив значение в поле размера предшественника и поместив правильный указатель в конец b_j. Если предшествующий блок занят, то `free` добавляет b_j в начало списка свободных.

Чтобы эта схема работала, `allocate` и `free` должны поддерживать концевые указатели. Всякий раз, как `free` обрабатывает блок, она должна обновить этот указатель, записав в него адрес начала блока. Функция `allocate` должна сделать недействительным либо указатель `next`, либо концевой указатель, чтобы помешать `free` объединить освобожденный блок с занятым, в котором эти поля не были перезаписаны.

Функция `free` также может попытаться объединить b_j со следующим за ним блоком, b_k. Чтобы найти начало b_k, она может воспользоваться полем размера b_j. А поле размера и концевой указатель b_k можно использовать, чтобы понять, свободен ли b_k. Если b_k свободен, то `free` может объединить оба блока, выполнив следующие действия: удалить b_k из списка свободных, добавить b_j в список свободных и соответственно обновить поле размера и концевой указатель b_j. Чтобы обновление списка свободных было эффективным, его следует сделать дважды связанным. Накладные расходы на организацию дважды связанного списка минимальны.

В том виде, как описана, схема объединения полагается на тот факт, что связь между указателями в свободном блоке никогда не встретится в занятом блоке. Хотя крайне маловероятно, что распределитель примет занятый блок за свободный, такое все же может случиться. Чтобы застраховаться от этого, разработчик может потребовать, чтобы концевой указатель хранился

и в свободном, и в занятом блоке. Если `allocate` сбрасывает этот указатель в нуль, а `free` записывает в него адрес начала блока, то распределитель никогда не ошибется в идентификации занятого блока.

Распределение памяти на основе арены

Внутри самого компилятора разработчик может счесть выгодным использование специализированного распределителя. Работу компилятора можно разбить на отдельные фазы, поэтому схема распределения на основе арены выглядит привлекательно.

В этом случае программа в начале фазы создает арену – пул непрерывной памяти в куче. Арена используется для хранения выделенных объектов, связанных общностью использования. Обращения к арене за объектами обслуживаются по принципу стека. При выделении указатель на верхнюю границу арены увеличивается и возвращается указатель на вновь выделенный блок. Способа освободить отдельный объект не предусмотрено; они освобождаются все разом вместе с содержащей их ареной.

Распределитель на основе арены – это компромисс между традиционными распределителями и распределителями со сборкой мусора. Обращения к `allocate` можно сделать очень дешевыми (как в распределителе с несколькими пулами). Никаких вызовов для освобождения памяти не нужно; программа освобождает всю арену за один вызов, когда заканчивается фаза, для которой арена была создана.

Было предложено много вариаций на тему алгоритма «первый подходящий». В них реализованы различные компромиссы между стоимостью `allocate`, стоимостью `free`, степенью фрагментации после длинной серии выделений и количеством впустую потраченной памяти из-за возврата блоков, больших запрошенного.

Распределители с несколькими пулами

Распределители с несколькими пулами основаны на алгоритме «первый подходящий», но упрощают его благодаря некоторым наблюдениям над поведением программ. По мере увеличения объема памяти в начале 1980-х годов стало оправданным потратить впустую сколько-то памяти, если при этом удавалось добиться более быстрого выделения. В то же время исследования поведения программ показали, что в реальных программах память часто выделяется блоками нескольких типичных размеров и гораздо реже – блоками большого или необычного размера.

В современных распределителях используются отдельные пулы памяти нескольких употребительных размеров. Как правило, размеры являются степенями двойки, они начинаются с небольшой величины (скажем, 32 или 64 байта) и растут до размера страницы виртуальной памяти (обычно 4096 или 8192 байта). В каждом пуле имеются блоки только одного размера, поэтому `allocate` может вернуть первый блок из подходящего списка свободных, а `free` просто добавляет блок в начало соответствующего списка свободных. Если запрошенный размер превышает страницу, то используется отдельный

распределитель типа «первый подходящий». Распределители, основанные на этой идее, работают быстро. Особенно они хороши для выделения ЗА из кучи.

Эти изменения упрощают и `allocate`, и `free`. Функция `allocate` должна проверить, что список свободных не пуст, а если пуст, то добавить в него новую страницу блоков нужного размера. Функция `free` вставляет освобожденный блок в начало списка свободных для его размера. Реализация могла бы определить размер освобожденного блока, сравнив его адрес с границами сегментов памяти, выделенной для каждого пула. Альтернативная схема подразумевает использование поля размера, как и раньше, а если распределитель помещает всю принадлежащую странице память в один пул, то размер всех блоков на этой странице можно хранить в специально отведенном на странице слове.

Помощь в отладке

Многолетний опыт показывает, что программисты склонны забывать об освобождении всей памяти, которую выделили. Кроме того, они нередко освобождают объекты, на которые еще хранятся указатели. Неявное освобождение решает обе проблемы.

Хорошо известно, что программы, в которых память выделяется и освобождается явно, отлаживать нелегко. Бывает, что программисту трудно решить, когда освобождать объекты, выделенные в куче. Если распределитель может быстро отличить свободный блок от занятого, то средства управления кучей могут оказать программисту определенную помощь при отладке.

Например, чтобы объединить соседние свободные блоки, распределителю необходим указатель из конца блока на его начало. Если в занятом блоке этот указатель делается недействительным, то функция освобождения может проверить это поле и сообщить об ошибке, когда программа пытается освободить свободный блок или блок по неверному адресу, т. е. адресу, не совпадающему с началом какого-либо выделенного блока.

Ценой умеренных дополнительных затрат средства управления памятью могут предоставить и дополнительную помощь. Связав все выделенные блоки, распределитель может создать среду для работы инструментов отладки выделения памяти. Программа создания моментального снимка может обойти список выделенных блоков. Пометка блоков местом вызова, где они были выделены, позволяет этой программе обнаруживать утечки памяти. Если еще и снабдить их временной меткой, то программист получит подробную информацию об использовании памяти. Такого рода инструменты могут оказать неоценимую помощь в поиске выделенных, но не освобожденных блоков.

6.6.2. Неявное освобождение

Многие языки программирования поддерживают неявное освобождение объектов в куче. Система времени выполнения автоматически освобождает объ-

Сборка мусора
неявное освобождение объектов, выделенных в куче.

Мертвый объект
Объект называется мертвым, если исполняемая программа не может добраться до него.

екты, которые больше не используются. Чтобы произвести неявное освобождение, или *сборку мусора*, компилятор и система времени выполнения должны включать механизм определения того, что объект больше не представляет интереса, т. е. является *мертвым*, а также механизм возвращения в оборот памяти, занятой мертвыми объектами.

Действия, направленные на возврат памяти в оборот, могут выполняться оперативно – для каждого предложения в отдельности – или пакетно по запросу, когда свободная память в пуле заканчивается. *Подсчет ссылок* – классическая техника оперативного возврата памяти. *Сборка с пометкой и очисткой* – классический подход к выполнению пакетного возврата.

Подсчет ссылок

Этот метод подразумевает добавление счетчика к каждому выделенному в куче объекту. Счетчик отслеживает количество активных указателей, или ссылок на объект. Создавая объект, распределитель устанавливает счетчик ссылок равным 1. Каждое присваивание указательной переменной изменяет два счетчика ссылок. Счетчик ссылок на объект до присваивания уменьшается, а счетчик ссылок на объект после присваивания увеличивается. Если счетчик ссылок обращается в нуль, значит, никаких указателей, позволяющих добраться до объекта, не осталось, поэтому система может безопасно удалить объект.

Освобождение объекта может, в свою очередь, сделать недействительными указатели на другие объекты. Программа должна уменьшить счетчики ссылок на эти объекты. Например, когда становится недействительным последний указатель на корень дерева, должно быть освобождено все дерево. Когда счетчик ссылок на корневой узел обращается в нуль, он освобождается, и счетчики ссылок на его сыновние узлы уменьшаются. Это, в свою очередь, должно привести к освобождению сыновних узлов и уменьшению счетчиков ссылок на их сыновние узлы. Процесс продолжается, пока не будет освобождено все дерево.

Наличие указателей в выделенных объектах создает несколько проблем для схем с подсчетом ссылок.

1. Программе необходимо как-то отличать указатель от других данных объекта. Система может завести в каждом объекте поле заголовка, в котором будет хранить дополнительную информацию, например битовую карту, в которой указано, какие слова объекта являются указателями. Она может ограничить диапазон указателей, отведя несколько старших битов для «пометки» указателя. Она может хранить детальную информацию о типе во время выполнения и использовать ее для опознания указателей. Отметим, что пакетные сборщики мусора сталкиваются с той же проблемой и применяют те же самые решения.

2. Уменьшение до нуля счетчика ссылок на такой объект, как корень дерева, может повлечь за собой большой объем работы. Если на время освобождения налагаются внешние ограничения, как, например, в системах реального времени, то система времени выполнения может принять схему, ограничивающую количество объектов, освобождаемых при каждом присваивании указателю. Она может хранить очередь объектов, подлежащих освобождению, и ограничивать количество объектов, освобождаемых при каждом изменении счетчика ссылок. При таком подходе затраты распределяются между большим набором операций, а работа, выполняемая в ответ на одно присваивание, ограничивается.
3. Программа может создать в куче циклическую структуру. Счетчики ссылок в циклической структуре данных никогда не обращаются в нуль. Когда последний внешний указатель исчезает, весь цикл становится недостижимым; занятую им память нельзя освободить с помощью механизма подсчета ссылок. Чтобы освободить циклическую структуру, *программист* должен разорвать цикл до того, как будет уничтожен последний указатель на него. Многие выделяемые в куче объекты, например строки переменной длины и ЗА, не могут образовывать такие циклы.

Подсчет ссылок требует дополнительной работы при каждом присваивании указателю. Объем работы для конкретного присваивания можно ограничить; в любой хорошо спроектированной схеме общую стоимость можно ограничить произведением некоторой константы на общее число присваиваний указателю плюс число выделенных объектов. Сторонники подсчета ссылок аргументируют этот подход тем, что накладные расходы достаточно малы, при этом паттерн повторного использования обеспечивает хорошую локальность программ. Противники же говорят, что в реальных программах количество присваиваний указателям больше, чем количество выделений памяти, поэтому сборка мусора дает эквивалентную функциональность, выполняя меньше работы.

Пакетные сборщики мусора

Если сборщик не может найти свободной памяти, то он должен запросить дополнительную память у системы. Если и там ее нет, то выделение памяти завершается неудачно.

Пакетные сборщики мусора приступают к освобождению памяти, только когда пул свободной памяти исчерпан. Если распределитель не может найти нужное количество памяти, то он вызывает сборщик мусора. Сборщик приостанавливает выполнение программы, просматривает пул выделенной памяти в поисках неиспользуемых объектов и возвращает занятую ими память в пул. По завершении работы сборщика пул свободной памяти должен быть не пуст. Распределитель может закончить свою задачу и вернуть выделенный объект вызывающей стороне. (Как и в случае подсчета ссылок,

существуют схемы, которые выполняют сборку мусора постепенно, чтобы разнести затраты на более длительный период времени.)

Логически работа сборщика мусора состоит из двух этапов. На первом он находит множество объектов, которых можно достичь по указателям, хранящимся в переменных, и из временных объектов, сгенерированных компилятором. Сборщик осторожно предполагает, что всякий достижимый таким образом объект жив, а все остальные мертвы. На втором этапе сборщик освобождает память, занятую мертвыми объектами, и возвращает ее в пул. Для этого чаще всего используется один из двух методов: *сборка с пометкой и очисткой* и *копирующие* сборщики. Различаются они реализацией второго этапа сборки – возврата в пул.

Идентификация живых данных

Распределители со сборкой мусора находят живые объекты с помощью алгоритма пометки. Сборщику необходим один бит для каждого объекта в куче, называемый *битом пометки*. Концептуально можно считать, что биты пометки хранятся в самих объектах. Сразу после создания бит пометки должен находиться в состоянии «не помечен», или «очищен».

На первом шаге этапа пометки строится рабочий список, который содержит все указатели, хранящиеся в регистрах и в переменных, доступных активным процедурам. На втором шаге алгоритм проходит по списку этих указателей и помечает каждый объект, достижимый из любого из них.

На рис. 6.8 показан высокоуровневый набросок простого алгоритма пометки. Он останавливается, потому что куча конечна, а пометка гарантирует, что каждый указатель попадает в *Worklist* не более одного раза. Стоимость пометки пропорциональна количеству просмотренных указателей.

```
Очистить все пометки
Worklist ← { значения указателей в записях активации и регистрах }

while (Worklist ≠ ∅) do
   удалить p из Worklist
   if (p → object не помечен) then
      пометить p → object
      добавить указатели из p → object в Worklist
```

Рис. 6.8 Простой алгоритм пометки

Алгоритм пометки может быть точным или консервативным. Разница в том, как алгоритм определяет, что конкретное значение является указателем, в последней строке цикла *while*.

- В случае точного этапа пометки компилятор и система времени выполнения знают тип и внутреннюю структуру каждого объекта. Эта информация может быть записана в заголовках объектов или известна неявно от системы типов. В любом случае на этапе отметки просматриваются только настоящие указатели.

- В случае консервативного этапа пометки компилятор и система времени выполнения не всегда уверены в типе и внутренней структуре объектов. Поэтому, помечая объект, система рассматривает все поля, которые потенциально могут быть указателями. Если значение может быть указателем, то считается, что так оно и есть. Любое значение, которое не представляет выровненный на границу слова адрес, следует исключить, равно как и значения, указывающие не на кучу.

У консервативных сборщиков есть ограничения. Они не возвращают в кучу память объектов, которые точный сборщик мог бы найти. Тем не менее консервативные сборщики были задним числом успешно внедрены в реализации таких языков, как C, который вообще-то сборку мусора не поддерживает.

Когда алгоритм пометки останавливается, все непомеченные объекты должны быть недостижимы из программы. Поэтому на втором этапе сборщик может рассматривать такие объекты как мертвые. Некоторые объекты, помеченные живыми, также могут быть мертвы. Однако сборщик оставляет им возможность воскреснуть, потому что не может доказать, что они мертвы. Когда на втором этапе алгоритм обходит кучу для сборки неиспользуемой памяти, он может очистить биты пометки.

Сборщики с пометкой и очисткой

Сборщики с пометкой и очисткой освобождают память и возвращают ее в пул в ходе линейного прохода по куче. Сборщик добавляет каждый непомеченный объект в список свободных (или в один из списков свободных), где распределитель сможет найти и вновь использовать его. В случае одного списка свободных применяются те же способы объединения блоков, что и в распределителе типа «первый подходящий». Чтобы уплотнить кучу, сборщик может сдвигать живые объекты вниз прямо в процессе очистки или на отдельном проходе уплотнения после очистки.

Копирующие сборщики

Копирующие сборщики делят память на два пула: *старый* и *новый*. Распределитель всегда работает со старым пулом. Простейший тип копирующих сборщиков называется «остановка и копирование». Когда выделение памяти завершается неудачно, сборщик типа «остановка и копирование» копирует все живые данные из старого пула в новый и меняет старый и новый пулы местами. В процессе копирования живые данные уплотняются; после завершения сборки вся свободная память занимает один непрерывный блок.

Копирование можно производить на отдельном проходе, как в алгоритме пометки и очистки, или оперативно, по мере обнаружения живых данных. В оперативной схеме объекты в старом пуле помечаются перед копированием, чтобы не скопировать один и тот же объект несколько раз.

Поколенческие сборщики – важное семейство копирующих сборщиков. Они основаны на том наблюдении, что если объект пережил одну сборку, то он,

скорее всего, переживет и последующие. В поколенческом сборщике при копировании используются два «новых» пула, которые иногда называют *яслями*. Кроме того, поколенческий сборщик поддерживает один или несколько более старых, или *стабильных*, пулов. Когда ясли становятся слишком заполненными, распределитель собирает и копирует живые данные из яслей в стабильный пул. При последующих сборках в яслях просматриваются только вновь выделенные объекты. Поколенческие сборщики различаются количеством стабильных пулов и частотой сборки в них.

Сравнение методов

Сборка мусора снимает с программиста заботу о том, когда освобождать память, и позволяет ему не думать о неизбежных утечках памяти, связанных с явным ее освобождением. Хотя у отдельных схем есть сильные и слабые места, практические преимущества неявного освобождения в большинстве приложений перевешивают их недостатки.

Подсчет ссылок распределяет стоимость освобождения памяти более равномерно по времени выполнения программы, чем пакетная сборка мусора. Однако он увеличивает стоимость каждого присваивания, в котором участвует указатель на память, выделенную из кучи, – даже если программа вообще не испытывает недостатка в памяти. С другой стороны, пакетные сборщики не влекут никаких накладных расходов, пока у распределителя не кончится память. Однако в этот момент на программу обрушивается вся тяжесть сборки мусора. Кроме того, любое выделение памяти может спровоцировать запуск сборщика мусора.

Сборщики с пометкой и очисткой просматривают всю кучу, тогда как копирующие сборщики интересуются только живыми данными. Копирующие сборщики перемещают каждый живой объект, а сборщики с пометкой и очисткой оставляют их на месте. Кто выигрывает от этого компромисса, зависит от поведения приложения и фактической стоимости различных ссылок на память.

PYTHON подсчитывает ссылки на объекты, но вызывает пакетный сборщик мусора, когда место в куче заканчивается. Пакетный сборщик может вернуть память, «потерянную» из-за циклических структур.

Реализации подсчета ссылок и консервативные пакетные сборщики мусора сталкиваются с проблемами при распознавании циклических структур, потому что не могут отличить ссылки внутри цикла от внешних по отношению к циклу ссылок. Сборщики с пометкой и очисткой начинают работу с внешнего множества указателей, поэтому могут понять, что мертвая циклическая структура недостижима. Копирующие сборщики, начинающие работу с того же самого множества указателей, просто не копируют объекты, образующие цикл.

Копирующие сборщики уплотняют память – это естественная часть процесса. Сборщик может либо обновить все хранящиеся указатели, либо потребовать, чтобы для каждого доступа к объекту использовалась таблица

косвенной адресации. Точный сборщик с пометкой и очисткой тоже может уплотнять память. Сборщик мог бы перемещать освобожденные объекты из одного конца памяти в другой. И на этот раз сборщик может либо переписывать существующие указатели, либо использовать таблицу косвенной адресации.

В общем случае разработчик может написать алгоритмы пометки с очисткой и копирования настолько хорошо, что они будут приемлемы для большинства приложений. Для тех же приложений, которые не могут мириться с непредсказуемыми накладными расходами, например контроллеров реального времени, система времени выполнения может разбить процесс на части, как в амортизированной схеме с подсчетом ссылок. Такие сборщики мусора называются *сборщиками реального времени*.

6.7. Итоги и перспективы

Главные побудительные причины, заставившие выйти за пределы языка ассемблера, – построить более абстрактную модель языка программирования, повысить продуктивность программистов и сделать программы понятнее. Компилятор должен быть готов транслировать каждую абстракцию исходного языка. В этой главе мы рассмотрели методы реализации абстракций, связанных с вызовом процедур.

Процедурное программирование было изобретено на заре истории программирования. Некоторые из первых процедур были написаны для отладки ранних компьютеров. Эти заранее подготовленные процедуры позволяли программистам понять, в каком состоянии находится неправильно работающая программа. Без таких процедур даже такие простые задачи, как просмотр содержимого переменной, трассировка стека вызовов или печать содержимого памяти, можно было бы решить лишь при условии, что программист способен ввести длинную последовательность машинного кода без единой ошибки.

Включение лексических областей видимости в такие языки, как ALGOL 60, на многие десятилетия вперед определило дизайн языков программирования. Большинство современных языков унаследовали часть философии ALGOL, связанную с именованием и адресуемостью. Методы, придуманные, чтобы сократить затраты на организацию лексических областей видимости во время выполнения, в частности связи доступа и дисплеи, находят широкое применение и поныне.

Объектно-ориентированные языки (ООЯ) позаимствовали концепции видимости у ALGOL-подобных языков и переориентировали их на данные. Обычно такие языки поддерживают как иерархию лексических областей видимости, так и иерархию наследования. Компилятор ООЯ использует структуры времени компиляции и времени выполнения, придуманные для лексической видимости, чтобы смоделировать и реализовать дисциплину именования, налагаемую иерархией наследования.

С появлением новых парадигм программирования появляются и новые абстракции, требующие вдумчивого осмысления и реализации. Изучив успешные методы прошлого и разобравшись в ограничениях и затратах имеющихся реализаций, авторы компиляторов смогут разработать стратегии, уменьшающие плату за использование более высоких уровней абстракции на этапе выполнения.

Замечания к этой главе

Большая часть материала этой главы взята из опыта, накопленного сообществом разработчиков компиляторов. Лучший способ узнать больше о структурах пространства имен в различных языках – обратиться к определению самого языка. Эти документы должны обязательно присутствовать в библиотеке автора компилятора.

Процедуры появились в самых первых языках высокого уровня, т. е. языках, более абстрактных, чем язык ассемблера. В FORTRAN [28] и ALGOL 60 [282] уже были процедуры, обладающие большинством свойств, которые мы встречаем в современных языках. В ранних реализациях ALGOL 60 были развиты многие ключевые идеи, касающиеся соглашений о связях; например, Ingerman впервые описал переходники для передачи параметров по имени в 1961 году [213]. Объектно-ориентированные языки появились в конце 1960-х годов: сначала SIMULA-67 [287], а за ним SMALLTALK 72 [242]. В различных SMALLTALK-системах использовались кеши методов для уменьшения стоимости динамической диспетчеризации [137, 242].

В ALGOL 60 впервые появились лексические области видимости, оставшиеся до сего дня. В ранних компиляторах ALGOL было введено большинство вспомогательных механизмов, описанных в этой главе: записи активации, связи доступа, дисплеи и методы передачи параметров. Большая часть идей, представленных в разделах 6.3–6.5, присутствовала в одной из этих ранних систем [303]. Довольно быстро были предложены оптимизации соглашения о связях, например включение памяти для блочной области видимости в запись активации для процедуры, содержащей этот блок. Мэртаг систематически подошел к объединению записей активации, соединив ЗА процедур, которые часто вызываются вместе [281]. В соглашении о связях для IBM 370 было проведено различие между листовыми и прочими процедурами; для листовых процедур области сохранения регистров не создавались.

Классическая работа по схемам выделения памяти – книга Кнута «Искусство программирования для ЭВМ» [240, § 2.5]. Современные распределители с несколькими пулами появились в начале 1980-х годов. Метод подсчета ссылок восходит к началу 1960-х годов и использовался во многих системах [102, 136]. Шорр и Уайт описывают ранний алгоритм пометки [319]. Коэн, а позднее Уилсон составили обширные обзоры литературы по сборке мусора [99, 362]. Консервативные сборщики мусора были введены в работах Боэма и Вайзера [48, 50, 131]. Копирующие сборщики были разработаны в ответ на

системы виртуальной памяти [86, 154]; они естественно привели к поколенческим сборщикам, широко распространенным в наши дни [255, 349]. Хэнсон ввел понятие выделения памяти на основе арены [189].

Упражнения

К разделу 6.2

1. Покажите дерево вызовов и историю выполнения для следующей программы на C:

```
int Sub(int i, int j) {
    return i - j;
}
int Mul(int i, int j) {
    return i * j;
}
int Delta(int a, int b, int c) {
    return Sub(Mul(b,b), Mul(Mul(4,a),c));
}
void main() {
    int a, b, c, delta;
    scanf("%d %d %d", &a, &b, &c);
    delta = Delta(a, b, c);
    if (delta == 0)
        puts("Two equal roots");
    else if (delta > 0)
        puts("Two different roots");
    else
        puts("No root");
}
```

К разделу 6.3

2. Некоторые языки программирования позволяют использовать функции при инициализации локальных, но не глобальных переменных.
   a. Есть ли разумное объяснение этой, на первый взгляд, странной детали определения языка?
   b. Какие механизмы понадобились бы, чтобы разрешить инициализацию глобальной переменной результатом вызова функции?

3. Автор компилятора может оптимизировать выделение памяти для ЗА несколькими способами. Например, компилятор мог бы:
   a. Выделять память для ЗА листовых процедур статически.
   b. Объединять ЗА процедур, которые всегда вызываются вместе (если вызывается α, она всегда вызывает β.)

   Для каждой схемы ответьте на следующие вопросы:
   a. Вызовы какого вида могут выиграть от такой оптимизации?
   b. Как вы думаете, насколько часто такие вызовы встречаются в реальных программах?

c. Какое влияние оказывает оптимизация на использование памяти во время выполнения?

4. Рассмотрите следующие определения класса `Dog` и класса `Retriever`. Нарисуйте структуры, которые понадобятся компилятору для поддержки объектов типа `Retriever`.

```
class Dog {
    private int Height;
    private int Weight;
    public int Sheds;

    public int GetHeight();
    public int GetWeight();
    public boolean PlaysFetch();
}

class Retriever extends Dog {
    private boolean Calm;
    public boolean Disposition();
    public boolean PlaysFetch();
}
```

Метод `PlaysFetch` класса `Dog` просто возвращает `false`, а тот же метод класса `Retriever` возвращает `true`. Метод `Disposition` класса `Retriever` справляется с булевым членом `Calm`.

5. Рассмотрите определения `Dog` и `Retriever` из предыдущего вопроса.

a. Если программа приводит объект класса `Retriever` к объекту класса `Dog`, т. е. имеет место *повышающее приведение*, то какие изменения в поведении будут наблюдаться? Как работает это повышающее приведение на операционном уровне?

b. Если программа приводит объект класса `Dog` к объекту класса `Retriever`, т. е. имеет место *понижающее приведение*, то каких изменений в поведении вы ожидаете? Какие проблемы могут возникнуть в результате понижающего приведения?

c. Как, по вашему мнению, язык мог бы разумно обработать попытки понижающего приведения?

6. В качестве альтернативы кешу глобальных методов реализация могла бы поддерживать встроенный кеш в каждом месте вызова – кеш с одной записью для хранения последнего вызванного в этом месте класса и метода.

Разработайте псевдокод использования и обслуживания такого встроенного кеша методов. Объясните, как инициализируются встроенные кеши методов и как следует модифицировать общую процедуру поиска метода, чтобы она поддерживала встроенные кеши.

К разделу 6.4

7. Рассмотрите следующую программу, написанную на PASCAL-подобном псевдокоде:

```
procedure main;
    var a : array[1...3] of integer;
        i : int;
    procedure p2(e : integer);
        begin
            e := e + 3;
            a[i] := 5;
            i := 2;
            e := e + 4;
            end;
    begin
        a := [1, 10, 77];
        i := 1;
        p2(a[i]);
        for i := 1 to 3 do
            print(a[i]);
        end.
```

Смоделируйте ее выполнение в случае передачи параметров по значению, по ссылке и по имени. Покажите результаты предложений `print` в каждом случае.

8. При передаче параметров по ссылке могут возникнуть ситуации, когда два разных параметра ссылаются на одно и то же место в памяти. Рассмотрите следующую процедуру на PASCAL, параметры которой передаются по ссылке:

```
procedure mystery(var x, y : integer);
    begin
        x := x + y;
        y := x - y;
        x := x - y;
    end;
```

В предположении, что при выполнении арифметических операций не возникает ни переполнения, ни потери значимости:

a. Какой результат породит `mystery` при вызове с двумя разными переменными, `a` и `b`?

b. Какого результата вы ожидаете, если при вызове `mystery` в обоих параметрах передается одна и та же переменная? А каков фактический результат в этом случае?

9. Рассмотрите PASCAL-программу на рис. 6.9(a). Предположим, что в реализации используются ЗА, показанные на рис. 6.9(b). (Некоторые поля для простоты опущены.) Память для ЗА выделяется в стеке, и стек растет в направлении начала страницы. УЗА – единственный указатель на ЗА, поэтому связи доступа – это предыдущие значения УЗА. Наконец, на рис. 6.9(c) показана ЗА в начале вычисления.

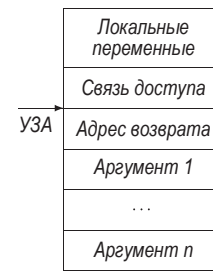
a. Для программы на рис. 6.9(a) нарисуйте набор ее ЗА прямо перед возвратом из функции F1. Используйте номера строк для задания адресов возврата. Связи доступа изображайте ориентированными дугами. По-

кажите место, отведенное для локальных переменных и параметров. Пометьте каждую ЗА именем ее процедуры.

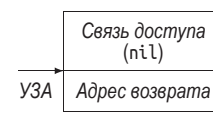
b. Покажите то же состояние в предположении, что реализация пользуется глобальным дисплеем, а не связями доступа.

```
1  program main(input, output);
2      procedure P1(function g(b: integer): integer);
3          var a: integer;
4          begin
5              a := 3;
6              writeln(g(2))
7              end;
8      procedure P2;
9          var a: integer;
10         function F1(b: integer): integer;
11             begin
12                 F1 := a + b
13                 end;
14         procedure P3;
15             var a: integer;
16             begin
17                 a := 7;
18                 P1(F1)
19                 end;
20         begin
21             a := 0;
22             P3
23             end;
24     begin
25         P2
26         end.
```

(a) Пример PASCAL-программы



(b) Структура записи активации



(c) Начальная запись активации

Рис. 6.9 Программа для упражнения 9

К разделу 6.5

10. Для каждого из следующих действий ответьте, принадлежит ли оно предварительной последовательности, прологу, эпилогу или заключительной последовательности. Кратко обоснуйте свои ответы.

```
procedure main
    integer a, b, c
    b = a + c;
    c = f1(a,b);
    call print(c);
    end;
```

```
procedure f1(integer x, y)
    integer v;
    v = x * y;
    call print(v);
    call f2(v);
    return -x;
    end;
```

```
procedure f2(integer q)
    integer k, r;
    ...
    k = q / r;
    end;
```

Рис. 6.10 Код к упражнению 12

a. Сохранение регистров, сохраняемых вызывающей стороной.
b. Восстановление регистров, сохраняемых вызываемой стороной.
c. Сохранение возвращенного значения.
d. Сохранение адреса возврата.
e. Использование адреса возврата.

11. При проектировании соглашения о связях некоторые действия могут перемещаться между предварительной последовательностью и прологом или между заключительной последовательностью и эпилогом.
   a. Почему может иметь смысл перенести что-то из предварительной последовательности в пролог? А из заключительной последовательности в эпилог?
   b. Какие действия вы могли бы захотеть перенести из одной последовательности в другую?

12. Предположим, что компилятор умеет анализировать код и устанавливать такие факты, как «*начиная с этой точки переменная v в этой процедуре больше не используется*» или «*переменная v в следующий раз используется в строке 11 этой процедуры*», и что для всех трех процедур на рис. 6.10 компилятор хранит все локальные переменные в регистрах.
   a. Переменная `x` в процедуре `f1` переживает два вызова процедуры. Чтобы код выполнялся максимально быстро, должен ли компилятор хранить `x` в регистре, сохраняемом вызывающей стороной, или в регистре, сохраняемом вызываемой стороной? Обоснуйте свой ответ.
   b. Рассмотрите переменные `a` и `c` в процедуре `main`. Должен ли компилятор хранить их в регистрах, сохраняемых вызывающей стороной, или в регистрах, сохраняемых вызываемой стороной, если требуется обеспечить максимальную скорость работы откомпилированного кода? Обоснуйте свой ответ.

Глава 7

Форма кода

АННОТАЦИЯ

Для трансляции программы компилятор должен преобразовать каждое предложение исходного языка в последовательность одной или нескольких команд языка целевой машины. Компилятор должен выбирать из многих возможных способов реализации каждой конструкции. Его выбор оказывает прямое и сильное влияние на качество конечного кода.

В этой главе мы рассмотрим некоторые стратегии реализации, которые компилятор может применять для типичных конструкций языков программирования, с упором на отображение конструкций исходного уровня в набор команд целевой машины. Мы обсудим вычисление выражений, методы доступа к переменным и агрегатным структурам данных, управляющие конструкции и вызовы процедур.

КЛЮЧЕВЫЕ СЛОВА

Генерирование кода, качество кода, управляющие конструкции, вычисление выражений

7.1. Введение

Транслируя код приложения в исполняемую форму, компилятор вынужден принимать мириады решений по поводу конкретных деталей организации вычислений и размещения данных. Такие решения часто влияют на производительность получающегося кода. В своих решениях компилятор руководствуется информацией, которую постепенно собирает в процессе трансляции. Собранную информацию компилятор должен закодировать в форме ПП и вспомогательных структур данных. Часть информации кодируется непосредственно и прозрачно. Например, если *x* объявлена как 64-разрядное целое, то компилятор запоминает этот факт в соответствующей таблице символов.

Другие факты кодируются неявно. Например, если компилятор понимает, что *y* – локальная скалярная переменная, и решает, что она однозначна, то он

должен закодировать это решение, сохраняя y в виртуальном регистре на протяжении всего времени ее жизни. Такой способ кодирования информации о неоднозначности делает этот факт очевидным остальным частям компилятора и исключает необходимость заново выводить эту информацию на последующих проходах. Подобный вид неявного кодирования знания в ПП программы мы называем *формой кода*. Автор компилятора принимает явное решение сформировать ПП определенным образом, чтобы закодировать информацию и сделать ее доступной на последующих проходах.

Установить неоднозначность бывает трудно. Некоторые ситуации, например взятие адреса скалярной переменной, очевидны. Другие, как, скажем, выполнение арифметических действий над указателями в программе на C, могут потребовать сложного анализа.

Концептуальная дорожная карта

Трансляция исходного кода в машинный – одно из основных действий компилятора. У компилятора должен быть план для каждой конструкции исходного языка. Многие проблемы, возникающие на этапе генерирования ПП в фазе анализа, возникают также при генерировании ассемблерного кода для реального процессора в фазе синтеза. Целевой процессор может оказаться более крепким орешком из-за конечности ресурсов и специфических особенностей, но принципы те же самые.

В этой главе мы будем говорить о способах реализации конструкций исходного языка в создаваемом компилятором ПП. Во многих случаях конкретные детали реализации влияют на способность компилятора анализировать и улучшать код. Концепция формы кода инкапсулирует все решения, большие и малые, принимаемые автором компилятора по части того, как представлять вычисление в ПП и ассемблерном коде. Пристальное внимание к форме кода может упростить процесс анализа и улучшения кода и позволит компилятору порождать более качественный конечный код.

Несколько слов о времени

На этапе проектирования автор компилятора должен решить, как в ПП программы будут выражены конкретные конструкции исходного языка. Эти решения напрямую влияют на эффективность конечного откомпилированного кода. В данной главе мы рассмотрим ряд конструкций исходного языка и при необходимости остановимся на некоторых вариантах, доступных автору компилятора.

Решения, касающиеся формы кода, принятые автором компилятора на этапе проектирования, могут определить способ выполнения трансляции компилятором. Чтобы оценить варианты, автору компилятора необходимо рассмотреть их возможное влияние на последующие фазы и в конечном итоге на производительность. Поэтому в этой главе мы будем много говорить о влиянии решений о форме ПП, принимаемых во время проектирования, на поведение во время компиляции и выполнения.

Краткий обзор

Автор компилятора должен уделить внимание форме кода, чтобы на последующих проходах компилятор мог порождать высококачественный код. На практике у компилятора имеется несколько вариантов реализации многих языковых конструкций. В них используются различные операции и различные подходы. Одни реализации быстрее прочих, другие занимают меньше памяти, третьим необходимо меньше регистров, четвертые потребляют меньше энергии во время работы. Мы считаем эти различия относящимися к форме кода.

От формы кода сильно зависит как поведение откомпилированного кода, так и способность оптимизатора и фазы синтеза улучшать код. Рассмотрим, к примеру, трансляцию предложения `switch` языка C, которое выбирает один из 256 случаев в зависимости от однобайтовой литеры. Компилятор мог бы реализовать предложение `switch` с помощью последовательности предложений `if-then-else`. В зависимости от организации проверок результаты могут быть различными. Если сначала производится сравнение с нулем, затем с единицей и т. д., то этот подход вырождается в линейный поиск на множестве ключей. Если проверяемые литеры распределены равномерно, то в среднем будет проверена половина ключей – довольно дорогой способ реализовать `switch`.

С другой стороны, компилятор мог бы реализовать `switch` с помощью двоичного поиска на множестве ключей. При такой схеме пришлось бы проверить в среднем $log_2(|keys|)$ ключей – гораздо лучше. Решив пожертвовать памятью ради скорости, компилятор мог бы построить таблицу 256 меток и в процессе интерпретации литеры загрузить из таблицы соответствующую метку и перейти на нее – в этом случае накладные расходы в расчете на одну литеру линейны.

Эти три схемы настолько сильно различаются, что оптимизатор вряд ли сможет преобразовать одну в другую. Решение, принятое в процессе трансляции, определит конечную форму кода для предложения `switch`.

Всё это правильные реализации предложения `switch`. Какая из них лучше, зависит от многих факторов. В частности, важную роль играет число ветвей `case` и их относительные частоты. Даже когда компилятор не может точно установить эти факты, он все равно должен выбрать какую-то стратегию для каждого предложения `switch`. Различия между возможными реализациями и выбор компилятора – все это вопросы формы кода.

В качестве другого примера рассмотрим простое выражение `x + y + z`, где `x`, `y` и `z` – целые числа. На рис. 7.1 показано несколько способов реализации этого выражения. На уровне исходного кода мы можем рассматривать эту операцию как тернарное сложение, показанное слева. Однако при преобразовании этой идеализированной операции в последовательность

На рисунке предполагается, что все три переменные хранятся в регистрах и что исходный язык не определяет порядок вычисления этого выражения.

бинарных сложений начинает играть роль порядок вычислений. Версии справа демонстрируют три возможных порядка – в виде трехадресного кода и абстрактного синтаксического дерева. Если сложение коммутативно и ассоциативно (для целых так оно и есть), то все три порядка эквивалентны, и компилятор должен выбрать один из них.

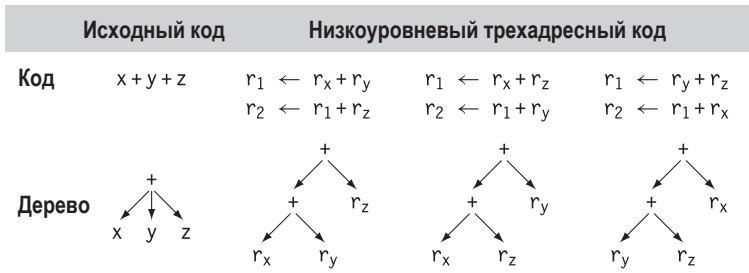


Рис. 7.1 Различные формы кода для выражения x+y+z

Левая ассоциативность порождает первое двоичное дерево. Это дерево кажется «естественным» в том смысле, что левая ассоциативность соответствует чтению текста на языке программирования слева направо. Посмотрим, что будет, если заменить y константой 2, а z – константой 3. Разумеется, x + 2 + 3 эквивалентно x + 5. Хорошо бы, чтобы компилятор обнаружил вычисление 2 + 3, произвел его и вставил результат прямо в код. В формах (x + y) + z и (x + z) + y сумма 2 + 3 не вычисляется. Напротив, в форме (y + z) + x явно вычисляется 2 + 3, что открывает возможность заменить поддерево на 5. Однако для каждого потенциального дерева существует такое назначение переменных и констант символам x, y и z, при котором константное выражение, пригодное для оптимизации, не проявляется.

Добавим к этому, что компилятор не может выбрать наилучшую форму этого выражения, если не понимает окружающий контекст. Например, если выражение x + y недавно вычислялось и значения x и y не были изменены, то использование левой формы позволило бы компилятору заменить первую операцию, $r_1 \leftarrow r_x + r_y$, ссылкой на недавно вычисленное значение.

В этой главе рассматриваются проблемы формы кода, возникающие при реализации многих типичных конструкций языка программирования. В основном нас занимает код, который должен быть сгенерирован для конкретных конструкций, а алгоритмы выбора конкретных ассемблерных команд мы игнорируем. Вопросы выбора команд, распределения регистров и планирования выполнения команд будут рассмотрены отдельно в последующих главах.

7.2. Арифметические операторы

Современные процессоры предлагают разностороннюю поддержку для вычисления выражений. Типичная RISC-машина имеет полный репертуар трехадресных операций, включая арифметические операторы, сдвиги и булевы

операторы. Трехадресная форма позволяет компилятору поименовать результат операции и сохранить его для последующего использования. Кроме того, она устраняет основную трудность, связанную с двухадресной формой: деструктивные операции.

```
loadI  @a      ⇒ r1
loadAO rarp,r1 ⇒ ra
```

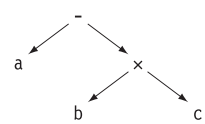
Чтобы сгенерировать код для тривиального выражения, например a + b, компилятор сначала генерирует код, гарантирующий, что значения a и b находятся в регистрах, скажем r_a и r_b. Если a хранится в памяти со смещением @a от начала текущей ЗА, то компилятор мог бы сгенерировать код, показанный во врезке, для перемещения a в регистр r_a. (Предполагается, что r_{arp} содержит текущий УЗА.) После того как a и b оказались в регистрах, компилятор может сгенерировать операцию add для вычисления их суммы.

С другой стороны, если значение уже находится в регистре, то код может просто сослаться на этот регистр. Если и a, и b находятся в регистрах, скажем r_a и r_b, то компилятор может просто сгенерировать операцию add.

Как мы видели в разделе 5.3, компилятор легко может сгенерировать код для выражений, если используется синтаксически управляемая схема трансляции. Если компилятор сначала строит графовое ПП, то может сгенерировать линейную форму позже, воспользовавшись простым обходом дерева вроде того, что показан на рис. 7.2(a). Он опирается на несколько функций, набранных *курсивным шрифтом*, чтобы сгенерировать код, который вызывает метод доступа к имени или числу (подробности см. в разделе 7.3.)

```
expr(node) {
    int result, t1, t2;
    switch(type(node)) {
        case ×, ÷, +, -:
            t1 ← expr(LeftChild(node));
            t2 ← expr(RightChild(node));
            result ← NextRegister();
            emit(op(node), t1, t2, result);
            break;
        case NAME:
            entry  ← STLookup(node);
            result ← ValueIntoReg(entry);
            break;
        case NUMBER:
            num    ← NumberFromNode(node);
            result ← NumberIntoReg(num);
            break;
    }
    return result;
}
```

(a) Кодогенератор, обходящий дерево



(b) Абстрактное синтаксическое дерево для a - b × c

```
loadI   @a       ⇒ r1
loadAO  rarp, r1 ⇒ r2
loadI   @b       ⇒ r3
loadAO  rarp, r3 ⇒ r4
loadI   @c       ⇒ r5
loadAO  rarp, r5 ⇒ r6
mult    r4, r6   ⇒ r7
sub     r2, r7   ⇒ r8
```

(c) Наивный код

Рис. 7.2 Простой кодогенератор, обходящий дерево выражения

На рис. 7.2(b) показано простое АСД для выражения a - b × c. А на рис. 7.2 (c) показан код, который кодогенератор, обходящий дерево, породил бы для

этого АСД. В примере предполагается, что a, b и c имеют известные смещения от УЗА: @a, @b и @c.

Генерирование команды Load Address Immediate

Внимательный читатель, возможно, обратил внимание, что кодогенератор, обходящий дерево, на рис. 7.2 генерирует последовательность из двух операций, loadI и loadAO, а не одну операцию loadAI:

```
loadI  @a       ⇒ r1
loadAO rarp, r1 ⇒ r2
```

вместо `loadAI rarp, @a ⇒ r2`.

Во всех примерах в этой книге предполагается, что выгоднее сгенерировать такую последовательность, чем одну операцию. В пользу такого решения есть два аргумента:

1) в более длинной последовательности кода @a присваивается явное имя. Если @a повторно используется в других контекстах, то это имя окажется полезно;
2) смещение @a может не поместиться в поле непосредственного операнда команды loadAI. Выяснить это лучше в селекторе команд.

Компилятор может преобразовать последовательность двух операций в одну на этапе оптимизации, если это окажется допустимо и уместно (например, если @a не используется повторно или если дешевле заново загрузить @a). Однако лучше отложить решение до выбора команд – по сути дела, иметь дело с машинно-зависимой длиной константы на том этапе компилятора, который и так уже в высшей степени машинно-зависим.

В простом кодогенераторе, обходящем дерево, используется один шаблон для всех операций +, -, × и ÷. Он генерирует код, который (1) гарантирует, что все операнды находятся в регистрах, и (2) выполняет операцию, оставляя результат в новом виртуальном регистре.

Большинство трехадресных операций укладываются в эту схему. Но некоторые, например возведение в степень или вычисление тригонометрических функций, требуют более сложных составных последовательностей на шаге (2). Их можно встроить прямо в код или обратиться к библиотечной функции, предоставленной компилятором или операционной системой.

7.2.1. Вызовы функций в выражении

До сих пор мы предполагали, что все операнды в выражении – переменные, константы и временные значения, порожденные подвыражениями. Но операндами могут быть и вызовы функций. Чтобы вычислить вызов функции, компилятор сначала генерирует для нее предварительную последовательность, а после вызова – заключительную последовательность, которая перемещает результат в нужное место на вызывающей стороне. Соглашение о связях ограничивает влияние вызываемой стороны на вызывающую.

Присутствие функции может ограничить способность компилятора менять порядок вычисления подвыражений. У функции могут быть побочные эффекты, модифицирующие значения переменных, используемых в выра-

жении. Компилятор должен строго придерживаться предложенного порядка вычисления исходного выражения, по крайней мере относительно вызова. Если он ничего не знает о возможных побочных эффектах вызова, то не вправе перемещать ссылки с одной стороны вызова на другую. Он обязан предполагать худшее – что функция модифицирует и использует все переменные, до которых может дотянуться. Стремление ослабить такие предположения в худшем случае привело к разработке межпроцедурного сводного анализа (см. раздел 9.4).

7.2.2. Выражения со смешанными типами

Большинство языков программирования допускают операции, содержащие операнды разных типов. В таких языках точно определена семантика «выражения со смешанными типами», или просто «смешанного выражения». (Мы будем говорить только о базовых типах исходного языка, оставляя в стороне типы, определенные пользователями.)

Во врезке на стр. 265 приведена сводка правил из стандарта C++.

В определении типичного языка разрешены одни смешанные выражения, например сложение целого и числа с плавающей точкой, но запрещены другие, например умножение на строку литер. В разрешенных ситуациях компилятору, возможно, придется вставить код, который преобразует одно или несколько участвующих в выражении значений к другому типу.

В случае явных преобразований должен быть способ представления их в ПП: соответствующие вершины в графовом ПП или операторы в линейном ПП.

Автор компилятора может поступить с неявными преобразованиями двумя способами: сделать их явными на ранних этапах трансляции и положиться на то, что оптимизатор уменьшит их стоимость, или оставить неявными до более позднего этапа трансляции, чтобы не загромождать ПП.

+	int	float
int	int	float
float	float	float

Таблица преобразований для +

Вставить явные преобразования просто: транслируя выражение, компилятор применяет правила преобразования исходного языка. Большую их часть можно выразить в виде зависящей от оператора таблицы, которая дает тип результата как функцию от типов операндов. Простая таблица преобразований для + показана на полях. Компилятор находит тип результата и при необходимости генерирует код преобразования операндов в тип результата. Затем он генерирует код выполнения операции уже с операндами типа результата.

C и C++ предоставляют базовые типы для целых разной ширины, каждый из которых позволяет представить разные диапазоны чисел. Оба языка применяют правило неявного «расширения», которое требует преобразовать операнды к самому узкому типу, способному представить оба операнда. Таким образом, чтобы сложить 8-разрядное целое с 64-разрядным, C предварительно преобразует 8-разрядный операнд в 64-разрядную форму.

Если язык не позволяет определить типы на этапе компиляции, то проверку и преобразование типов необходимо производить на этапе выполнения. Ситуация может потребовать добавления явного поля типа в представление значения на этапе выполнения и анализа вариантов для проверки типов, преобразования операндов и вычисления операции. Проверка на этапе выполнения влечет за собой затраты при каждом выполнении кода; при проверке на этапе компиляции затраты приходится нести только один раз.

Поля типа на этапе выполнения часто называются *тегами*.

Коммутативность, ассоциативность и системы счисления

Часто компилятор может воспользоваться алгебраическими свойствами операций. Сложение, умножение и булевы операции (and, or, xor) коммутативны и ассоциативны. Поэтому компилятор должен понимать, что для любых `a` и `b` значения выражений `a + b` и `b + a` одинаковы. Аналогично, видя выражения `a + b + c` и `d + a + b`, компилятору хорошо бы понять, что оба содержат подвыражение `a + b`. Если компилятор вычисляет выражения слева направо, то он не увидит общего подвыражения, потому что будет вычислять второе выражение как `(d + a) + b`.

По возможности компилятор должен использовать коммутативность и ассоциативность для повышения качества кода. Изменение порядка вычисления выражений может открыть дополнительные возможности для улучшения.

В силу конечной точности числа с плавающей точкой представляют лишь подмножество вещественных чисел, в котором ассоциативность не сохраняется. Поэтому компилятору не следует изменять порядок вычисления выражения с плавающей точкой, если только язык явно не разрешает это делать.

Рассмотрим следующий пример: вычислить `a - b - c`, где `a`, `b`, `c` – числа с плавающей точкой. Мы можем присвоить значения так, что

`b, c < a  a - b = a  a - c = a`

но `a - (b + c) ≠ a`. В таком случае результат зависит от порядка вычисления. Вычисление `(a - b) - c` дает значение, неотличимое от `a`, тогда как вычисление `a - (b + c)` дает значение, отличное от `a`.

Эта проблема возникает из-за приближенной природы чисел с плавающей точкой; мантисса мала относительно диапазона показателя степени. Чтобы вычесть одно число из другого, оборудование должно нормировать их; если разность показателей степени больше точности мантиссы, то меньшее число будет усечено до нуля. Компилятор не может просто так обойти эту проблему, поэтому в общем случае должен избегать переупорядочения вычислений с плавающей точкой.

Операция присваивания как выражение

Компилятор может рассматривать присваивание как оператор с необычным методом вычисления. Вычисление выражения в правой части присваивания дает конкретное значение, которое часто называют *r-значением*. Все обсуждавшиеся выше приемы вы-

R-значение
выражение, результатом вычисления которого является значение, как в правой части присваивания.

L-значение
выражение, результатом вычисления которого является место, как в левой части присваивания.

Вычисление l-значения может потребовать кода, как в случае элемента массива. В других случаях, например когда скалярное значение хранится в регистре, кода может и не потребоваться.

числения выражений порождают r-значения. Вычисление выражения в левой части присваивания дает место, или *l-значение*, которое может быть адресом в памяти или регистром. Вычисление l-значений обсуждается в разделе 7.3.

Чтобы сгенерировать код присваивания, компилятор сначала генерирует код вычисления правой части с получением r-значения. Затем вычисляется l-значение левой части, или место. Если типы r-значения и l-значения различны, то язык может потребовать, чтобы компилятор вставил код преобразования r-значения к типу l-значения. Наконец, компилятор генерирует код перемещения r-значения в указанное место.

7.2.3. Уменьшение спроса на регистры

В общем случае компилятор может хранить однозначные скалярные значения в регистрах (см. раздел 4.7.2). Типичная стратегия формирования кода заключается в том, чтобы назначить каждому значению, которое может находиться в регистре, виртуальный регистр, а затем положиться на распределитель регистров, который решит, какие значения должны размещаться в физических регистрах в каждой точке кода (см. главу 13). Однако компилятор может формировать код так, чтобы уменьшить спрос на регистры в конечном коде.

Вычисление выражений дает понятный пример. Снова рассмотрим вычисление `a - b × c`, где `a`, `b` и `c` хранятся в ЗА с известными смещениями от УЗА. На рис. 7.3 показано два порядка вычисления выражения – до и после распределения регистров. На рисунке (a) показан код, взятый с рис. 7.2(c), в предположении, что множество виртуальных регистров не ограничено. На рисунке (b) показан тот же код после распределения регистров; требуется три физических регистра, помимо r_{arp}. Этот результат имеет смысл; код загружает значения `a`, `b` и `c` в регистры до выполнения умножения и вычитания. Такой результат дает и синтаксически управляемая схема, и обход дерева слева направо.

Изменение порядка вычислений может уменьшить спрос на регистры. На рисунке (c) показана версия кода, в которой сначала производится умножение, т. е. `b × c` вычисляется до загрузки `a`. После распределения регистров эта версия требует на один физический регистр меньше, как показано на рисунке (d).

Порядок вычислений на рисунке (a) соответствует обходу дерева слева направо, а порядок вычислений на рисунке (c) – обходу справа налево. Поскольку приоритет умножения выше, чем сложения, при обходе слева на-

право значение a загружается в регистр, но должно быть сохранено во время вычисления b × c. В этом примере вычисление справа налево откладывает загрузку a до завершения умножения.

```
loadI  @a      ⇒ r1
loadAO rагр,r1 ⇒ r2
loadI  @b      ⇒ r3
loadAO rагр,r3 ⇒ r4
loadI  @c      ⇒ r5
loadAO rагр,r5 ⇒ r6
mult   r4,r6   ⇒ r7
sub    r2,r7   ⇒ r8
```

(a) Код на рис. 7.2(c)

```
loadI  @a      ⇒ r1
loadAO rагр,r1 ⇒ r1
loadI  @b      ⇒ r2
loadAO rагр,r2 ⇒ r2
loadI  @c      ⇒ r3
loadAO rагр,r3 ⇒ r3
mult r2,r3     ⇒ r2
sub r1,r2      ⇒ r2
```

(b) Код после распределения регистров

```
loadI @c       ⇒ r1
loadAO rагр,r1 ⇒ r2
loadI @b       ⇒ r3
loadAO rагр,r3 ⇒ r4
mult r2,r4     ⇒ r5
loadI @a       ⇒ r6
loadAO rагр,r6 ⇒ r7
sub r7,r5      ⇒ r8
```

(c) Сначала вычислить b × c

```
loadI @c       ⇒ r1
loadAO rагр,r1 ⇒ r1
loadI @b       ⇒ r2
loadAO rагр,r2 ⇒ r2
mult r1,r2     ⇒ r1
loadI @a       ⇒ r2
loadAO rагр,r2 ⇒ r2
sub r2,r1      ⇒ r1
```

(d) Код после распределения регистров

Рис. 7.3 Переписывание a - b x c с целью уменьшить спрос на регистры

Конечно, лучший порядок зависит от контекста. В случае бинарной операции, если компилятору для вычисления одного операнда нужно больше регистров, чем для вычисления второго, он должен вычислять более требовательный операнд сначала. При этом спрос на регистры уменьшится на единицу.

Для применения этого правила компилятор должен исследовать выражение дважды: первый раз для вычисления спроса, а второй – когда выбирает порядок вычислений подвыражений и генерирует код. Это наблюдение применяется во многих контекстах оптимизации и генерирования кода. Для получения хороших результатов часто необходимо, чтобы компилятор проанализировал код, перед тем как его перезаписывать.

Это наблюдение не новое. Флойд обратил внимание на сам принцип в 1961 году [160]. Сети применил его к уменьшению спроса на регистры десятью годами позже [321].

ИТОГИ РАЗДЕЛА

В главе 5 мы познакомились с трансляцией выражений, а также с каркасом для генерирования графовых или линейных ПП. В этом разделе показано, как компилятор может применить те же идеи при обходе ПП для генерирования представления более низкого уровня. Также были рассмотрены некоторые осложнения, возникающие в ПП выражений, такие как встроенные вызовы функций, преобразования типов и присваивание.

Построение ПП для выражения и обход его несколько раз позволяют компилятору проанализировать код, вывести знания о нем и скорректировать его, так чтобы повысить эффективность конечного кода. Поскольку значительная доля времени выполнения тратится на вычисление выражений, эффективный код вычислений очень важен с точки зрения общей скорости работы сгенерированной программы.

ВОПРОСЫ

1. В ILOC есть команда rsubI, immediate reverse subtract (вычитание аккумулятора из непосредственного операнда). Она может упростить трансляцию выражений. В каких случаях она может быть полезна? Почему не включена также версия rsub (вычитание аккумулятора из операнда в памяти)?
2. Опишите два случая, когда несколько проходов по ПП могут улучшить качество кода.

7.3. Методы доступа к значениям

В разделе 7.2 неявно предполагается, что один метод доступа работает для всех идентификаторов. На практике же разные виды переменных требуют различных методов доступа. Мы начнем этот раздел рассмотрением методов доступа к скалярным переменным. Затем опишем сложности, возникающие из-за необходимости находить отдельные элементы агрегатов, таких как структуры, объекты и массивы.

7.3.1. Методы доступа к скалярным переменным

Для скалярных переменных метод доступа должен сначала определить, находится значение в регистре или в памяти. С доступом к регистру все просто: в коде просто указывается имя виртуального регистра. Доступ к памяти требует различных значений в зависимости от класса памяти переменной.

Переменные, хранящиеся в регистре

Если компилятор может хранить значение переменной в регистре в части кода, близкой к точке определения или использования, то при ссылке на это значение достаточно просто указать имя регистра.

На большинстве машин команды прямого доступа к значению в регистре позволяют выполнить определение и использование в одном такте. Например, операция `add` $r_1, r_2 \Rightarrow r_1$ может в одном такте прочитать и записать регистр r_1. Одна из целей оптимизации и основная цель распределения регистров – хранить значения в регистрах на тех участках кода, где они часто используются.

Переменные, хранящиеся в памяти

Есть ряд причин, по которым основным местом хранения переменной является память. Имя может быть неоднозначным, потому что в коде имеет-

ся несколько имен, ссылающихся на одно и то же место. Имя может иметь глобальную видимость, поэтому при компиляции части приложения невозможно судить о его однозначности (или неоднозначности). Имя может иметь статическое время жизни, поэтому его значение следует сохранять между вызовами процедуры.

ПП может представлять каждую переменную в языке с лексической видимостью ее статической координатой: парой ⟨*l*, *o*⟩, где *l* – лексический уровень, на котором определена переменная, а *o* – ее смещение от начала области данных уровня *l*. Глобальным и статическим переменным обычно назначается уровень 0; для них компилятор должен также запомнить их конкретную область данных.

Вообще, адрес переменной в памяти вычисляется по формуле *base* + *offset*, где *base* – виртуальный адрес начала области данных, а *offset* – смещение переменной от *base*. Для разных уровней в статической координате требуются разные стратегии вычисления *base*.

Локальные переменные

Для локальных автоматических переменных компилятор может выделить память в области локальных данных, в конце ЗА процедуры. Чтобы сослаться на переменную, компилятор генерирует код, прибавляющий известное смещение переменной к текущему УЗА, который хранится в выделенном регистре r_{arp}.

Однозначные локальные скалярные автоматические переменные могут размещаться в регистрах. Если спрос на регистры велик, распределитель регистров может временно выгружать часть этих значений в память. При этом адреса выгрузки обычно находятся в области локальных данных, чтобы код выгрузки и восстановления мог использовать r_{arp} в качестве базового адреса.

Локальные переменные в объемлющих областях видимости

Для локальных автоматических переменных, объявленных в лексически объемлющих областях видимости, компилятор должен сгенерировать код, в котором используется механизм адресуемости, диктуемый соглашением о связях (см. раздел 6.4.3).

Хотя глобальная область видимости объемлет все лексические области видимости, для доступа к ней используется другой механизм.

Чтобы найти переменную уровня *l* с помощью связей доступа, компилятор генерирует код для нахождения связи доступа уровня *l*, которая служит базовым адресом. В случае глобального дисплея необходимо загрузить этот базовый адрес (УЗА объемлющей области видимости уровня *l*) напрямую. Затем он вычисляет виртуальный адрес как *base* + *offset* и использует его в операциях `load` и `store`.

Статические и глобальные переменные

Для переменной, расположенной в статической области данных, базовый адрес помечается ассемблерной меткой, которая обычно порождается путем

Для каждой глобальной переменной компилятор мог бы сгенерировать уникальную метку с декорированным именем, избежав тем самым сложения со смещением.

декорирования имени, ассоциированного с областью видимости. Компилятор получает текстовое имя метки для ссылки и организует ее загрузку в регистр в виде перемещаемой метки уровня ассемблера.

Как именно компилятор получает адрес текстового имени метки на этапе выполнения, определяется свойствами ISA. Мы предполагаем, что имя метки помещается в непосредственный операнд команды загрузки. Если это не так, то компилятору может потребоваться сохранить метку в области статических данных (пуле констант), ассоциированной с процедурой, и загружать ее значение оттуда. Конечно, коду еще необходимо найти пул констант. Одна из возможных схем – разместить пул констант на известном расстоянии от начала или конца процедуры и обращаться к нему по относительному адресу.

Если целое число хранится со смещением 8 от начала области статических данных, то код вычисления адреса мог бы иметь вид:

```
loadI <label> ⇒ r_i  // получить базовый адрес
addI  r_i,8   ⇒ r_j  // прибавить смещение
```

где <*label*> – декорированная метка соответствующей области видимости. За этой последовательностью может располагаться команда загрузки из r_j или сохранения в r_j.

Переменные, передаваемые как параметры

Формальные параметры ставят концептуальную проблему. Рассмотрим функцию *f(x)*, принимающую один целый параметр *x*. В каждом месте вызова в *x* может передаваться другая переменная, или *фактический параметр*. У фактических параметров могут быть разные классы памяти. Код доступа к *x* внутри *f* не должен зависеть от класса памяти фактического параметра.

Отметим, что для доступа к параметрам, передаваемым по ссылке, требуется на один уровень косвенности больше, чем для параметров, передаваемых по значению.

Многие соглашения о связях предусматривают специальные регистры для передачи первых нескольких параметров. Для параметров, передаваемых по значению, компилятор сохраняет фактическое значение в регистре, а затем использует имя регистра в операциях, ссылающихся на формальный параметр. Для параметров, передаваемых по ссылке, компилятор сохраняет в регистре адрес фактического параметра, а затем генерирует команды `load` и `store` для доступа к значению формального параметра.

Для параметров, передаваемых в памяти, компилятор может сгенерировать код, в котором УЗА используется в качестве базового адреса. Каждый параметр имеет свое смещение относительно УЗА; компилятор может сгенерировать код для вычисления виртуального адреса его позиции и использовать этот адрес. Повторим, что для параметра, передаваемого по значению,

в ЗА хранится само значение, а для параметра, передаваемого по ссылке, – его виртуальный адрес.

Переменные, хранящиеся в куче

У переменных, хранящихся в куче, обычно нет ни базового адреса, ни смещения. Процедура выделения памяти из кучи возвращает виртуальный адрес, который следует сохранить для последующего доступа к сущности. Этот виртуальный адрес можно хранить либо в переменной, либо в поле какой-то другой выделенной в куче структуры.

Для получения доступа к сущности в куче компилятор должен извлечь адрес сущности, пользуясь подходящим методом доступа. Зная адрес, он может сгенерировать операции для работы с его значением, находящимся в памяти.

Сборка мусора опирается на тот факт, что к сущности, выделенной в куче, невозможно обратиться, если в программе не хранится копия ее адреса.

Если откомпилированный код содержит арифметические операции с произвольными указателями или в нем имеется переполнение буфера, то это предположение может оказаться неверным.

Рассмотрим локальную автоматическую переменную `root`, которая указывает на список выделенных в куче структур, каждая из которых имеет указатель `next`. Чтобы сгенерировать код обхода списка, компилятор генерирует код для загрузки значения `root`. Это значение используется как адрес начальной структуры. Для перемещения по списку смещение поля `next` относительно начала структуры прибавляется к адресу текущего элемента списка и загружается адрес следующего элемента.

7.3.2. Методы доступа к агрегатам

Методы доступа к агрегатным объектам сложнее, чем к скалярным, потому что компилятор должен генерировать год для поиска конкретных элементов внутри агрегата. Для доступа к элементу структуры, объекта или массива компилятору нужен адрес начала агрегатного объекта, смещение внутри агрегата и тип.

Элементы структуры

Компилятор определяет, как найти адрес начала структуры, просматривая таблицу символов на пути поиска для контекста, в котором встретилась ссылка. Эта часть процесса аналогична поиску скалярной переменной. Компилятор использует класс памяти, чтобы определить, как искать начальный адрес. Это может быть смещение от текущего УЗА или УЗА объемлющей области видимости. Это может быть также смещение от какой-то статической или глобальной метки. Наконец, начальный адрес может храниться в какой-то переменной, которую саму еще нужно найти.

Сгенерировав код, гарантирующий, что начальный адрес находится в регистре, компилятор ищет информацию о смещении и типе в таблице симво-

лов для внутренней области видимости структуры. Он генерирует код сложения начального адреса со смещением – чтобы получить эффективный виртуальный адрес элемента. Информация о типе позволяет компилятору сгенерировать подходящую операцию загрузки (или несколько операций).

Члены объекта

Чтобы найти заданный член конкретного объекта, компилятор должен сначала найти запись этого объекта (ЗО). Поскольку в работающем коде в качестве идентификатора объекта используется указатель записи объекта (УЗО), нахождение УЗО, или начального адреса ЗО, тривиально. В исходном языке для ссылки на объект используется имя; это имя содержит УЗО. Компилятор обеспечивает помещение УЗО в регистр, применяя методы, описанные для скалярной переменной.

Чтобы получить тип члена и его смещение от начала ЗО, компилятор ищет имя члена в таблице символов для внутренней области видимости объекта. Затем он генерирует код сложения базового адреса ЗО со смещением, чтобы узнать начальный адрес члена. Информация о типе позволяет компилятору сгенерировать правильный код операции (или нескольких операций) загрузки.

Векторы



Размещение вектора в памяти

Обычно языки программирования определяют вектор как одномерный непрерывный массив с заданными верхней и нижней границами индексов. Вектор `V[3:9]` содержит (9 – 3) + 1 элементов, как показано во врезке. Чтобы обратиться к *i*-му элементу `V`, компилятору нужен начальный адрес `V`, который мы будем обозначать `@V`, а также нижняя и верхняя границы `V` и смещение *i*-го элемента.

Компилятор может вычислить смещение по формуле (*i* – *low*) × *w*, где *low* – объявленная нижняя граница `V`, а *w* – длина одного элемента `V`. Так, если *low* = 3, *i* = 6 и *w* = 4, то смещение от `@V` *i*-го элемента равно (6 – 3) × 4 = 12. Если регистры $r_{@V}$ и r_i содержат значения `@V` и `i` соответственно, то следующий фрагмент кода загружает значение `V[i]` в $r_{V[i]}$.

```
subI  ri,3    ⇒ r1     // (i - нижняя граница)
multI r1,4    ⇒ r2     // x длина элемента (4)
add   r@V,r2  ⇒ r3     // адрес V[i]
load  r3      ⇒ rV[i]  // получить значение V[i]
```

Вычисление этой простой ссылки требует трех арифметических операций. Но компилятор может поступить лучше. Если нижняя граница, *low*, известна, то компилятор может включить вычитание в `@V`, вычислив константу $@V_0$ = `@V` – *low* × *w*, которую мы будем называть *сдвинутым нулем* `V`.

Сдвинутый нуль
Сдвинутым нулем вектора `V` называется адрес, по которому должен был бы располагаться `V[0]`.

В случае многомерного массива это положение нулевого индекса по каждому измерению.

Многие ссылки на память имеют вид базового адреса и смещения. Поэтому большинство процессоров поддерживают двухоперандную операцию `load`, аналогичную операции ILOC `loadAO` (load-address-offset – загрузить по адресу и смещению). При использовании `loadAO` сложение включается в операцию с памятью.

Третья арифметическая операция – умножение. В большинстве процессоров целочисленное умножение занимает несколько тактов. Если *w* – степень двойки и индекс *i* положителен, то компилятор может заменить умножение сдвигом влево. В итоге получаем следующую последовательность команд ILOC:

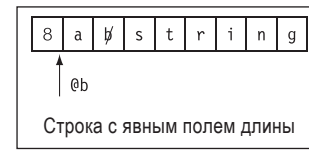
```
lshiftI r_i,2      ⇒ r_2      // i × 4
loadAO  r_@V[0],r_2 ⇒ r_V[i]  // получить значение V[i]
```

Этот код короче и предположительно быстрее. Хороший программист на ассемблере написал бы именно такой код.

Автор компилятора мог бы раскрыть детали адресной арифметики для более общей оптимизации, например устранения избыточности или снижения стоимости операций (см. разделы 8.4.1 и 10.7.2). Низкоуровневые улучшения, такие как замена умножения сдвигом и сворачивание сложения и загрузки, можно произвести на более поздних этапах компиляции.

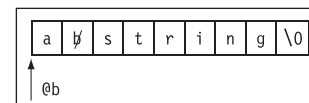
Строки

Строки обычно хранятся как линейные последовательности элементов. Поэтому вычисление адреса элемента строки производится так же, как для элемента вектора. Форматы строк различаются способом представления длины строки. Хотя мы в этом обсуждении ограничиваемся строками литер, другие типы строк аналогичны.



Строка с явным полем длины

В случае представления с явной длиной в строке хранится целый счетчик числа элементов, как показано во врезке. Видно, что смещение длины от указателя на строку отрицательно. Чтобы узнать длину строки, код во время выполнения просто читает значение из поля длины.



Строка, завершающаяся нулем

В случае представления с неявной длиной конец строки обозначается специальным значением. Самый известный формат строки с неявной длиной – завершающаяся нулем строка литер в языке C. Как показано во врезке, последним символом строки является нуль, обозначаемый как `"\0"`. Чтобы узнать длину строки, код во время выполнения должен обойти ее и посчитать количество символов до завершающего нуля.

Мы отложим реализацию операций со строками до раздела 7.6, а сначала поговорим о реализации циклов и других управляющих конструкций.

Многомерные массивы

Для обращения к элементу многомерного массива компилятор также применяет схему «базовый адрес плюс смещение». Но в многомерном случае вычисление смещения сложнее, чем для вектора или строки. Мы покажем это вычисление на примере двумерного массива, хранимого по строкам или в виде множества векторов указателей (см. раздел 5.6.3). Эти схемы легко обобщаются на случай большего числа измерений. Код для массива, хранимого по столбцам, аналогичен; оставляем его в качестве упражнения для читателя.

Хранение по строкам. Чтобы описать вычисления адреса для многомерных массивов, мы обобщим нотацию, применяемую для метода доступа к вектору. Рассмотрим массив `A` с такими размерностями `A[1:2,1:4]`. Обозначим low_1 нижнюю границу по первому измерению, а low_2 – по второму. Для `A` $low_1 = 1$, $high_1 = 2$, $low_2 = 1$, $high_2 = 4$. Чтобы упростить изложение, обозначим $len_k = high_k - low_k + 1$ размер массива по измерению k.

Для доступа к элементу `A[i,j]` компилятор может сгенерировать код, который (1) вычисляет смещение начала i-й строки; (2) вычисляет смещение j-го элемента этой строки; (3) вычисляет сумму смещения строки, смещения столбца и базового адреса массива. Для `A[i,j]` вычисление адреса выглядит так:

$$\texttt{@A} + (\texttt{i} - low_1) \times len_2 \times w + (\texttt{j} - low_2) \times w$$

Здесь `@A` – базовый адрес `A`; $(\texttt{i} - low_1) \times len_2$ – число элементов до начала `i`-й строки, $(\texttt{j} - low_2)$ – смещение (в элементах) `j`-го элемента строки. Смещения умножаются на w, длину элемента в байтах, чтобы перейти к адресам, выраженным в байтах.

Подставляя фактические значения `i`, `j`, low_1, len_2, low_2 и w, находим, что смещение `A[2,3]` от `A[1,1]` равно

$$(2 - 1) \times (4 - 1 + 1) \times 4 + (3 - 1) \times 4 = 24$$

(в предположении, что `@A` указывает на элемент `A[1,1]` со смещением 0). Глядя на размещение `A` в памяти, находим, что адрес `A[1,1]` + 24 и есть адрес `A[2,3]`.

В случае вектора мы смогли упростить вычисление при условии, что верхняя и нижняя границы были известны на этапе компиляции. Применение аналогичных алгебраических выкладок к вычислению сдвинутого нуля в двумерном случае дает:

$$\texttt{@A} + (\texttt{i} \times len_2 \times w) - (low_1 \times len_2 \times w) + (\texttt{j} \times w) - (low_2 \times w).$$

После изменения порядка членов получаем формулу

$$\texttt{@A} + (\texttt{i} \times len_2 \times w) + (\texttt{j} \times w) - (low_1 \times len_2 \times w + low_2 \times w),$$

в которой последний член, ($low_1 \times len_2 \times w + low_2 \times w$), не зависит от `i` и `j`. Этот член можно включить в сдвинутый нуль `A`:

$@A_0 = @A - (low_1 \times len_2 \times w + low_2 \times w) = @A - 20.$

Теперь вычисление адреса `A[i,j]` сводится просто к

$@A_0 + i \times len_2 \times w + j \times w.$

Наконец, мы можем вынести w за скобки, сэкономив одно умножение:

$@A_0 + (i \times len_2 + j) \times w.$

Адрес `A[2,3]` по этой формуле вычисляется так:

$@A_0 + (2 \times 4 + 3) \times 4 = @A_0 + 44.$

Поскольку $@A_0$ – просто `@A` − 20, это эквивалентно `@A` − 20 + 44 = `@A` + 24 – тот же адрес, что был вычислен первоначально.

Если предположить, что `i` и `j` находятся в регистрах r_i и r_j, а len_2 – константа, то эта переработанная формула приводит к такой последовательности команд:

```
loadI  @A0       ⇒ r@A0  // скорректированная база A
multI  ri, len2  ⇒ r1    // i × len2
add    r1, rj    ⇒ r2    // + j
multI  r2, 4     ⇒ r3    // x w
loadAO r@A0, r3  ⇒ ra    // значение A[i,j]
```

Теперь вычисление содержит два умножения и два сложения (одно включено в `loadAO`). Вторую `multI` можно заменить сдвигом.

Если у компилятора нет доступа к границам массива, то он должен либо вычислять сдвинутый нуль на этапе выполнения, либо использовать полную формулу, включающую нижние границы. Первый вариант может быть выгоден, если процедура обращается к нескольким элементам массива; вычисление сдвинутого нуля для всей процедуры позволяет использовать в коде менее дорогое вычисление адреса. Более сложное вычисление имеет смысл, только если обращений к массиву немного.

> Компилятор мог бы вставить вычисление сдвинутого нуля после пролога процедуры и положиться на устранение мертвого кода, если оно не нужно, – например, если единственным использованием массива является передача фактического параметра в месте вызова.

Идеи, лежащие в основе вычисления адресов для двумерного массива, обобщаются на массивы с большим числом измерений. Формулу адреса для массива, хранимого по столбцам, можно вывести, пользуясь теми же соображениями, что для хранения по строкам. Оптимизации, которые мы применили для уменьшения стоимости вычисления адресов, равным образом применимы и к формулам для таких массивов.

Векторы указателей. Представление с помощью векторов указателей концептуально просто. На каждом уровне массива используется набор указателей, как показано на рис. 7.4. Самый внутренний уровень векторов содержит фактические элементы данных; на более внешних уровнях находятся указатели, ведущие на векторы следующего уровня, так что в итоге цепочка операций доступа по указателям приводит к искомому элементу массива.

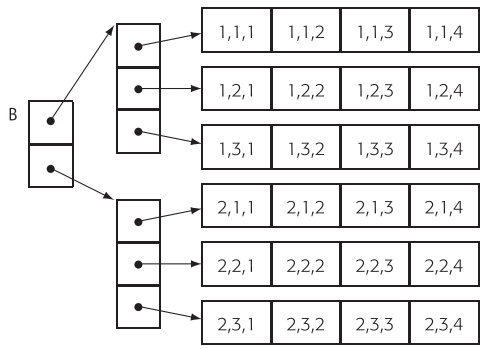


Рис. 7.4 Векторы указателей в порядке по строкам для массива `B[1:2,1:3,1:4]`

Рассмотрим трехмерный массив `B`, показанный на рис. 7.4. Чтобы обратиться к элементу `B[i,j,k]`, компилятор генерирует код, в котором используется $@B_0$, `i` и длина указателя для нахождения начала вектора указателей для подмассива `B[i,*,*]`. Затем он генерирует код, в котором используется этот результат, а также `j` и длина указателя для нахождения начала вектора для `B[i,j,*]`. Наконец, этот результат наряду с `k` и длиной *w* элемента `B` используется для вычисления адреса `B[i,j,k]`.

Если значения `i`, `j` и `k` находятся в регистрах r_i, r_j и r_k соответственно, а $@B_0$ –сдвинутый нуль для адреса по первому измерению, то компилятор мог бы сгенерировать следующий код загрузки `B[i,j,k]`:

```
loadI  @B0     ⇒ r@B0  // сдвинутый нуль B
multI  ri, 4   ⇒ r1    // предполагается, что длина указателя 4 байта
loadAO r@B0, r1 ⇒ r2   // получить @B[i,*,*]
multI  rj, 4   ⇒ r3    // длина указателя 4 байта
loadAO r2, r3  ⇒ r4    // получить @B[i,j,*]
multI  rk, 4   ⇒ r5    // предполагается, что длина элемента равна 4
loadAO r4, r5  ⇒ rb    // значение B[i,j,k]
```

В этом коде предполагается, что указатели в векторе скорректированы с учетом ненулевых нижних границ. Если это не так, то значения в r_j и r_k следует уменьшить на величину соответствующих нижних границ. В примере умножения могут быть заменены сдвигами.

При использовании векторов указателей вычисление ссылок требует всего двух операций на каждое измерение. На машинах, где доступ к памяти быстрый по сравнению с арифметическими операциями – например, на

большинстве компьютеров, выпущенных до 1985 года, – этот метод доступа был быстрым. Но по мере того как стоимость доступа к памяти начала расти относительно стоимости арифметики, эта схема утратила преимущество в быстродействии.

На машинах с процессорным кешем локальность играет важнейшую роль для производительности. Когда массив перестает помещаться в кеш, порядок хранения влияет на локальность. Хранение по строкам и по столбцам дает хорошую локальность для некоторых операций с массивами. Свойства локальности массива, реализованного с помощью векторов указателей, компилятору труднее предсказать и, пожалуй, оптимизировать.

Массив структур. Некоторые языки программирования разрешают объявлять массив структур – несколько экземпляров структуры или записи с индексами, как в массиве, позволяющими обращаться к конкретным экземплярам (элементам массива структур). Метод доступа к массиву структур опирается на формулу для массива, чтобы найти начало экземпляра. А внутри этого экземпляра используется стандартный метод доступа к структуре (база + смещение члена).

Говоря *массив структур*, мы имеем в виду, что все элементы массива имеют одинаковый структурный тип.

Доступ к параметрам-массивам

Большинство языков позволяют передавать процедурам массивы по ссылке, чтобы не копировать при вызове значения каждого элемента. Вызывающая и вызываемая стороны должны договориться о размещении массива в памяти и методе доступа – варианте полиномиальной формулы. Вызывающая сторона должна предоставить вызываемой значения, необходимые методу доступа.

Некоторые языки требуют, чтобы необходимые значения передавал программист. Например, в FORTRAN программист должен объявить размерности массива, используя либо константы, либо другие формальные параметры. В других языках компилятор сам собирает, организует и передает необходимую информацию вызываемой стороне. Компилятор строит дескриптор, называемый *вектором предварительной информации* (dope vector), который содержит базовый адрес массива и всю необходимую информацию о каждой размерности. Размер вектора предварительной информации известен, поэтому компилятор может выделить для него место в ЗА вызывающей или вызываемой стороны. Адрес этого вектора передается в согласованном параметре как часть предварительной последовательности.

Вектор предварительной информации дескриптор фактического параметра-массива.

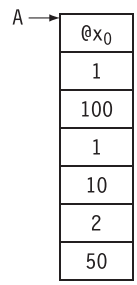
Вектор предварительной информации можно также использовать для массивов, границы которых определяются во время выполнения.

На рис. 7.5 приведен пример из PL/I. Процедура `main` дважды вызывает `fee`. На рисунках (b) и (c) показаны векторы предварительной информации для

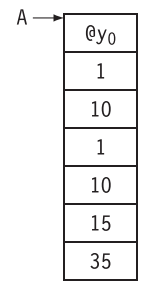
обоих мест вызова. Если при любом доступе используется вариант формулы вычисления адреса со сдвинутым нулем, то векторы предварительной информации будут содержать длины A по каждому измерению, а не верхние и нижние границы.

```
program main;
  begin;
    declare x(1:100,1:10,2:50),
      y(1:10,1:10,15:35) float;
    call fee(x);
    call fee(y);
  end main;
procedure fee(A)
  declare A(*,*,*) float;
      ...
  end fee;
```

(a) Код на PL/I для передачи массивов целиком



(b) Вектор предварительной информации для первого места вызова



(c) Вектор предварительной информации для второго места вызова

Рис. 7.5 Векторы предварительной информации

Для доступа к элементу массива-параметра компилятор генерирует ту же самую полиномиальную формулу, что для ссылки на локальный массив, с тем отличием, что значения базового адреса и размерностей берутся из вектора предварительной информации. Таким образом, в активациях процедуры для доступа к разным массивам из разных мест вызова можно использовать один и тот же код. Ту же технику можно применить к массивам динамического размера, выделяемым из кучи. В момент выделения памяти компилятор может построить вектор предварительной информации, который будет использоваться при последующих ссылках.

Заметим, что доступ через вектор предварительной информации обходится дороже доступа к локальному массиву с известными границами. В лучшем случае вектор предварительной информации привносит дополнительные обращения к памяти, а в худшем препятствует компилятору производить оптимизации, нуждающиеся в сведениях из объявления массива.

7.3.3. Проверка диапазона

В большинстве языков программирования предполагается явно или неявно, что программа обращается только к адресам, не выходящим за определенные границы структур. Например, нельзя обращаться к элементу массива, находящимуся до нижней или после верхней границы. Как и к элементам строки, находящимся за ее границами.

Программа, обращающаяся к элементу за границей структуры, по определению, некорректна. В некоторых языках, например JAVA и ADA, требуется, чтобы доступ за границей структуры обнаруживался и диагностировался. Другие языки оставляют этот вопрос на усмотрение автора компилятора.

Некоторые компиляторы включают код *проверки диапазона*, другие не включают. На отсутствие такой проверки в C возлагают по крайней мере частичную ответственность за атаки с переполнением буфера – исторически, пожалуй, одну из самых широко распространенных проблем компьютерной безопасности.

Проверка диапазона
проверка того, что доступ к агрегатной структуре данных производится в пределах границ, заданных при ее объявлении или выделении для нее памяти.

При простейшей реализации проверка вставляется перед каждым обращением к агрегатной сущности. Проверяется, что вычисленный индекс попадает в разрешенный диапазон. Если это не так, то возбуждается исключение.

В примере во врезке проверка вставлена перед обращением к массиву во внутреннем цикле. Предполагается, что массив `a` имеет границы $a[lb_1:ub_1,lb_2:ub_2]$. Такая наивная проверка сопряжена со значительными накладными расходами.

```
for i ← 1 to n
  for j ← 1 to m
    if (lb1 ≤ i ≤ ub1 and
        lb2 ≤ j ≤ ub2) then
      ... a[i,j] ...
    else возбудить исключение
```

Наивная проверка диапазона

Часто компилятор может улучшить наивные проверки, объединив их или доказав, что обращения безопасны. В примере на полях проверки, совершаемые во внутреннем цикле, можно объединить и вынести за пределы циклов. В преобразованном коде имеется всего одна проверка, а не по одной на каждое обращение.

```
if (lb1 ≤ 1 and n ≤ ub1 and
    lb2 ≤ 1 and m ≤ ub2)
    then
      for i ← 1 to n
        for j ← 1 to m
          ... a[i,j] ...
      else возбудить исключение
```

Оптимизированная проверка диапазона

Если во вложенном цикле есть несколько обращений к `A`, то выигрыш может быть еще больше. При наличии нескольких одинаковых обращений компилятор может проверять границы только один раз. Если во вложенном цикле несколько обращений с разными индексными выражениями, то компилятор должен рассмотреть каждое из них. Он может также объединить несколько проверок алгебраически, что еще снизит стоимость.

Если компилятор знает значение некоторых или всех имен, используемых в проверках, то сможет перенести некоторые проверки на этап компиляции. В нашем примере компилятор знает, что переменные цикла, *i* и *j*, начинаются с единицы. Если он знает значение lb_1, то может выполнить проверку $lb_1 \le 1$ во время компиляции, а не во время выполнения. Такая же стратегия применима к каждой проверке – если значения известны, то можно переносить проверку на этап компиляции.

Для проверки диапазона компилятор должен знать размер структуры данных. Если этот размер статический и известный, то компилятор может включить его прямо в проверку. Если же размер структуры данных динами-

Для объекта, выделенного в куче, компилятор мог бы использовать выделенный размер для аппроксимации протяженности. При такой схеме оценка объявленных границ может быть завышенной, но обращения за пределами выделенной памяти удастся избежать.

ческий, то система времени выполнения должна запомнить информацию о размере, которая потребуется при проверке диапазона.

ИТОГИ РАЗДЕЛА

Компилятору необходим метод доступа к любому участвующему в вычислениях значению. Метод может быть тривиальным, как в случае, когда значение находится в регистре. Он может потребовать нескольких команд, как в случае скалярного значения, хранящегося в ЗА объемлющей лексической области видимости. Он может быть и достаточно сложным, как в случае элемента массива структур. Для любого значения и типа компилятору нужен план, чтобы он мог построить подходящую конструкцию ПП для обращения к сущности исходной программы.

В этом разделе описаны стандартные приемы доступа к большинству видов значений в языках программирования. Компилятор использует комбинацию этих приемов, диктуемую реализацией других конструкций и определением самого исходного языка. На практике сгенерированный код должен содержать достаточно деталей, чтобы последующие проходы в фазах оптимизации и синтеза могли создать компактный и эффективный код.

ВОПРОСЫ

1. Рассмотрим массив `A[0:99,0:89,0:109]`. Напишите код вычисления адреса `A[i,j,k]` в предположении, что (a) `A` хранится по строкам и (b) `A` хранится с использованием векторов указателей.
2. Объясните в терминах организации размещения в памяти и кода, требуемого для доступа к конкретному элементу, разницу между массивом структур и структурой массивов.

7.4. Булевы и реляционные операторы

Реляционные выражения обычно порождают булевы значения.

Языки программирования почти всегда включают средства задания потока управления, например условные предложения и циклы. Для этих средств нужны булевы и реляционные выражения. Язык нуждается в грамматике для описания таких выражений. Автору компилятора требуется как представление булевых значений, так и схема трансляции булевых операций в систему команд целевой машины.

В грамматике символ `∨` обозначает `or`, `∧` – `and`, а `¬` – `not`, чтобы не путать с операциями ILOC.

Грамматические изменения очевидны и показаны на рис. 7.6. Приоритет реляционных операторов выше, чем булевых. Продукции *RelExpr* допускают булевы сочетания реляционных выражений, но запрещают такие каскадные операции, как `a<b<c`. У булевых операторов традиционное предшествование: сначала ¬, затем ∧, потом ∨. Реляционное выражение порождает булево значение.

Язык программирования может определить значения `true` и `false`, например 1 и 0 или –1 и 0.

Представление булевых значений также не вызывает трудностей. Автор компилятора может сопоставить `true` и `false` числовые значения. Как правило, `false` сопоставляет-

ся нуль, а `true` – ненулевое значение. Если значения выбраны удачно, то компилятор сможет для действий с ними воспользоваться командами целевой машины. Большинство процессоров предлагают операции для отрицания, И, ИЛИ и ИСКЛЮЧАЮЩЕГО ИЛИ.

Expr	→ *Expr* ∨ *AndTerm*	*Term*	→ *Term Mults Value*
	| *AndTerm*		| *Value*
AndTerm	→ *AndTerm* ∧ *NotTerm*	*Value*	→ – *Factor*
	| *NotTerm*		| *Factor*
NotTerm	→ ¬ *RelExpr*	*Factor*	→ (*Expr*)
	| *RelExpr*		| num
RelExpr	→ *NExpr Rels NExpr*		| name
	| *NExpr*	*Rels*	→ < | ≤ | = | ≥ | > | _=
NExpr	→ *NExpr Adds Term*	*Adds*	→ + | –
	| *Term*	*Mults*	→ × | ÷

Рис. 7.6 Добавление булевых и реляционных операций в грамматику выражений

Сложности с трансляцией булевых и реляционных выражений возникают из-за различной поддержки, предоставляемой ISA процессора. В разделе 7.4.1 описывается базовая схема трансляции этих выражений, а в разделе 7.4.2 обсуждается, как особенности их поддержки в ISA могут повлиять на трансляцию.

7.4.1. Аппаратная поддержка реляционных выражений

Системы команд процессоров предоставляют несколько механизмов вычисления реляционных выражений. Операции, доступные в ISA целевой машины, определяют, какой код может генерировать компилятор. Поскольку результатом вычисления реляционного выражения обычно является булево значение, эти различия распространяются также на представление и вычисление булевых операций.

Аппаратная поддержка реляционных выражений обычно включает операцию сравнения и набор операций для интерпретации результата этого сравнения. И хотя детали для разных ISA различны, в большинстве применяются похожие схемы. Код должен различать шесть основных отношений: <, ≤, =, ≥, > и ≠.

Предположим ненадолго, что для каждого из этих отношений ISA включает операцию сравнения, которая записывает результирующее булево значение в регистр. Например, `cmp_EQ` r_a, r_b ⇒ r_c устанавливает r_c в `true`, если значения в r_a и r_b равны, и в `false` в противном случае. Для интерпретации этого значения команда условного перехода `cbr` r_c → L_T, L_F передает

```
   if (a < b) then
       statement1
   else statement2
```
Конструкция If-Then-Else

```
    cmp_LT ra,rb ⇒ r1
    cbr    r1 → L1,L2
L1: statement1
    jumpI  → L3
L2: statement2
    jumpI  → L3
L3: nop
```
Схема реализации

```
  cmp_LT ra,rb ⇒ r1
  cmp_GT rc,rd ⇒ r2
  and    r1,r2 ⇒ r3
  cbr    r3 → L1,L2
  L1: statement1
  jumpI  → L3
  L2: statement2
  jumpI  → L3
  L3: nop
```
Вычисление (a < b ∧ c > d)

управление на ассемблерную метку L_T, если r_c содержит true, и на метку LF в противном случае.

Пример во врезке на предыдущей странице показывает, как компилятор мог бы реализовать простую конструкцию if-then-else с помощью этих операций. Предполагается, что значения a и b находятся в регистрах r_a и r_b. Операция сравнения, cmp_LT, записывает булево значение в r_1. Команда условного перехода, cbr, интерпретирует значение в r_1, чтобы выбрать одну из меток L_1 и L_2. После выполнения нужного предложения *statement* программа переходит на следующее предложение по метке L_3.

Компилятор может реализовать булевы операции ∧, ∨ и ¬ по аналогии с арифметическими. Почти все процессоры предоставляют прямые аппаратные реализации этих операций, так что трансляция не вызывает никаких сложностей.

Если бы конструкция исходного языка содержала более сложное условие, например (a < b ∧ c > d), то реализация могла бы выполнить две операции сравнения, объединить их результаты с помощью аппаратной команды and и выполнить условный переход в зависимости от результата, как показано во врезке выше.

Закороченное вычисление

```
cmp_LT ra,rb ⇒ r1
cmp_LT rc,rd ⇒ r2
cmp_LT re,rf ⇒ r3
and    r2,r3 ⇒ r4
or     r1,r4 ⇒ rx
```

Полное вычисление
x ← a < b ∨ c < d ∧ e < f

В булевом выражении значение одного подвыражения может определять значение всего выражения. Во врезке показан ILOC-код, который компилятор мог бы сгенерировать для присваивания

x ← a < b ∨ c < d ∧ e < f

в предположении, что все шесть значений находятся в регистрах. Код вычисляет каждое сравнение, а затем пользуется булевыми операциями для объединения их результатов. В предположении, что команды cmp_LT, and и or занимают один такт, весь код вычисления выражения занимает пять тактов.

Компилятор опирается на два булевых тождества:

∀x, false ∧ x = false
∀x, true ∨ x = true

```
     cmp_LT ra,rb ⇒ rx
     cbr    rx → L3,L1
L1:  cmp_LT rc,rd ⇒ rx
     Cbr    rx → L2,L3
L2:  cmp_LT re,rf ⇒ rx
     jumpI  → L3
L3:  nop
```

Код закороченного вычисления после рефакторинга

Чтобы сгенерировать закороченный код, компилятор должен проанализировать выражение в свете этих двух тождеств и найти минимальный набор условий, определяющий его значение. Во многих случаях закороченное вычисление позволяет уменьшить стоимость вычисления булевых выражений.

Компилятор может подвергнуть код рефакторингу, чтобы использовать минимальное число сравне-

ний, необходимое для определения значения выражения (см. код во врезке). Сначала он вычисляет x ← a < b; если x равно true, то выполняется переход на следующее предложение исходного кода с меткой L_3. С другой стороны, если x равно false, то выполняется переход на L_1, чтобы вычислить x ← c < d.

Если результатом вычисления c < d является false, то x равно false, и производится переход на метку L_3, чтобы выполнить следующее предложение исходного кода. Если в этой точке c < d, то значение x определяется результатом вычисления e < f; чтобы выполнить это последнее вычисление, код совершает безусловный переход на метку L_2.

Длина пути по переработанному коду зависит от значений всех трех сравнений. Если a < b, то выражение равно true, и код выполняет всего одну команду cmp_LT и одну cbr.

Если же a ≥ b, то результат зависит от значений c, d, e и f. Если c ≥ d, то управление передается на метку L_3, где выполняются две операции cmp_LT и две cbr. Наконец, если c < d, то выполняется три cmp_LT, две cbr и одна jump. (Если блок, помеченный L_3, следует сразу после блока, помеченного L_2, то jump нужно исключить, так что останется пять операций, как в оригинальном коде.)

Фактическая стоимость зависит от стоимости отдельных операций. На большинстве машин невыполненная команда перехода занимает один такт, выполненная может занимать больше. Во многих случаях закороченный код быстрее кода полного вычисления.

7.4.2. Варианты аппаратной поддержки

Архитекторы компьютеров придумали много вариантов поддержки вычисления булевых и реляционных выражений. Автор компилятора должен понимать, какую поддержку предоставляет целевая ISA, и в полной мере использовать ее возможности. В этом разделе рассматриваются три типичных варианта поддержки: операции условной пересылки, предикатное выполнение и коды условий.

Операции условной пересылки

Чтобы избежать части ветвей, возникающих при вычислении реляционного выражения, некоторые ISA включают операцию *условной пересылки*. Она принимает два значения и присваивает одно из них результату в зависимости либо от булева значения, либо от кода условия. Если операция условной пересылки принимает явный булев аргумент, то она должна быть «четырехадресной», как команда «умножить и сложить» для чисел с плавающей точкой.

```
if (a < b) then
   x ← c + d
else x ← e + f
```

Пример условной пересылки

```
add    r_c,r_d     ⇒ r_1
add    r_e,r_f     ⇒ r_2
cmp_LT r_a,r_b     ⇒ r_3
c_i2i  r_3,r_1,r_2 ⇒ r_x
```

Соответствующий ILOC-код

Условная пересылка позволяет просто реализовать некоторые конструкции if-then-else, как показано на по-

лях. Компилятор может сгенерировать код, который вычисляет оба сложения и выполняет присваивание только для выбранного случая – либо части `then`, либо части `else`.

Если сложение занимает меньше тактов, чем ветвление, то эта стратегия экономит время выполнения. Если оборудование умеет выполнять несколько команд `add` в одном такте, то хороший планировщик команд сможет избежать платы за дополнительный такт для операции `add`, результат которой не используется.

Предикатное выполнение

Предикатное выполнение – еще одна архитектурная особенность, цель которой – избежать условных переходов. При этом некоторые или все операции могут задать предикатный регистр, определяющий, будет операция выполнена или нет. В некоторых архитектурах операция с предикатом `false` вообще не выполняется, в других выполняется, но ее результат игнорируется. (То есть если предикат равен `false`, то операция `add` не присваивает свое значение, а операция условного перехода не передает управление.)

```
(r1)? add ra,rb ⇒ rx
```

В ILOC предикат имеет вид имени регистра, заключенного в скобки, за которым следует вопросительный знак. Предикатная операция сложения, показанная во врезке, говорит, что r_x принимает значение (r_a + r_b) тогда и только тогда, когда r_1 содержит значение `true`.

```
      cmp_LT ra,rb ⇒ r1
      not    r1    ⇒ r2
(r1)? add    rc,rd ⇒ rx
(r2)? Add    re,rf ⇒ rx
```

ILOC-код, сгенерированный с применением предикатного выполнения

В примере, продемонстрировавшем условную пересылку, схема предикатного выполнения генерирует похожий код, показанный во врезке. И снова архитекторы могут реализовать предикацию двумя способами. Процессор мог бы вычислить все предикатные операции и присвоить значение, только если предикат равен `true`. Или же процессор может проверить предикат и выполнять только те операции, для которых он равен `true`.

Коды условий

```
    comp   ra,rb  ⇒ cc1
    cbr_LT cc1    → L1,L2
L1: loadI  true   ⇒ r1
    jumpI  → L3
L2: loadI  false  ⇒ r1
    jumpI  → L3
L3: nop
```

Вывод булева значения из кода условия

До сих пор мы предполагали, что машинная операция сравнения возвращает булево значение. На практике многие процессоры возвращают более детальную информацию в форме *кода условия*. В таких ISA операция сравнения, например `compare` r_a, r_b, устанавливает отдельные биты в коде условия, показывающие, какое условие истинно: `a < b`, `a = b` или `a > b`. Код условия можно было бы сохранить в отдельном регистре; во многих архитектурах он хранится в слове состояния

процессора. Мы можем рассматривать код условия как неявный аргумент, который некоторые операции определяют или используют.

```
    if (a < b) then
       statement1
    else statement2
```
Конструкция If-Then-Else

```
    comp   ra,rb ⇒ cc1
    cbr_LT cc1 → L1,L2
L1: loadI  true ⇒ r1
    jumpI  →  L3
L2: loadI  false ⇒ r1
    jumpI  →  L3
L3: loadI  true ⇒ r2
    comp   r1,r2 ⇒ cc2
    cbr    cc2 → L4,L5
L4: statement1
    jumpI  → L6
L5: statement2
    jumpI  → L6
L6: nop
```
Наивная трансляция

В схемах с кодом условия необходим оператор условного перехода, который может проверить отдельные биты кода. В ISA, где используются коды условий, вероятно, имеется меньше операторов сравнения и больше операций условного перехода, чем в ISA с булевыми сравнениями, но в целом различия в сложности ISA пренебрежимо малы.

Сложность, однако, проявляется в коде, который должен порождать конкретное булево значение из сравнения, как показано во врезке. Компилятор должен сгенерировать код, в котором для интерпретации кода условия и включения присваиваний на путях `true` и `false` используется условный переход. Операция `comp` записывает свои результаты в регистр кода условия, cc_1; оператор условного перехода проверяет это значение на «меньше». В ветви для < в результат r_1 записывается `true`, а в ветви для ≥ – `false`.

Наивная трансляция реляционных выражений в ISA с кодами условий может порождать неуклюжий код. Во врезке показан код для предыдущего примера `if-then-else`. Он вычисляет `a < b`, получает булево значение и использует его для выбора части `then` или `else`. Код неуклюжий и длинный.

Блоки с метками L_1 и L_4 выполняются, если `a < b`, тогда как блоки с метками L_2 и L_5 – если `a ≥ b`. Человек упростил бы этот код, переместив *statement*$_1$ в блок L_1, а *statement*$_2$ в блок L_2. Компилятор мог бы достичь того же результата путем комбинации распространения констант, перемещения кода и упрощения потока выполнения (см. главу 10). Сочетание оптимизаций перемещает оба предложения, опустошает блоки L_4 и L_5, удаляет эти блоки и переадресует безусловные переходы на метку L_6.

```
    comp    ra, rb ⇒ cc1
    cbr_LT  cc1 → L1,L2
L1: statement1
    jumpI   → L6
L2: statement2
    umpI    → L6
L6: nop
```
Неявное представление для `a < b`

Получающийся код, показанный во врезке, представляет значение `a < b` неявно, положением в коде. Внутри блока L_1 значение равно `true`, а внутри блока L_2 – `false`. Значение бесполезно и недоступно, начиная с метки L_6 и далее.

Компилятор может сгенерировать код, который представляет результат сравнения неявно. Компиляторы, генерирующие код для ISA с кодами условий, часто переключаются между явным и неявным представлениями таких значений. В общем случае компилятор может использовать неявное представление, если в исходном коде переменной не присваивается значение. Этот подход устраняет громоздкий и неуклюжий код, порождаемый при наивной трансляции.

ИТОГИ РАЗДЕЛА

ISA процессора реализуют и интерпретируют сравнения разными способами. Схемы трансляции булевых и реляционных значений отражают это разнообразие. В настоящем разделе была показана базовая схема, а также несколько ее адаптаций для различных видов поддержки со стороны ISA.

Компилятор должен принимать обоснованные решения о том, как отобразить булевы и реляционные выражения на ISA, выбирая между явным и неявным представлениями и пользуясь такими средствами, как условные пересылки и предикация. Внимание к таким деталям повышает эффективность сгенерированного кода.

ВОПРОСЫ

1. В главе 5 было показано место конструкции `if-then-else` в синтаксически управляемой схеме трансляции (см. стр. 242). Напишите набросок синтаксически управляемой схемы, которая генерирует код реляционного выражения, например `a<b`, пользуясь грамматикой на рис. 7.6.
2. Чтобы включить закороченное выполнение в схему трансляции, компилятор должен выбрать порядок вычисления членов выражения. Как компилятор мог бы выбрать порядок, который уменьшает ожидаемое время выполнения?

7.5. Управляющие конструкции

Напомним, что простой блок – это максимальная последовательность линейного кода без предикатов.

Языки программирования предоставляют управляющие конструкции, которые позволяют программисту разными полезными способами соединять линейные последовательности кода, или простые блоки. К числу типичных управляющих конструкций относятся `if-then-else`, циклы и предложения выбора. Для каждой управляющей конструкции компилятору нужна схема трансляции.

В разделе 7.4 была описана трансляция конструкций `if-then-else`. В этой схеме создавалось три метки: одна для блока, начинающего часть `then`, другая для блока, начинающего часть `else`, а третья для блока, который следует за конструкцией `if-then-else`. Компилятор может транслировать конструкцию `if-then-else`, построив представление каждого блока и соединив блоки, как того требуют схемы трансляции `if-then-else`. Все управляющие конструкции обрабатываются компиляторами аналогично: строятся представления блоков, которые соединяются между собой метками и командами условного и безусловного переходов.

Для построения простого блока компилятор агрегирует код последовательных, непомеченных непредикатных операций. Любая передача управления завершает блок. Любое помеченное предложение завершает блок, исходя из предположения, что метка может быть целью условного перехода. Любое предикатное предложение завершает блок. В ILOC каждый блок начинается помеченным предложением и заканчивается командой условного или безусловного перехода. В ПП, поддерживающем неявные ветви, операция, следующая за условным переходом, начинает блок. У него может и не быть начальной метки.

Чтобы связать блоки вместе, компилятор вставляет код реализации управляющей конструкции исходного языка. Во врезке показана исходная конструкция `if-then-else`, а также ILOC-код, следующий за обсуждением в разделе 7.4.1. В примере опущены детали блоков, образующих части `then` и `else`.

```
   if (a < b) then
      block1
   else block2
```
Конструкция If-Then-Else

```
L0: cmp_LT ra,rb ⇒ r1
    cbr    r1 → L1,L2
L1: block1
    jumpI  → L3
L2: block2
    jumpI  → L3
L3: nop
```
Схема реализации

Код, реализующий управляющие конструкции, располагается в конце простых блоков или близко к нему. В общем случае блоки начинаются помеченными предложениями и заканчиваются командами условного или безусловного перехода. Если ПП моделирует слоты отсрочки, то команда перехода может находиться раньше в блоке. В последующих примерах ILOC-кода блоки начинаются меткой. Поскольку в операциях условного перехода в ILOC помечаются и выбранная, и неявная ветви, первая операция после перехода достижима, только если она помечена. Если ПП или целевая ISA допускают относительные переходы, то блок может начинаться непомеченным предложением, являющимся целью относительного перехода.

Хотя в языках используется различный синтаксис для выражения потока управления, базовых концепций немного. В этом разделе рассматривается реализация условных предложений, циклов и предложений выбора.

Предсказание переходов пользователями

Одна байка о компиляторах касается предсказания переходов для конструкций `if-then-else`. В FORTRAN 66 было арифметическое предложение `if`, которое могло пойти по одной из трех ветвей в зависимости от значения управляющего выражения: отрицательное, нуль или положительное. Один из ранних компиляторов позволял пользователю задать веса каждой метки, отражающие относительные вероятности пойти по ней. А компилятор использовал эти веса, чтобы упорядочить ветви и минимизировать полную ожидаемую задержку.

История гласит, что по прошествии года эксплуатации компилятора один из сопровождавших его программистов заметил, что веса ветвей использовались способом, в точности противоположным задуманному, так что ожидаемая задержка максимизировалась. Но никто не жаловался.

Обычно эту байку рассказывают как иллюстрацию ценности мнения программистов о поведении написанного ими кода. (Конечно, никто не сообщил об улучшении, достигнутом после исправления ошибки с весами, – если таковое вообще было.)

7.5.1. Условное выполнение

В предыдущем примере была показана схема реализации конструкций `if-then-else` с помощью сравнений, возвращающих булево значение, и условных переходов. В разделе 7.4 обсуждалось вычисление управляющего выражения. Было показано, как машинная система команд влияет на стратегии обработки управляющего выражения.

Поскольку части `then` и `else` могут содержать сколь угодно много кода, автор компилятора может задействовать разные стратегии трансляции и выбирать ту или иную в зависимости от контекста. Если части `then` и `else` велики, то трехблочная схема из раздела 7.4 работает хорошо. Но если они малы, то компилятору следует рассмотреть использование условных пересылок или предикатного выполнения, чтобы избежать условных и безусловных переходов.

Предикация уменьшает плотность полезных операций в блоке. В какой-то точке отсрочка перехода на несколько тактов оказывается дешевле стоимости тактов, потраченных на бесполезные операции.

Рассмотрим конструкцию `if-then-else`, в которой части `then` и `else` содержат по 20 операций. Код с предикацией заставил бы процессор запустить все 40 операций; со стратегией ветвления было бы запущено 20 операций плюс необходимые условные и безусловные переходы. Для блока достаточно малого размера предикация может оказаться быстрее. По мере роста размер блока достигает точки, в которой стоимость условных и безусловных переходов меньше стоимости запуска неиспользуемых операций.

Та же проблема возникает, когда стратегия основана на операциях условных пересылок. Для небольших блоков использование операций условной пересылки может порождать более эффективный код, чем трехблочная стратегия.

При выборе одной из стратегий – ветвления, предикации и операции условной пересылки – следует учитывать несколько факторов.

1. *Ожидаемая частота выполнения*. Если одна часть конструкции `if-then-else` выполняется значительно чаще другой, то компилятору следует сосредоточиться на методах, повышающих быстродействие на этом пути. Это предпочтение может принимать форму предсказания перехода, спекулятивного выполнения некоторых операций или переупорядочивания логики.
2. *Неравный объем кода*. Если одна часть конструкции `if-then-else` содержит намного больше операций, чем другая, то это может служить аргументом против использования предикации или условных пересылок.
3. *Поток управления внутри конструкции*. Если хотя бы одна из частей конструкции `if-then-else` содержит управляющие конструкции, например еще одно предложение `if-then-else`, предложение выбора, цикл или вызов процедуры, то реализации на базе предикации или условной пересылки выбирать, скорее всего, не стоит.

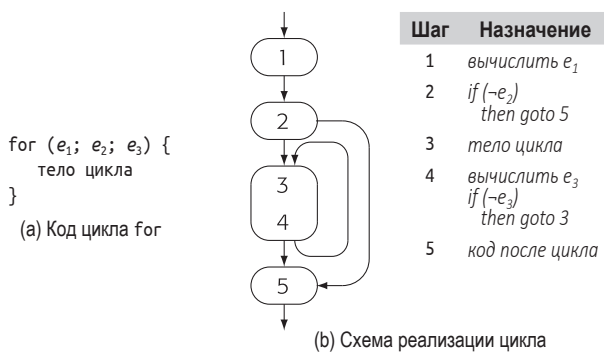
Для принятия наилучшего решения компилятор должен принимать во внимание все эти факторы, а также объемлющий контекст. Из этого наблюдения вытекает, что на практике компилятору может потребоваться (1) построить начальное ПП, представляющее `if-then-else` более абстрактно, (2) про-

анализировать код в этой форме и (3) выбирать ветвление, предикацию или условную пересылку, только имея контекст для обоснования решения.

7.5.2. Циклы и итерация

В большинстве языков программирования имеется одна или несколько конструкций циклов для выполнения итераций. В первом компиляторе FORTRAN для этой цели предлагался цикл `do`. Сегодня циклы встречаются в разных формах. По большей части их структура схожа.

Рассмотрим цикл `for` в языке C, показанный на рис. 7.7(a). На рисунке (b) представлена КСГ, иллюстрирующая, как компилятор мог бы организовать код этого цикла. В цикле имеется три управляющих выражения: e_1, предназначенное для инициализации; e_2, вычисляющее булево значение и направляющее выполнение цикла; e_3, которое выполняется в конце каждой итерации и потенциально модифицирует значения, используемые в e_2. Эта базовая схема применяется к разным видам циклов.



Шаг	Назначение
1	*вычислить* e_1
2	*if* $(\neg e_2)$ *then goto 5*
3	*тело цикла*
4	*вычислить* e_3 *if* $(\neg e_3)$ *then goto 3*
5	*код после цикла*

Рис. 7.7 Общая схема организации цикла `for`

Если тело цикла состоит из одного блока – т. е. никаких других управляющих конструкций в нем нет, – то эта схема порождает цикл с одним начальным условным переходом и одним переходом на каждую итерацию. Некоторые ISA предлагают механизмы, позволяющие скрыть задержку цикла, например определенные пользователем предсказания или слоты отсрочки перехода. Зачастую компиляторы могут использовать такие средства, чтобы замаскировать задержку условного перехода, осуществляющего выход из цикла.

> Если ISA позволяет, то компилятор должен предсказывать переходы, осуществляющие выход из цикла, как явные.

Циклы for

Для трансляции цикла `for` компилятор придерживается общей схемы, показанной на рис. 7.7(b). Рассмотрим пример:

```
for (i=1; i<=100; i++)
{
  тело цикла
}
следующее предложение
```

```
        loadI  1     ⇒ ri       // Шаг 1
        loadI  100   ⇒ r1       // Шаг 2
        cmp_GT ri,r1 ⇒ r2
        cbr    r2    → L2,L1
L1:     тело цикла              // Шаг 3
        addI   ri,1  ⇒ ri       // Шаг 4
        cmp_LE ri,r1 ⇒ r3
        cbr    r3    → L1,L2
L2:     следующее предложение   // Шаг 5
```

Трансляция шагов 1, 2 и 4 очевидна. Она порождает один простой блок для шагов 1 и 2 и еще один для шага 4. Трансляция тела цикла производится предложение за предложением. Если тело цикла состоит из одного простого блока, то компилятор может объединить блоки для шагов 3 и 4, возможно, добившись улучшения кода. Например, планировщик команд мог бы использовать операции в конце шага 3, чтобы заполнить слоты отсрочки в условном переходе на шаге 4.

Цикл DO в FORTRAN

В FORTRAN итеративной конструкцией является цикл `do`. Он не столь гибкий, как цикл `for` в C. В заголовке цикла определяется переменная цикла, начальное и конечное значения. Программист может дополнительно задать шаг приращения. В примере ниже переменная цикла `i` изменяется от 1 до 100 с шагом 1.

```
   do 10 i = 1, 100, 1
      тело цикла
10    continue
   следующее предложение
```

```
        loadI  1     ⇒ ri       // Шаг 1
        loadI  100   ⇒ r1       // Шаг 2
        cmp_GT ri,r1 ⇒ r2
        cbr    r2    → L2,L1
L1:     тело цикла              // Шаг 3
        addI   ri,1  ⇒ ri       // Шаг 4
        cmp_LE ri,r1 ⇒ r3
        cbr    r3    → L1,L2
L2:     следующее предложение   // Шаг 5
```

Комментарии связывают ILOC-код со схемой на рис. 7.7.

Программисты часто передают переменную цикла процедуре, чтобы ее можно было включить в распечатку. В FORTRAN с его параметрами, передаваемыми по ссылке, это действие может ввести компилятор в заблуждение, заставив его поверить, что переменная цикла будет изменена в результате вызова.

В FORTRAN, как и во многих других языках, есть кое-какие интересные особенности. Одна такая странность касается циклов `do` и их переменных. Число итераций цикла фиксируется еще до попадания управления в цикл. Если программа изменяет значение переменной цикла или передает его подпрограмме, которая может изменить ее значение, то такое изменение никак не влияет на число выполняемых итераций. Чтобы обеспечить правильное поведение, компилятор может сгенерировать

скрытую, или *теневую*, *переменную цикла* для управления числом итераций. Что до переменной цикла, передаваемой процедуре, то межпроцедурный сводный анализ может доказать, что значение не изменилось, позволив тем самым компилятору устранить лишние накладные расходы (см. раздел 9.2.4).

Циклы с одной и несколькими проверками

В схеме цикла, показанной на рис. 7.7, последовательность «сравнить и перейти» используется как до начала цикла, так и в конце его тела. Очевидная альтернатива – принять схему, в которой имеется одна проверка в начале цикла и безусловный переход в конце, избежав тем самым второй операции сравнения.

Цикл с одной проверкой создает тело с двумя блоками даже для самых простых циклов. Кроме того, он удлиняет проход по циклу по крайней мере на одну операцию.

Тест с двумя проверками должен порождать тело цикла с одним блоком для циклов, не имеющих внутренних управляющих конструкций. Затем ко всему телу цикла применяются локальная оптимизация и планирование команд. Если нижняя и верхняя границы постоянны, то начальную проверку можно исключить. Короче говоря, цикл с двумя проверками оптимизировать проще, чем с одной.

Циклы while

Цикл `while` также вписывается в стандартную схему цикла. В отличие от цикла `for` в C и цикла `do` в FORTRAN, в цикле `while` нет инициализации. Из-за этого код получается еще более компактным.

```
while (x < y) {
   тело цикла
}
   следующее предложение
```

```
    cmp_GE rx,ry ⇒ r1       // Шаг 2
    cbr    r1 → L2,L1
L1: тело цикла              // Шаг 3
    cmp_LT rx,ry ⇒ r2       // Шаг 4
    cbr    r2 → L1,L2
L2: следующее предложение   // Шаг 5
```

Повтор проверки на шаге 4 дает возможность свести тело несложного цикла к одному простому блоку. Те же преимущества, которые вытекают из этой структуры для цикла `for`, свойственны и циклу `while`.

Циклы until

Цикл `until` продолжается, пока управляющее выражение принимает значение `true`. Управляющее выражение проверяется в конце каждой итерации тела цикла. Так ведет себя цикл `repeat until` в PASCAL.

Цикл `until` всегда выполняет по крайней мере одну итерацию. Его структура особенно проста, потому что шаги 1 и 2 схемы отсутствуют.

```
{
   тело цикла
} until (x < y)
следующее предложение
```

```
L1: тело цикла              // Шаг 3
    cmp_LT rx,ry ⇒ r2       // Шаг 4
    cbr    r2 → L2,L1
L2: следующее предложение   // Шаг 5
```

В C нет цикла `until`. Имеющаяся в нем конструкция `do` похожа на `until`, но только смысл условия противоположный. Цикл продолжается, пока условие равно `false`, тогда как цикл `until` продолжается, пока условие равно `true`.

Выражение итерации в виде хвостовой рекурсии

Хвостовой вызов
Вызов процедуры, являющийся последним действием той же процедуры, называется *хвостовым вызовом*. Рекурсивный хвостовой вызов называется *хвостовой рекурсией*.

В Lisp-подобных языках программисты часто реализуют итерацию как стилизованную форму рекурсии. Если последним действием, выполняемым функцией, является вызов ее самой, то такой вызов называется *хвостовым*. Например, чтобы найти последний элемент списка в SCHEME, можно написать такую простую функцию:

```
(define (last alist)
    (cond
        ((empty? alist) empty)
        ((empty? (cdr alist)) (car alist))
        (else (last (cdr alist)))))
```

Рекурсивный вызов находится в последней строке хвостового вызова.

Коды условий и циклы с обратным отсчетом

Одним из аргументов в пользу ISA с кодами условий является их способность порождать эффективный код в некоторых специфических, но важных ситуациях. В частности, в таких ISA арифметические операции типа `add` и `sub` обычно устанавливают код условия, показывающий соотношение между результатом операции и нулем. И тогда последующие управляющие конструкции иногда могут отказаться от явного сравнения.

При наличии такой возможности компилятор часто может транслировать итеративный цикл с известными заранее границами в цикл с обратным отсчетом, который заканчивается, когда переменная цикла обращается в нуль. (Например, в FORTRAN цикл, нуждающийся в теневой переменной цикла, можно преобразовать в цикл, который уменьшает, а не увеличивает переменную.) Такое преобразование исключает из цикла одну операцию. Если цикл содержит мало операций, то это улучшение может оказаться существенным.

Компиляторы могут генерировать особенно эффективный код для хвостовых вызовов (см. раздел 10.4.1). Итерацию можно выразить в виде хвостовой рекурсии, как показано на примере следующего кода на языке SCHEME:

```
(define (count alist ct)
    (cond
        ((empty? alist) ct)
        (else (count (cdr alist) (+ ct 1)))))

(define (len alist)
    (count alist 0))
```

Вызов len для списка возвращает длину списка. Функция len обращается к count, которая реализует простой счетчик с использованием хвостовых вызовов.

Предложения break

Во многих языках имеется предложение break или exit для структурного выхода из цикла. break передает управление первому предложению, следующему за циклом. Для вложенных циклов break обычно осуществляет выход из самого внутреннего цикла. В некоторых языках, например ADA и JAVA, в break разрешается указывать метку, обозначающую цикл, из которого производится выход. Помеченное предложение break позволяет выйти сразу из нескольких циклов. В языке C break используется также в предложении switch для передачи управления предложению, следующему за switch.

Предложение break реализуется просто. Поскольку циклы и предложения выбора заканчиваются меткой следующего за ними предложения, то компилятору достаточно сгенерировать команду безусловного перехода на эту метку. В некоторых языках имеется предложение skip или continue для перехода к следующей итерации цикла. Эту конструкцию можно реализовать как безусловный переход на код, который повторно вычисляет управляющее выражение и проверяет его значение. Или же компилятор может просто вставить копию вычисления, проверки и условного перехода в точке, где встречается skip.

7.5.3. Предложения выбора

Во многих языках программирования имеется какая-то форма предложения выбора. В FORTRAN есть вычисляемый goto. В ALGOL-W предложение выбора появилось в его современной форме. В BCPL и C имеется конструкция switch. В PL/I есть обобщенное предложение switch; в одном из его вариантов в качестве меток ветвей допускаются логические выражения. Как было отмечено во введении к этой главе, эффективно реализовать предложение выбора трудно.

Рассмотрим реализацию switch в C. Базовая стратегия проста: (1) вычислить управляющее выражение; (2) выполнить условный переход на выбранную ветвь case; (3) выполнить код этой ветви. Шаги 1 и 3 понятны, они уже обсуждались выше.

В C отдельные ветви case обычно заканчиваются предложением break, осуществляющим выход из предложения switch. Если ветвь case заканчивается без break, то управление проваливается в следующую ветвь. Чтобы эта возможность работала, компилятор должен либо (1) сохранять исходный порядок ветвей, либо (2) добавлять явный безусловный переход в конце любой ветви без break, либо (3) клонировать код ветви, куда проваливается исполнение.

Чтобы реализация была эффективной, компилятор должен выбрать хороший метод поиска нужной ветви. Какую ветвь выбрать, неизвестно до момента выполнения, да и на этапе выполнения выбор может всякий раз быть

другим. Таким образом, компилятор должен генерировать код, который будет использовать значение управляющего выражения для поиска соответствующей ветви. Не существует какой-то одной схемы, которая бы хорошо работала во всех случаях. Во многих компиляторах есть возможность генерировать разные схемы и выбирать конкретную в зависимости от контекста.

В этом разделе мы рассмотрим три распространенные стратегии: линейный поиск, вычисляемый адрес и двоичный поиск. Каждая из них применима в определенных – различных – обстоятельствах.

Линейный поиск

Самый простой способ найти нужную ветвь – рассматривать предложение выбора как описание набора вложенных предложений `if-then-else`. Например, предложение `switch` на рис. 7.8(a) можно транслировать в псевдокод на рис. 7.8(b). Эта схема сохраняет семантику `switch`, но стоимость вычисления отдельных ветвей зависит от порядка их записи.

```
switch (e1) {
  case 0: block0;
          break;
  case 1: block1;
          break;
  case 3: block3;
          break;
  default: blockd;
          break;
}
```

(a) Предложение switch

```
t1 ← e1
if (t1 = 0)
    then block0
    else if (t1 = 1)
       then block1
       else if (t1 = 3)
          then block3
          else blockd
```

(B) Реализация на основе линейного поиска

Рис. 7.8 Предложение выбора, реализованное с помощью линейного поиска

При такой стратегии линейного поиска компилятор должен попытаться упорядочить ветви `case` по гипотетической частоте вычислений. И тем не менее когда ветвей немного, скажем три или четыре, эта стратегия может оказаться эффективной.

Прямое вычисление адреса

Таблица переходов
вектор меток, используемый для передачи управления на основе вычисленного индекса таблицы.

Если метки ветвей образуют компактное множество, то компилятору доступны решения, более эффективные, чем линейный поиск. Рассмотрим предложение `switch` на рис. 7.9(a). В нем ветви пронумерованы от 0 до 9 плюс ветвь по умолчанию. В этом случае компилятор может построить вектор, или *таблицу переходов*, которая содержит метки блоков, и найти нужную ветвь по индексу в этой таблице. Таблица переходов показана на рис. 7.9(b), а код вычисления правильной метки – на рис. 7.9(c). В коде поиска предполагается, что таблица переходов начинается по адресу `@T`, а одна метка занимает `@LL` байт.

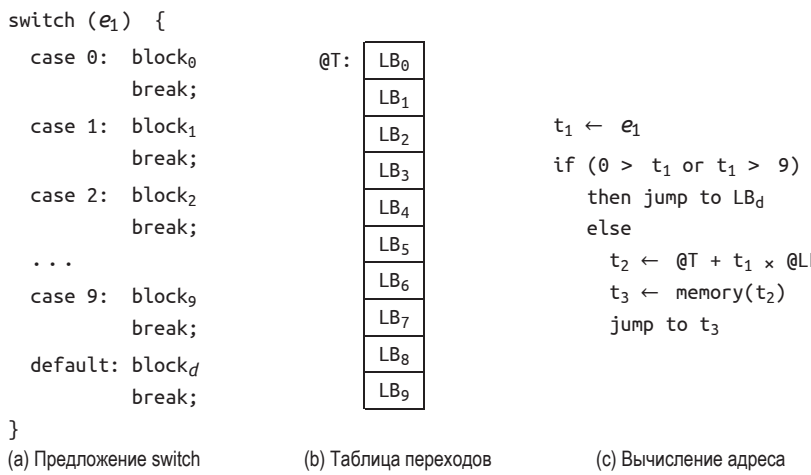


Рис. 7.9 Предложение выбора, реализованное с помощью прямого вычисления адреса

Для плотного набора меток эта схема генерирует компактный и эффективный код. Затраты невелики и постоянны – короткое вычисление, обращение к памяти и безусловный переход. Если в наборе меток есть пропуски, то компилятор может заполнить соответствующие позиции меткой ветви по умолчанию. Если такой ветви не существует, то действие зависит от языка. Например, в C следует перейти на первое предложение после `switch`, поэтому компилятор может поместить его метку во все незаполненные позиции таблицы. Если язык рассматривает отсутствие ветви как ошибку, то компилятор может заполнить пропуски в таблице переходов меткой блока, генерирующего ошибку времени выполнения.

Двоичный поиск

По мере роста числа ветвей линейный поиск становится неэффективным. Аналогично, когда набор меток становится разреженным, размер таблицы переходов ставит препятствия на пути прямого вычисления адреса. Но в этом случае применимы классические решения из теории эффективного поиска. Если компилятор может установить порядок на множестве меток ветвей, то двоичный поиск обеспечит логарифмическую сложность поиска вместо линейной.

Идея проста. Компилятор строит компактную упорядоченную таблицу меток с адресами соответствующих им блоков. Он генерирует код двоичного поиска метки ветви или установления ее отсутствия. И наконец, производится переход на соответствующую метку или ветвь `default`.

На рис. 7.10(а) показано предложение выбора для разреженного набора меток (0, 15, 23, 37, 41, 50, 68, 72, 83, 99) и ветви по умолчанию. Конечно, разброс меток

> Точная форма цикла поиска может изменяться. Например, в коде на рисунке (c) не закорочен случай, когда метка найдена быстро. Для определения наилучшего варианта может потребоваться эмпирическое тестирование.

мог бы быть гораздо больше. На рис. 7.10(b) показана соответствующая таблица поиска, а на рис. 7.10(c) – код двоичного поиска, который компилятор мог бы сгенерировать для поиска нужной ветви и перехода на нее.

```
switch (e1)  {
  case 0:  block0
           break;
  case 15: block15
           break;
  case 23: block23
           break;
  ...
  case 99: block99
           break;
  default: blockd
           break;
}
```

(a) Предложение switch

Значение	Метка
0	LB_0
15	LB_{15}
23	LB_{23}
37	LB_{37}
41	LB_{41}
50	LB_{50}
68	LB_{68}
72	LB_{72}
83	LB_{83}
99	LB_{99}

(b) Таблица поиска

```
t1 ← e1
down ← 0 // нижняя граница
up ← 10  // верхняя граница + 1
while (down + 1 < up)  {
  middle ← (up + down) ÷ 2
  if (Value[middle] ≤ t1)
    then down ← middle
    else up ← middle
}
if (Value [down] = t1
  then jump to Label[down]
  else jump to LBd
```

(c) Код двоичного поиска

Рис. 7.10 Предложение выбора, реализованное с помощью двоичного поиска

Для упрощения последующего анализа компилятору следовало бы зарегистрировать набор целей переходов в ПП (например, с помощью псевдооперации ILOC `tbl`, описанной в приложении A.5). В противном случае метки будут выглядеть как данные, а не код, и на проходах, анализирующих поток управления, например при построении ГПУ, их будет трудно найти (см. раздел 4.6).

ИТОГИ РАЗДЕЛА

В языках программирования много средств для реализации потока управления. Компилятору необходима схема для каждой такой конструкции исходного языка. В некоторых случаях, например для циклов, один подход годится для нескольких конструкций. В других случаях, например предложения `if-then-else` или выбора, компилятор должен выбирать между несколькими схемами реализации. Чтобы выбрать наилучший вариант в сложных случаях, автор компилятора может построить более абстрактное начальное ПП и проанализировать его для установления конкретных фактов, которые нужны компилятору для принятия обоснованного решения. Затем, располагая этими фактами, компилятор может переписать конструкцию с целью реализации выбранной схемы.

ВОПРОСЫ

1. Напишите ILOC-код для цикла FORTRAN во врезке. Напомним, что тело цикла должно быть выполнено 100 раз, хотя значение `i` в нем модифицируется.

```
    do 10 i = 1, 100
       тело цикла
       i = i + 2
10     continue
```

2. Многие языки программирования содержат предложение безусловного перехода, как, например, `goto` в C. Как компилятор мог бы управлять меткой для такого перехода в синтаксически управляемой схеме трансляции?

7.6. Операции со строками

В FORTRAN 66 не было поддержки для строк литер – существенное упущение.

Строки возникают в нескольких контекстах. Чаще всего строки состоят из литер, как принято в большинстве языков, начиная с COBOL. Некоторые языки поддерживают строки с элементами разных типов. Однако структура базовых операций со строками не зависит от типа элементов.

Поддержка операций со строками варьируется: от C, где большинство операций осуществляются библиотечными функциями, до PL/I, где сам язык предоставляет прямые механизмы для присваивания целым строкам, подстрокам и отдельным символам и для конкатенации строк с образованием новых строк. Для иллюстрации проблем трансляции, создаваемых строками, мы в этом разделе обсудим вычисление длины строки, присваивание строке и конкатенацию строк. Другие операции, например подстановка по таблице трансляции, основаны на тех же идеях.

Строковые операции могут обходиться дорого. В старых CISC-архитектурах, например IBM S/370 и DEC VAX-11, имелось специальное оборудование для операций со строками литер. RISC-машины в большей степени полагаются на компилятор, кодирующий эти сложные операции с помощью более простых. Основная операция, копирование байтов из одного места в другое, возникает во многих контекстах.

7.6.1. Длина строки

Программы, работающие со строками, часто должны знать длину строки. Реализация вычисления длины строки зависит от ее представления (см. раздел 5.6.3). Если в строке явно хранится ее длина, то компилятор может просто сгенерировать код, который считает длину из поля с фиксированным смещением от начала строки (обычно отрицательным).

Для строки, завершающейся нулем, программа должна обойти строку и подсчитать ее элементы, как показано во врезке. Псевдокод оставляет число элементов строки в переменной `len`.

```
len ← 0
while (string[len] ≠ '\0')
    len ← len + 1
```

Нахождение длины строки, завершающейся нулем

Вычисление длины иллюстрирует компромисс между двумя представлениями. Версия с явной длиной требует больше памяти, но сложность вычисления длины составляет O(1). Версия, завершающаяся нулем, требует меньше памяти, но сложность составляет O(*длина строки*).

7.6.2. Присваивание строке

В присваивании строке есть два случая: присваивание одной литере и нескольким литерам. Присваивание одной литере прямолинейно. В C присваивание третьей литеры строки `b` второй литере строки `a` можно записать в виде

```
cloadAI   r@b,2 ⇒ r2
cstoreAI  2     ⇒ r@a,1
```
Код присваивания `a[1] = b[2];`

`a[1] = b[2];`. (Напомним, что в C нумерация элементов вектора или строки начинается с нуля.) В ISA с однобайтовыми литерами и байт-ориентированными операциями с памятью (`cload` и `cstore`) это присваивание можно транслировать в простой код, показанный на полях. Предполагается, что $r_{@a}$ и $r_{@и}$ содержат базовые адреса `a` и `b`.

Без аппаратно поддерживаемых операций с памятью на уровне байтов код становится сложнее. Компилятор должен сгенерировать код для загрузки слова, содержащего `b[2]`, и замаскировать остальную часть слова. Далее необходимо загрузить слово, содержащее `a[1]`, и очистить этот байт (снова с наложением подходящей маски с помощью `and`), а затем поместить байт из `b[2]` в позицию очищенного байта (с помощью операции `or`). Наконец, слово, содержащее `a[1]`, нужно записать обратно в память. Сложность этой последовательности операций объясняет, почему так распространены операции сохранения и загрузки отдельных байтов.

Компилятор может сгенерировать код, в котором используются булевы операции с подходящими масками для очистки байтов и копирования байта из одного слова в другое.

```
la ← size_of(a)
lb ← length(b)
if (la < lb)
    then выдать ошибку
i ← 0
while (i < lb) do
   a[i] ← b[i]
   i ← i + 1
```
Присваивание всей строке

Для присваивания нескольким литерам, будь то вся строка или ее часть, компилятор может обернуть код присваивания одной литере циклом. При наличии операций на уровне байтов это просто. Цикл во врезке присваивает всю строку `b` строке `a`. Предполагается, что нумерация литер начинается с нуля и что `size_of(a)` возвращает длину выделенной строке памяти, т. е. ее емкость. Для присваивания подстроке нужно изменить нижнюю и верхнюю границы обеих строк.

Если в ISA нет байт-ориентированных операций, то присваивание нескольким литерам может порождать сложные ситуации. В лучшем случае у источника и приемника выравнивание в памяти одинаково. В худшем случае код должен маскировать и сдвигать каждую литеру для построения слов в строке-приемнике. Эти случаи возникают только в ISA без байт-ориентированных операций с памятью; оставляем их разбор читателю в качестве упражнения.

Чтобы работа с длинными строками, выровненными на границу слова, была эффективной, компилятор может генерировать код с применением команд загрузки и сохранения слова. Концептуально компилятор хочет генерировать байт-ориентированные операции, пока не дойдет до позиции, выровненной на границу слова или двойного слова, затем использовать операции со словами или двойными словами так долго, как получится, а оставшиеся литеры снова обработать байт-ориентированными операциями. Эффект аналогичен разворачиванию цикла (см. раздел 8.5.2) и объединению операций с предциклом для достижения выровненной позиции и постциклом для обработки концевых литер.

Как и в случае присваивания одной литере, код усложняется, если в ISA нет байт-ориентированных операций с памятью. На практике библиотека времени выполнения может включать тщательно оптимизированные функции для реализации нетривиальных случаев (например, функцию `memmove` из стандарта POSIX).

7.6.3. Конкатенация строк

Конкатенация – это просто краткое обозначение последовательности одного или нескольких присваиваний. Существует две основные формы: дописывание строки `b` в конец строки `a` и создание новой строки, содержащей `a`, за которой следует `b`.

В первом случае нужно сначала вычислить длину, а затем выполнить присваивание. Сгенерированный код определяет длину `b`, после чего, если места достаточно, копирует литеры из `b` в память, расположенную непосредственно после содержимого `a`. Во втором случае копируются все литеры `a` и все литеры `b`. По сути дела, компилятор трактует конкатенацию как пару присваиваний.

> Если места недостаточно, то код должен возбудить исключение во время выполнения или еще каким-то способом уведомить об ошибке.

В любом случае компилятор должен убедиться, что выделено достаточно памяти для хранения результата. На практике компилятор или система выполнения должны знать, сколько места выделено каждой строке. Если компилятор знает эти длины, то может выполнить проверку во время генерирования кода и тем самым избежать проверки во время выполнения. Если же длины `a` и `b` компилятору неизвестны, то он должен сгенерировать код их вычисления во время выполнения и выполнить соответствующую проверку и ветвление.

7.6.4. Оптимизация операций со строками

Поскольку операции со строками часто требуют времени O(*длина строки*), то авторам компилятора следует попытаться их оптимизировать. Классический пример задачи оптимизации строк – найти длину результата конкатенации двух строк, `a` и `b`. Если оператором конкатенации является `+`, то соответствующее выражение можно записать как `length(a + b)`.

Для вычисления выражения `length(a + b)` есть две очевидные реализации: (1) построить `a + b` и вычислить ее длину (в C, `strlen(strcat(a,b))`) и (2) сложить длины `a` и `b` (в C, `strlen(a) + strlen(b)`). Второе решение, конечно, более желательно, поскольку так мы избегаем создания временной строки только для того, чтобы вычислить ее длину.

Как обсуждалось в разделе 7.3.3, операции со строками могут включать проверки диапазона, чтобы избежать чтения или записи литер за границами выделенной памяти. Проверки диапазона могут предотвратить логические ошибки вследствие повреждения данных, а также помешать вредоносному

коду прочитать или изменить состояние времени выполнения. В показанном выше примере присваивания строке проверка диапазона разумно вынесена за пределы цикла.

Конечно, для проверки диапазона нужна базовая информация о строке. Чтобы проверить операцию записи, нужно знать размер выделенной строки памяти. Чтобы проверить операцию чтения, нужно также знать текущий размер строки. (Рассмотрим строку, для которой выделено 1024 байта, но в настоящий момент она содержит только литеры `"hello world!"`. Попытка прочитать 128-ю литеру должна приводить к ошибке проверки диапазона.) К строкам применимы те же виды оптимизаций проверки диапазона, что и к массивам.

ИТОГИ РАЗДЕЛА

В принципе, операции со строками похожи на операции с векторами. Детали представления строк и трудности вследствие выравнивания и требований эффективности могут усложнить код, который должен сгенерировать компилятор. Циклы, копирующие по одной литере за раз, просты для генерирования, понимания и доказательства правильности. Более сложные циклы, в которых используются операции со словами и двойными словами, могут оказаться эффективнее, но за это приходится платить дополнительным кодом для обработки граничных случаев. Многие компиляторы обрабатывают сложные случаи, обращаясь к библиотечной функции копирования строк.

ВОПРОСЫ

1. Напишите ILOC-код присваивания строке a ← b с использованием операций загрузки и сохранения слов. (В конце строки используйте операции загрузки и сохранения байтов за пределами цикла.) Можете предполагать, что a и b выровнены на границу слова и не перекрываются.
2. Если обе строки в вопросе 1 выровнены на границу байта, то какие могут возникнуть сложности? И какие сложности могут возникнуть, если строки могут перекрываться?

7.7. Вызовы процедур

Реализация вызовов процедур по большей части прямолинейна. Как показано на рис. 7.11 и обсуждалось в разделе 6.5, вызов состоит из четырех последовательностей: предварительная и заключительная на вызывающей стороне и пролог и эпилог – на вызываемой. Одна процедура может вызываться из нескольких мест, каждое со своими предварительными и заключительными последовательностями. В большинстве языков процедура имеет одну точку входа, т. е. один пролог и один эпилог. Многие детали, связанные с этими последовательностями, описаны в разделе 6.5. Здесь же мы поговорим о проблемах, которые влияют на способность компилятора генерировать эффективный, компактный и единообразный код для вызовов процедур.

В языках, допускающих несколько точек входа, например PL/I, у каждой точки входа имеется свой пролог.

Есть общее правило: перемещение операций из предварительной и заключительной последовательностей в пролог и эпилог должно уменьшать общий размер конечного кода. Если вызов *q* из *p* на рис. 7.11 – единственный вызов *q* во всей программе, то перемещение из предварительной последовательности *p* в пролог *q* (или из заключительной последовательности *p* в эпилог *q*) не влияет на размер кода. Но если *q* вызывается из нескольких мест, то перемещение операции из вызывающей стороны в вызываемую (во всех местах вызова) должно уменьшить общий размер кода благодаря замене нескольких копий операции одной. Когда число точек вызова данной процедуры растет, возрастает и экономия. В общем случае такую оптимизацию можно выполнить только во время проектирования соглашения о вызове или в ситуациях, когда компилятор знает и компилирует все места вызова.

Например, в C статическую функцию можно вызывать только из функций, определенных в том же самом файле.

При условии что адрес функции никогда не берется, компилятор может изменить соглашение о ее вызове.

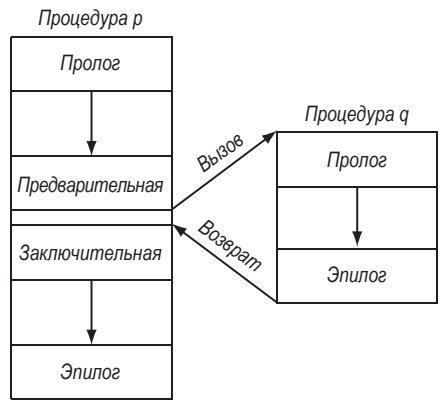


Рис. 7.11 Стандартное соглашение о связях при вызове процедур

Мы предполагаем, что большинство написанных пользователем процедур вызываются из нескольких мест; если нет, то программист вместе с компилятором должны подумать о встраивании процедуры в ее единственное место вызова.

Вызовы процедур в Algol-подобных и объектно-ориентированных языках похожи. Основное различие – способ поиска кода вызываемой стороны (см. раздел 6.3). Кроме того, в объектно-ориентированном языке вызов обычно включает запись объекта получателя в качестве неявного параметра.

7.7.1. Вычисление фактических параметров

Строя предварительную последовательность, компилятор должен сгенерировать код вычисления фактических параметров. Компилятор рассматривает каждый фактический параметр как выражение. Если параметр передается по

значению, то предварительная последовательность вычисляет это выражение и сохраняет его значение в месте, предназначенном для данного параметра, – в регистре или в ЗА вызываемой стороны. Если параметр передается по ссылке, то предварительная последовательность вычисляет параметр как адрес и сохраняет этот адрес в месте, предназначенном для данного параметра. Если для параметра, передаваемого по ссылке, не выделена память, то компилятору, возможно, понадобится выделить в ЗА вызывающей стороны место для хранения значения параметра, чтобы передать его адрес вызываемой стороне.

Если исходный язык определяет порядок вычисления фактических параметров, то компилятор обязан этот порядок реализовать. В противном случае он должен всегда применять один и тот же порядок – либо слева направо, либо справа налево. Порядок вычисления важен, если параметры могут давать побочные эффекты. Например, в программе, где для работы со стеком используются операции `push` и `pop`, результаты выполнения `subtract(pop(), pop())` будут различаться в зависимости от того, как вычисляются параметры: слева направо или справа налево.

Обычно у процедур имеется несколько неявных аргументов, например УЗА процедуры, УЗА вызывающей стороны, адрес возврата и любая информация, необходимая для установления адресуемости. В объектно-ориентированных языках в неявном параметре передается получатель. Одни из этих аргументов передаются в регистрах, другие обычно находятся в памяти. Во многих архитектурах есть операция вида

```
jsr label1
```

которая передает управление на метку $\texttt{label}_1$ и помещает адрес следующей за `jsr` операции в специальный регистр.

Процедуры, передаваемые в качестве фактических параметров, могут потребовать специального обращения. Если p вызывает q, передавая процедуру r в качестве аргумента, то p должна передать q больше информации, чем просто начальный адрес r. В частности, если в откомпилированном коде для поиска нелокальных переменных используются связи доступа, то вызываемой стороне нужно знать лексический уровень r, чтобы последующий вызов r мог найти правильную связь доступа для уровня r. Компилятор может построить пару ⟨*адрес*, *уровень*⟩ и передать ее (или ее адрес) вместо параметра-процедуры. Строя предварительную последовательность для параметра-процедуры, компилятор должен сгенерировать дополнительный код для выборки лексического уровня и соответственной настройки связи доступа.

7.7.2. Сохранение и восстановление регистров

При любом соглашении о вызове вызывающая сторона, вызываемая сторона или та и другая должна сохранить значения регистров. Нередко в соглашениях о вызове часть регистров сохраняется вызывающей стороной, а часть – вы-

зываемой. Поскольку и стоимость операций с памятью, и число регистров подросли, затраты на сохранение и восстановление регистров в месте вызова увеличились до такой степени, что этот вопрос заслуживает особого внимания.

При выборе стратегии сохранения и восстановления регистров автор компилятора должен учитывать как эффективность, так и размер кода. Некоторые особенности процессора оказывают влияние на выбор. Средства выгрузки части набора регистров в память могут уменьшить размер кода. Примерами таких средств могут служить окна регистров на машинах SPARC, операции загрузки и сохранения нескольких слов в архитектурах Power и высокоуровневые операции вызова на DEC VAX-11. Все они предлагают компилятору компактный способ сохранения и восстановления некоторой части набора регистров.

Хотя для больших наборов регистров количество регистров, нуждающихся в сохранении и восстановлении, может увеличиться, в общем случае использование дополнительных регистров повышает быстродействие результирующего кода. Если регистров мало, то компилятор будет вынужден часто генерировать команды загрузки и сохранения; если же регистров много, то многие из таких операций выгрузки производятся только в месте вызова. (Чем больше набор регистров, тем меньше должно быть операций выгрузки в память.) Сосредоточение сохранения и восстановления в местах вызова открывает перед компилятором возможность обрабатывать эти действия более эффективно, чем если бы они были разбросаны по всей процедуре.

- *Использование многорегистровых операций с памятью.* При сохранении и восстановлении соседних регистров компилятор может использовать многорегистровые операции с памятью. Многие ISA поддерживают операции загрузки и сохранения двойного и четверного слов. Их использование может уменьшить размер кода, а также повысить его быстродействие.
- *Использование библиотечных функций.* Вместе с увеличением числа регистров растут и размеры предварительной и заключительной последовательностей. Компилятор может выиграть от наличия специальных функций сохранения и восстановления с упрощенными последовательностями вызова. Будучи применена ко всем вызовам, эта стратегия может заметно уменьшить размер кода. Функции сохранения и восстановления могут принимать аргумент, сообщающий, какие регистры следует сохранять. Возможно, имеет смысл написать оптимизированные версии для типичных случаев, например сохранения всех регистров, сохраняемых вызывающей стороной, или всех регистров, сохраняемых вызываемой стороной.
- *Совместные обязанности.* Чтобы еще уменьшить накладные расходы, компилятор мог бы поручить предварительной последовательности передавать вызываемой стороне значение, указывающее, какие сохраняемые вызывающей стороной регистры сохранять. Затем вызываемая сторона может вызвать принадлежащую компилятору функцию сохранения, чтобы сохранить регистры, сохраняемые вызывающей и вызы-

ваемой сторонами, в одном вызове. Эпилог может передать такую же спецификацию функции восстановления, чтобы та загрузила сохраненные регистры. Этот подход ограничивает накладные расходы одним вызовом для сохранения и еще одним для восстановления. Он отделяет обязанность (сохранение вызывающей или вызываемой стороной) от стоимости вызова функции.

Автор компилятора должен обращать особое внимание на различия в размере и быстродействии кода при описанных выше подходах. Код должен использовать самые быстрые из возможных операций для сохранения и восстановления. Поэтому нужно внимательно изучить стоимость однорегистровых и многорегистровых операций в целевой архитектуре. Использование библиотечных функций для сохранения и восстановления регистров может сэкономить место, а их продуманная реализация может снизить дополнительную стоимость их вызова.

ИТОГИ РАЗДЕЛА

Код вызова процедуры разделен между вызывающей и вызываемой сторонами, а также между прологом, эпилогом, предварительной и заключительной последовательностями. Эти последовательности совместно реализуют соглашение о связях, рассмотренное в главе 6. Правила языка и правила передачи параметров диктуют порядок и способ вычисления фактических параметров. Системные соглашения определяют, кто отвечает за сохранение и восстановление регистров.

Авторы компиляторов уделяют особое внимание реализации вызовов процедур, потому что с помощью общих приемов трудно отыскать возможности оптимизации (см. главы 8 и 10). Тот факт, что между вызывающей и вызываемой сторонами имеется связь типа «многие к одному», осложняет анализ и преобразование, как, впрочем, и распределенная природа кооперативных последовательностей кода. Не менее важно и то, что малейшие отклонения от определенных соглашений могут привести к несовместимости кода, сгенерированного разными компиляторами.

ВОПРОСЫ

1. Когда процедура сохраняет регистры – либо сохраняемые вызываемой стороной в прологе, либо сохраняемые вызывающей стороной в предварительной последовательности, – куда попадают значения, находившиеся в этих регистрах? Сколько места должен зарезервировать компилятор для их хранения?
2. В некоторых ситуациях компилятор должен завести место в памяти, где будет храниться значение параметра, передаваемого по ссылке. Параметры каких видов могут не иметь собственного места хранения? Какие действия могут потребоваться в предварительной и заключительной последовательностях для правильной обработки таких фактических параметров?

7.8. Итоги и перспективы

Одна из самых тонких задач, стоящих перед автором компилятора, – выбрать схему реализации каждой конструкции исходного языка. Почти все конструкции можно реализовать несколькими способами. Выбор, сделанный

автором компилятора на этапе проектирования, окажет сильное влияние на генерируемый компилятором код.

Если компилятор не предназначен для промышленного применения – а значит, производительность откомпилированного кода не так важна, – то автор может выбрать легкие для реализации способы трансляции, порождающие простой код. Если же клиенты ожидают высокой производительности, как в случае оптимизирующего компилятора, то автор должен отдавать предпочтение таким способам трансляции, которые раскрывают как можно больше информации для последующих этапов, – низкоуровневой оптимизации, планированию команд и распределению регистров. Эти две разные точки зрения ведут к разным формам циклов, к разным дисциплинам именования временных переменных и, возможно, к разным порядкам вычисления выражений.

Классический пример этого различия – предложение выбора. В отладочном компиляторе его реализация в виде каскада конструкций `if-then-else` приемлема. Но для оптимизирующего компилятора неэффективность многочисленных проверок и переходов требует более сложной схемы реализации. Постараться улучшить предложение выбора нужно еще на этапе генерирования ПП; немногие оптимизаторы (если такие вообще существуют) будут преобразовывать каскад условных предложений в двоичный поиск или таблицу прямых переходов.

Замечания к этой главе

Материал этой главы можно отнести к двум большим категориям: генерирование кода выражений и обработка управляющих конструкций. Вычисление выражений хорошо исследовано в литературе. Обсуждения вопроса о том, как быть с потоком управления, встречаются реже; значительная часть материала на эту тему, приведенного в данной главе, взята из фольклора, личного опыта и тщательного изучения вывода компиляторов.

Флойд представил первый многопроходный алгоритм генерирования кода из деревьев выражений [160]. Он указывает, что устранение избыточности и алгебраическая реассоциация потенциально способны улучшить результаты алгоритма. Сети и Ульман [321] предложили двухпроходный алгоритм, оптимальный для модели простой машины; Прёбстинг и Фишер дополнили эту работу учетом небольших задержек памяти [299]. Ахо и Джонсон [6] впервые применили динамическое программирование для нахождения реализаций наименьшей стоимости.

Преобладание вычислений с массивами в научных программах привело к работам по выражениям с адресами элементов массива и оптимизациям (например, снижению стоимости операций, раздел 10.7.2), которые их улучшают. Алгебраические преобразования, используемые, чтобы упростить вычисление адресов в многомерных массивах, описаны в работе Скарборо и Кольски [317]. Векторы предварительной информации впервые появились

в системах на базе ALGOL-60; Сэттли описывает их применение для работы с динамически выделенными массивами [316].

Харрисон использовал операции со строками в качестве поясняющего примера при обсуждении встраивания и специализации [192]. Пример с `length(a+b)` в разделе 7.6.4 заимствован из этой статьи.

Мюллер и Уэлли описывают влияние различных форм циклов на производительность [280]. Бернштейн подробно обсуждает варианты, возникающие при генерировании кода для предложений выбора [43]. Самые лучшие описания соглашений о связях и их реализаций приведены в руководствах по процессорам и операционным системам.

Упражнения

К разделу 7.2

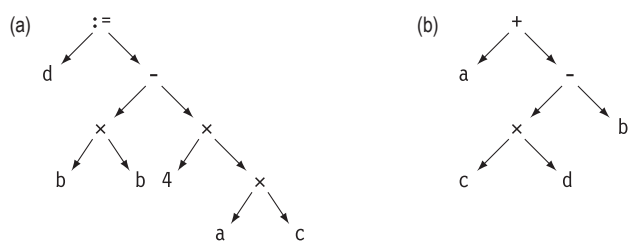
1. Обобщите псевдокод кодогенератора, обходящего дерево, на рис. 7.2, добавив преобразования типов для ×, ÷, + и -. Можете предполагать, что функция `type(node)` возвращает значение перечисления

```
{ Short, Int, Float, Double }
```

и что преобразования описываются таблицей:

	Short	Int	Float	Double
Short	Int	Int	Float	Double
Int	Int	Int	Float	Double
Float	Float	Float	Float	Double
Double	Double	Double	Double	Double

2. Для каждого из показанных ниже деревьев предъявите ILOC-код, который создает кодогенератор, обходящий дерево, на рис. 7.2. Можете предполагать, что множество виртуальных регистров не ограничено.



3. Для деревьев (a) и (b) выше предъявите ILOC-код, который мог бы сгенерировать транслятор, если бы попытался минимизировать использование регистров. Для каждого внутреннего узла укажите, какой его сыновний узел должен вычислять первым.

К разделу 7.3

4. В связи с особенностями реализации часто бывает так, что текстуально одинаковые ссылки в исходной программе влекут разные затраты во время выполнения. Сравните стоимость чтения значения локальной скалярной переменной `x`, когда:
   a) `x` однозначна;
   b) `x` неоднозначна;
   c) `x` – передаваемый по значению параметр текущей процедуры;
   d) `x` – передаваемый по ссылке параметр текущей процедуры;
   e) `x` – статическая переменная.

5. Для массива литер `A[10:12,1:3]`, хранящегося по строкам, напишите ILOC-код, который вычисляет адрес элемента `A[i,j]`. Можете предполагать, что `A` начинается по адресу `@A`. Используйте как можно меньше операций ILOC.

6. Рассмотрите массив `A[1:7,2:9]`, хранящийся по столбцам (как в FORTRAN). Можете предполагать, что `A` начинается по адресу `@A`. Выведите формулу вычисления адреса `A[i,j]` и напишите ILOC-код этого вычисления.

К разделу 7.4

7. Для присваивания $x \leftarrow a > b \wedge b > c$:
   a) напишите ILOC-код его вычисления, используя операции сравнения и условного перехода в ILOC (см. стр. 790).
   b) напишите ILOC-код его вычисления, используя предикатные операции, как показано в разделе 7.4.

8. Рассмотрите простую конструкцию `if-then-else`:

```
if (x < y) then
     блок1;
else блок2;
```

Какие свойства $block_1$ и $block_2$ должен учитывать компилятор при выборе между реализацией на основе (a) сравнения и условного перехода; (b) предикатного выполнения и (c) условной пересылки? Как на это решение влияет задержка перехода?

К разделу 7.5

9. Недавно вы вручную написали сканер. Сканер тратит большую часть времени в цикле `while`, который содержит большое предложение выбора.
   a. Как различные методы реализации предложений выбора будут влиять на эффективность вашего сканера?
   b. Как бы вы изменили свой исходный код, чтобы улучшить производительность, при выборе каждой из стратегий реализации?

10. Преобразуйте следующую функцию C с хвостовой рекурсией в цикл:

```
List * last(List *l) {
  if (l == NULL)
```

```
    return NULL;
else if (l->next == NULL)
    return l;
else
    return last(l->next); }
```

Предъявите С-код этого цикла.

```
do {
  *a++ = *b++;
} while (*b!='\0')
```

(a) Исходный код

```
    loadI  @b    ⇒ r@b
    loadI  @a    ⇒ r@a
    loadI  NULL  ⇒ r1
L1: cload  r@b   ⇒ r2
    cstore r2    ⇒ r@a
    addI   r@b,1 ⇒ r@b
    addI   r@a,1 ⇒ r@a
    cmp_NE r1,r2 ⇒ r4
    cbr    r4    → L1,L2
L2: nop    // следующее предложение
```

(b) Сгенерированный ILOC-код

Рис. 7.12 Пример к упражнению 11

К разделу 7.6

11. Рассмотрите цикл копирования литер на рис. 7.12. Модифицируйте ILOC-код, так чтобы он переходил на обработчик ошибки с меткой L_{sov} при любой попытке выйти за пределы области, выделенной для `a`. Можете предполагать, что длина этой области хранится в виде четырехбайтового целого без знака со смещением –8 от начала `a`.

12. Присваивание строке в случае, когда строки перекрываются, может оказаться довольно сложной задачей.

a. Напишите псевдокод функции `shuffle(a,s,l,o)`, которая копирует `l` литер, начиная с `a[s]`, в позицию `a[s+o]`. При этом `o` может быть как положительным, так и отрицательным.

b. Как бы вы стали проверять диапазон в функции `shuffle`? Можете предполагать, что `length(a)` возвращает число литер в `a`, а `size_of(a)` – выделенный для `a` размер.

К разделу 7.7

13. Рассмотрите однозначную локальную целую переменную *x*. В месте вызова, находящемся в процедуре, где *x* объявлена, она передается по ссылке в виде фактического параметра. Компилятор может хранить *x* в регистре всюду, кроме места вызова. На вызываемой стороне *x* должна иметь адрес в памяти.

a. Где компилятор должен хранить *x*?

b. Как компилятор должен обрабатывать *x* в месте вызова?

c. Как изменились бы ответы, если бы *x* передавалась по значению?

14. Соглашение о связях – это контракт между откомпилированным кодом процедуры и вызывающих ее сторон. Оно описывает интерфейс, кото-

рый вызывающие стороны могут использовать для безопасного вызова процедуры. Таким образом, компилятор вправе нарушить соглашение о связях, только если такое нарушение не видно извне откомпилированного кода.

a. При каких обстоятельствах компилятор может безопасно реализовать связь нестандартно? Приведите примеры из реальных языков программирования.

b. Какие изменения могли бы сделать последовательность вызова более эффективной и что требуется компилятору, чтобы произвести такие изменения?

Глава 8

Введение в оптимизацию

АННОТАЦИЯ

Чтобы улучшить качество кода, генерируемого компилятором, оптимизатор анализирует этот код и переписывает его в более эффективной форме. В этой главе речь пойдет о проблемах и методах оптимизации кода, на примере ряда конкретных оптимизаций будут представлены ключевые концепции. В главе 9 мы продолжим эту тему, более глубоко изучив анализ программы. В главе 10 рассматривается более широкий спектр оптимизирующих преобразований.

В этой главе мы посмотрим, как оптимизация может улучшить код. Обсуждается безопасность и целесообразность оптимизаций, а также возможности для улучшения, которые следует искать автору компилятора. Описываются различные области, в которых выполняется оптимизация, и приводятся по два примера для каждой.

КЛЮЧЕВЫЕ СЛОВА

Оптимизация, безопасность, целесообразность, область оптимизации, анализ, преобразование

8.1. Введение

В фазе анализа исходный код программы транслируется в некоторое промежуточное представление (ПП). В фазе синтеза ПП программы транслируется в форму, допускающую прямое выполнение на целевой машине – аппаратной платформе, например микропроцессоре в бытовом устройстве, или виртуальной машине, например JAVA. Между этими процессами располагается средняя фаза – оптимизатор. Задача оптимизатора – преобразовать ПП, созданное фазой анализа, способом, который улучшает качество кода, порождаемого в фазе синтеза. Нас будет интересовать скалярная оптимизация, т. е. оптимизация однопоточных вычислений. Оптимизация параллельных программ – сложная тема, выходящая за рамки этой книги.

В данной главе описывается предмет оптимизации кода и приводятся примеры различных методов, направленных на устранение разного рода не-

эффективностей и работающих в разных областях кода. В главе 9 некоторые методы анализа программ с целью оптимизации обсуждаются более глубоко. В главе 10 описываются дополнительные преобразования, улучшающие качество кода. Планирование команд (глава 12) и распределение регистров (глава 13) можно трактовать как задачи оптимизации или генерирования кода.

Концептуальная дорожная карта

Цель оптимизации кода – получить на этапе компиляции информацию о поведении программы во время выполнения и использовать ее для улучшения сгенерированного кода. Улучшение может принимать разные формы. Самая частая цель оптимизации – ускорить работу кода. Но в некоторых приложениях размер откомпилированного кода важнее его быстродействия; взять, к примеру, приложение, которое будет загружено в память, допускающую только чтение, когда размер кода напрямую влияет на стоимость системы. Возможны и другие цели оптимизации: сокращение энергопотребления, повышение скорости реакции на события реального времени и уменьшение трафика в памяти.

В оптимизаторах применяется много методов улучшения кода. Обсуждение оптимизации должно включать неэффективности, которые можно исправить, и применяемые для этого способы. Для каждого источника неэффективности автор компилятора должен выбрать одну из нескольких техник ее устранения; у каждой техники есть свои плюсы и минусы. Автор компилятора должен понимать, как в этих техниках используются возможности для улучшения, как они гарантируют безопасность переписывания кода и почему переписанный код оказывается эффективнее. Все эти вопросы существенны для успешной оптимизации кода.

Краткий обзор

Возможности для оптимизации проистекают из нескольких источников, но главный – реализация абстракций исходного языка. Поскольку трансляция исходного кода в ПП – локальный процесс, не подразумевающий тщательного анализа окружающего контекста, обычно генерируется ПП, реализующее наиболее общий случай конструкции. Имея дополнительный контекст, оптимизатор зачастую может установить, что такая общность не нужна; в таком случае код можно переписать в менее общей, но более эффективной форме.

Второй важный источник оптимизации – сама целевая машина. Компилятор должен хорошо понимать свойства целевой машины, влияющие на производительность. На форму генерируемого компилятором кода влияют такие детали, как число и возможности функциональных блоков, задержки и полоса пропускания различных уровней иерархии памяти, различные режимы адресации, поддерживаемые системой команд, и наличие необычных или сложных операций.

Безопасность
Преобразование безопасно, если оно не изменяет результаты исполняемой программы.

Целесообразность
Применение преобразования в некоторой точке целесообразно, если оно действительно дает улучшение.

Оптимизация – обширный и разветвленный предмет, изучению которого можно посвятить целые курсы и книги. В этой главе предлагается только введение в него. В разделе 8.2 изучаются понятия *безопасности* и *целесообразности*; оптимизацию следует применять, только если она безопасна и предположительно целесообразна. В разделе 8.3 описываются различные уровни детальности, или области, в которых производится оптимизация. В оставшейся части главы приводятся примеры, иллюстрирующие различные источники улучшений и области оптимизации. В этой главе нет раздела «Дополнительные темы», эту функцию исполняют главы 9 и 10. Глава 9 посвящена статическому анализу, в ней описываются виды аналитических задач, с которыми сталкивается оптимизирующий компилятор, и практические методы их решения. В главе 10 систематически рассматриваются скалярные оптимизации, ориентированные на одиночный процессор.

Несколько слов о времени

Глава 8 – введение в предмет оптимизации кода: анализ и преобразование. Оптимизации, описанные в этой главе, производятся целиком на *этапе компиляции*. Концептуально любая оптимизация читает ПП программы, анализирует ее, преобразует и записывает.

Чтобы решить, стоит или не стоит применять оптимизацию в конкретной точке компилируемого кода, компилятор должен проанализировать код. По сути дела, компилятор рассуждает о потоке значений и управления во время выполнения. Эту информацию он использует, чтобы решить, будет ли безопасно применить преобразование и насколько вероятно, что преобразование улучшит код. Анализ на этапе компиляции поддерживает принятие решение на этапе компиляции с целью улучшить поведение на этапе выполнения.

8.2. Предварительные сведения

До начала 1980-х годов многие авторы компиляторов считали, что оптимизацию следует добавлять только после того, как прочие части компилятора отлажены. Это мнение породило различие между *отладочными* и *оптимизирующими компиляторами*. Отладочный компилятор ставил во главу угла быструю компиляцию, пусть даже ценой низкого качества кода. Такие компиляторы производили простую трансляцию, так что откомпилированный код сохранял сильное сходство с исходным. При таком структурном сходстве было легко сопоставить ошибку с конкретной строкой исходного кода; отсюда и термин «*отладочный*» компилятор. С другой стороны, оптимизирующий компилятор призван улучшить время работы исполняемого кода за счет уве-

личенного времени компиляции. Потратив больше времени на компиляцию, можно получить улучшенный код. Однако оптимизация затрудняет сопоставление исходного кода с исполняемым, так что отладка тоже усложняется.

Оптимизация, анализ и преобразование

В сообществе разработки компиляторов три термина – *оптимизация*, *анализ* и *преобразование* – часто употребляются как синонимы. Но мы будем проводить различие между этими не вполне идентичными понятиями.

Под *оптимизацией* понимается общая схема повышения производительности программы, например перемещение кода. Имеется много алгоритмов перемещения кода. Они различаются способами идентификации кода, допускающего перемещение, и выбором места назначения.

Под *анализом* понимаются методы времени компиляции, которые извлекают знания о поведении откомпилированного кода на этапе выполнения. В процессе оптимизации кода такой анализ устанавливает, когда переписывание кода безопасно и потенциально целесообразно.

Под *преобразованием* понимаются методы времени компиляции, которые позволяют переписать код для достижения целей оптимизации; обычно они основаны на результатах какого-то анализа.

В некоторых простых контекстах оптимизацию можно реализовать в виде одного прохода, объединяющего анализ и преобразование. В тех же контекстах, где встречается сложный поток управления, анализ, как правило, должен быть завершен до начала преобразования.

По мере того как на рынок стали выходить RISC-процессоры, а методы, присущие RISC, стали применяться к CISC-архитектурам, бремя повышения производительности легло на плечи компиляторов. Для повышения производительности архитекторы процессоров обратились к средствам, требовавшим больше поддержки со стороны компиляторов. К ним относятся слоты отсрочки переходов, неблокирующие операции с памятью, более глубокие конвейеры и увеличение количества функциональных блоков. Из-за этих средств производительность процессоров становится более чувствительной к высокоуровневым вопросам размещения программ в памяти и их структуры, а также к низкоуровневым деталям планирования команд и распределения ресурсов. Вследствие увеличения разрыва между быстродействием процессора и производительностью приложений оптимизация стала обязательной частью каждого компилятора.

В свою очередь, рутинное включение оптимизатора изменяет среду, в которой работают фазы анализа и синтеза. Оптимизация еще сильнее изолирует фазу анализа от проблем производительности, что упрощает задачу трансляции в фазе анализа. В то же время оптимизация изменяет код, обрабатываемый фазой синтеза. Современные оптимизаторы предполагают, что фаза синтеза выполнит распределение ресурсов; таким образом, они обычно имеют дело с идеализированной машиной, располагающей неограниченным запасом регистров, памяти и функциональных блоков. А это,

в свою очередь, дает им больше свободы и возлагает большую ответственность на фазу синтеза.

Если компилятор согласен принять свою долю ответственности за производительность во время выполнения, то должен включать оптимизатор. Как мы увидим, инструменты оптимизации также играют большую роль в фазе синтеза. По этим причинам так важно познакомиться с оптимизацией и изучить некоторые возникающие при этом проблемы, прежде чем обсуждать методы, используемые в фазе синтеза компилятора.

8.2.1. Примеры

Для придания обсуждению конкретики мы подробно рассмотрим два примера. Первый, простое вычисление адреса элемента двумерного массива, показывает роль знаний и контекста для оптимизации кода. Второй, вложенные циклы в подпрограмме `dmxpy` из широко используемой библиотеки численных расчетов LINPACK, дает представление о самом процессе преобразования и проблемах, которые преобразованный код может создать компилятору.

Улучшение вычисления адреса элемента массива

Метод доступа к массиву, хранимому по строкам, описан в разделе 7.3.2.

Рассмотрим ПП, которое фаза анализа могла бы сгенерировать для ссылки на элемент массива `m(i,j)` в языке FORTRAN. Ничего не зная о значениях `m`, `i` и `j`, компилятор должен сгенерировать вычисление по формуле полного адреса для двумерного массива, хранимого по столбцам:

$$@m + (j - low_2) \times (high_1 - low_1 + 1) \times w + (i - low_1) \times w,$$

В обсуждении предполагается, что компилятор не преобразовал `@m` в форму со сдвинутым нулем $@m_0$.

где `@m` – адрес первого элемента `m` во время выполнения, low_i и $high_i$ – нижняя и верхняя границы по i-му измерению `m`, а w – размер элемента `m`. Способность компилятора уменьшить стоимость этого вычисления зависит от его способности анализировать код и окружающий его контекст.

Снижение стоимости операций преобразование, заключающееся в замене последовательности операций вида

$$i \cdot c, (i+1) \cdot c, \ldots, (i+k) \cdot c$$

эквивалентной последовательностью

$$i'_0, i'_1, \ldots, i'_k,$$

где $i'_0 = i \cdot c$ и $i'_j = i'_{j-1} + c$.

См. раздел 10.7.2.

Если `m` – локальный массив с $low_1 = low_2 = 1$ и значение $high_1$ известно, то компилятор может упростить вычисление до

$$@m + (j - 1) \times hw + (i - 1) \times w,$$

где $hw = high_1 \times w$. Если ссылка встречается в цикле, где `j` изменяется от 1 до k, то компилятор может использовать метод *снижения стоимости операций*, чтобы заменить член $(j - 1) \times hw$ последовательностью $j'_1, j'_2,$

..., j'_k, где $j'_1 = (1 - 1) \times hw = 0$ и $j'_x = j'_{x-1} + hw$. Если i также является переменной цикла и изменяется от 1 до l, то снижение стоимости операций может заменить (i − 1) × w последовательностью $i'_1, i'_2, \ldots, i'_l$, где $i'_1 = 0$ и $i'_x = i'_{x-1} + w$. После этих изменений вычисление адреса сводится просто к

```
@m + j' + i'.
```

Перемещение кода
перемещение вычисления в точку, где оно выполняется не так часто.

При подходящих обстоятельствах сложение в @m + j' можно вынести из внутреннего цикла (см. разделы 10.3.1 и 10.7.2).

После этих преобразований цикл по j должен увеличить j' на *hw*, а цикл по i – увеличить i' на *w*. В такой форме вычисление адреса сводится к увеличению до i' и, быть может, сложению во внутреннем цикле и увеличению до j' и сложению во внешнем цикле. Знание контекста, окружающего ссылку на m(i,j), позволяет компилятору значительно сократить стоимость вычисления адресного выражения элемента массива.

Если m передается в виде фактического параметра, то компилятор может не знать его верхней и нижней границы на этапе компиляции. На самом деле размерности m могут быть разными при разных вызовах процедуры. В таких случаях компилятор, возможно, не сумеет упростить вычисление адреса, и это еще один пример того, какую роль играет контекст с точки зрения производительности.

Улучшение вложенного цикла в LINPACK

В качестве еще более убедительного примера контекста рассмотрим вложенный цикл на рис. 8.1. Это центральный цикл в подпрограмме dmxpy из библиотеки численных расчетов LINPACK, написанной на FORTRAN. Код содержит два цикла, обертывающих одно длинное присваивание. Этот вложенный цикл лежит в основе процедуры вычисления *mx* + *y* для матрицы *m* и векторов *x* и *y*. Мы рассмотрим код с двух разных точек зрения: во-первых, преобразования, которое автор применил вручную для повышения производительности, а во-вторых, проблем, с которыми компилятор сталкивается при попытке трансляции этого цикла, так чтобы он эффективно выполнялся на конкретном процессоре.

До ручного применения автором преобразования вложенный цикл, выполняющий то же вычисление, был проще:

```
   do 60 j = 1, n2
      do 50 i = 1, n1
         y(i) = y(i) + x(j) * m(i,j)
50    continue
60 continue
```

Разворачивание цикла
При разворачивании тело цикла заменяется несколькими итерациями и соответствующим изменением индекса.

Чтобы повысить производительность, автор *развернул* 16 итераций цикла по j. При

этом создается 16 копий предложения присваивания с разными значениями индекса, от j до j-15, а переменная внешнего цикла изменяется сразу на 16. Далее автор объединил 16 присваиваний в одно предложение, исключив 15 повторений y(i) = y(i) + ...; это устранило 15 сложений и большинство операций загрузки и сохранения y(i). Разворачивание цикла устраняет 15 условных переходов и некоторые другие операции. Также оно может улучшить локальность кеша.

Для обработки случаев, когда границы массива не являются целыми кратными 16, в полной процедуре имеется четыре версии вложенного цикла, предшествующие показанной на рис. 8.1. Эти «предварительные циклы» обрабатывают до 15 столбцов m, после чего значение j таково, что n2 - j делится на 16. В первом цикле обрабатывается один столбец m, что соответствует нечетному n2. В остальных трех вложенных циклах обрабатываются два, четыре и восемь столбцов m. После предварительных циклов можно выполнить последний вложенный цикл, показанный на рис. 8.1, в котором за один раз обрабатывается 16 столбцов.

```
      subroutine dmxpy (n1, y, n2, ldm, x, m)
      double precision y(*), x(*), m(ldm,*)
        ...
      jmin = j+16
      do 60 j = jmin, n2, 16
          do 50 i = 1, n1
             y(i) = ((((((((((((((( (y(i))
     $             + x(j-15)*m(i,j-15)) + x(j-14)*m(i,j-14))
     $             + x(j-13)*m(i,j-13)) + x(j-12)*m(i,j-12))
     $             + x(j-11)*m(i,j-11)) + x(j-10)*m(i,j-10))
     $             + x(j- 9)*m(i,j- 9)) + x(j- 8)*m(i,j- 8))
     $             + x(j- 7)*m(i,j- 7)) + x(j- 6)*m(i,j- 6))
     $             + x(j- 5)*m(i,j- 5)) + x(j- 4)*m(i,j- 4))
     $             + x(j- 3)*m(i,j- 3)) + x(j- 2)*m(i,j- 2))
     $             + x(j- 1)*m(i,j- 1)) + x(j) *m(i,j)
   50        continue
   60 continue
        ...
      end
```

Рис. 8.1 Фрагмент подпрограммы dmxpy из LINPACK

В идеале компилятор должен автоматически преобразовать оригинальный вложенный цикл в эту более эффективную версию или иную, более подходящую для целевой машины. Однако лишь немногие компиляторы включают все оптимизации, необходимые для этой цели. В случае dmxpy автор решил выполнить оптимизации вручную, чтобы добиться хорошей производительности для как можно более широкого круга целевых машин и компиляторов.

С точки зрения компилятора, отображение вложенного цикла на рис. 8.1 на целевую машину представляет ряд трудных задач. Вложенный цикл содержит 33 выражения адреса элемента массива: 16 для m, 16 для x и одно для y.

Если компилятор не сумеет упростить эти вычисления адресов, то цикл погрязнет в целочисленной арифметике.

Предполагается, что нижняя граница любого массива равна 1, как принято в FORTRAN.

Рассмотрим ссылки на x. Они не изменяются во внутреннем цикле по i. Оптимизатор может вынести вычисления адреса и операции загрузки x из внутреннего цикла. Если ему удастся хранить значения x в регистрах, то значительная часть накладных расходов во внутреннем цикле будет исключена. Для таких ссылок, как x(j-12), вычисление адреса сведется к @x + (j – 12 – 1)×*w*. С целью дальнейшего упрощения компилятор может переписать все 16 ссылок на x в форме @x + *jw* – c_k, где *jw* – это j×*w*, а $c_k = (k+1) \times w$ для всех $0 \le k \le 15$. В такой форме все операции загрузки имеют один и тот же базовый адрес @x + *jw*, а изменяется только смещение c_k.

Для эффективного отображения этого вычисления на целевую машину нужно знать, какие имеются режимы адресации. Если целевая машина поддерживает операцию load с непосредственным адресным операндом (например, loadAI), то операции доступа к x можно переписать с одной переменной цикла, начинающейся с @x + jmin × *w*; в цикле по j она увеличивается на 16×*w*.

На каждой итерации внутреннего цикла производятся обращения к различным элементам m. Следовательно, во внутреннем цикле необходимо вычислять новые адреса и загружать новые значения каждого из 16 элементов m на каждой итерации. Тщательный рефакторинг адресных выражений вкупе с уменьшением сложности операций может сократить накладные расходы на доступ к m. Компилятор может организовать вычисление адреса m(1,j) в цикле по j. Затем он может использовать различные постоянные смещения в каждой из 15 других операций загрузки.

Чтобы добиться такой формы кода, компилятор должен переписать адресные выражения, произвести уменьшение сложности операций, распознать вычисления, инвариантные относительно цикла, и вынести их из внутреннего цикла, а также выбрать подходящий адресный режим для операций загрузки. И даже после всех этих улучшений во внутреннем цикле нужно будет выполнить 17 загрузок, 16 умножений с плавающей точкой, 16 сложений с плавающей точкой и одну операцию сохранения. Получившийся блок кода будет представлять сложности как для планировщика команд, так и для распределителя регистров.

Ограниченный памятью
Цикл, в котором на операции загрузки и сохранения приходится больше тактов, чем на вычисления, называют *ограниченным памятью*.

Чтобы определить, ограничен ли цикл памятью, необходимы детальные знания о цикле и целевой машине.

Если у компилятора что-то не получается в какой-то части этой последовательности преобразований, то результирующий код может оказаться значительно хуже оригинала. Например, если компилятор не сможет привести адресные выражения для x и m к общим базовым адресам, то потребуется 33 разных переменных цикла – по одной для каждой позиции в x, m и y. Если из-за возникшего при этом спроса на регистры

распределитель регистров будет вынужден выгружать регистры в память, то количество операций загрузки и сохранения окажется больше, чем в ограниченном памятью цикле. В таких случаях качество порожденного компилятором кода зависит от дружной работы всей последовательности преобразований; если хотя бы одно не достигает поставленной цели, то качество кода будет хуже, чем ожидает пользователь.

8.2.2. Соображения по поводу оптимизации

В подпрограмме `dmxpy` программист применил преобразования вручную в надежде, что программа станет работать быстрее (так и получилось). Кроме того, программист должен был точно знать, что эти изменения сохранят семантику программы. Безопасность, целесообразность и риск играют важнейшую роль для решения компилятора применить некоторое преобразование в конкретном месте кода.

Определение безопасности

Корректность – самое важное условие, которое обязан соблюсти компилятор, – код, порождаемый компилятором, должен сохранять семантику входной программы. После применения любого преобразования корректность трансляции должна быть гарантирована.

Обычно под *семантикой* понимается наблюдаемое поведение программы, т. е. состояние ее памяти в момент, непосредственно предшествующий остановке, плюс весь сгенерированный вывод. Если программа завершается, то значения всех видимых переменных непосредственно перед остановкой должны быть одинаковы при любой схеме трансляции. Поведение интерактивной программы сложнее, и определить его труднее.

Плоткин формализовал эту идею, введя понятие *наблюдаемой эквивалентности*:

> *Два выражения, M и N, наблюдаемо эквивалентны тогда и только тогда, когда в любом контексте C, где M и N замкнуты (т. е. не имеют свободных переменных), вычисление C[M] и C[N] либо дает одинаковые результаты, либо никогда не завершается* [295].

Таким образом, два выражения наблюдаемо эквивалентны, если они одинаково влияют на видимое внешнее окружение.

На практике компиляторы пользуются более простым и менее обязывающим понятием эквивалентности, а именно если в фактическом программном контексте два разных выражения *M* и *N* дают одинаковые результаты, то компилятор может подставить *N* вместо *M*. При таком определении приходится иметь дело только с контекстами, реально возникающими в программе. Подстраивание кода под контекст – это и есть суть оптимизации. Ничего не говорится о том, что происходит, когда вычисление идет не по плану или не доходит до конца.

На практике компиляторы принимают меры, чтобы избежать *расхождения*, т. е. не порождают код, который зацикливается или возбуждает исключение, например в результате деления на ноль. Противоположный случай, когда исходный код может разойтись, а оптимизированный – нет, упоминается редко.

Компилятору необходим механизм для доказательства того, что любое применение преобразования безопасно, т. е. что преобразование сохраняет семантику программы. У компилятора должны быть основания верить в то, что применение преобразования целесообразно, т. е. улучшает производительность программы. Наконец, компилятор должен понимать, как переписывание может ухудшить код, т. е. каков риск нецелесообразности преобразования. Автор компилятора должен принимать во внимание все три фактора, когда проектирует и реализует механизмы принятия решений о преобразовании кода.

Безопасность

Откуда программист LINPACK знал, что преобразования безопасны? То есть почему он верил, что преобразованный код даст те же результаты, что исходный? Пристальное изучение вложенного цикла показывает, что единственное взаимодействие между последовательными итерациями происходит с помощью элементов `y`.

- Значение, вычисленное как `y(i)`, не используется повторно до следующей итерации внешнего цикла. Итерации внутреннего цикла не зависят друг от друга, поскольку на каждой итерации определяется ровно одно значение и никакая другая итерация внутреннего цикла на это значение не ссылается. Следовательно, итерации можно выполнять в любом порядке. (Например, изменение порядка итераций внутреннего цикла на противоположный дало бы точно такие же результаты.)
- Эффект взаимодействия через `y` ограничен. В *i*-м элементе `y` накапливается сумма по всем *i*-м итерациям внутреннего цикла. В развернутом цикле это свойство сохранено.

Значительная часть анализа, выполняемого в ходе оптимизации, имеет целью доказать безопасность преобразований.

Целесообразность

Почему программист LINPACK считал, что разворачивание цикла улучшит производительность? То есть почему это преобразование целесообразно? На скорость работы кода разворачивание влияет несколькими способами.

- Количество итераций цикла уменьшается в 16 раз, что уменьшает число операций, необходимых для управления циклом: сложений, условных и безусловных переходов. Экономия может оказаться немалой.

  Этот эффект может навести на мысль о разворачивании с еще большим коэффициентом. Но конечность ресурсов заставила остановиться на 16. (См. обсуждение *спроса на регистры* ниже.)
- Вычисления адресов (= элементов) массива содержат дублирование. Рассмотрим использование `y(i)`. В исходном коде адрес `y(i)` вычислялся один раз на каждое умножение `x` и `m`. В преобразованном коде он вычисляется один раз на 16 умножений, т. е. объем работы уменьшился

в 16 раз. Все 16 ссылок на `m` и в меньшей степени на `x` также включают части, которые можно было бы вычислить в цикле один раз и затем использовать повторно.

В рассуждении предполагается, что `x(j)` находится в регистре.

- В преобразованном цикле выполняется больше операций с плавающей точкой на одну операцию с памятью. В исходном цикле было две операции с плавающей точкой на три операции с памятью, а в развернутом – 32 операции с плавающей точкой на 18 операций с памятью в предположении, что все значения `x` находятся в регистрах. Таким образом, меньше шансов, что развернутый цикл будет ограничен памятью. В нем достаточно независимых арифметических операций, которые могут перекрываться с операциями загрузки и частично компенсировать их задержки.

Если размер внутреннего цикла оказывается больше кеша кода, то производительность может пострадать.

Разворачивание может иметь и другие машинно-зависимые эффекты. Оно увеличивает объем кода во внутреннем цикле, что может дать планировщику команд больше возможностей для компенсации задержек. Если задержка операции условного перехода в конце цикла велика, то более длинное тело цикло может позволить компилятору заполнить больше слотов отсрочки перехода. В некоторых процессорах неиспользуемые слоты отсрочки должны быть заполнены командами `nop`, и в этом случае разворачивание цикла может уменьшить число выборок `nop`, сократить трафик в памяти и, быть может, снизить энергопотребление программы.

Риск

Если преобразования, призванные улучшить производительность, могут затруднить компилятору генерирование хорошего кода, то следует учесть это при решении о целесообразности. Так, ручные преобразования в `dmxpy` создали компилятору следующие проблемы.

- *Спрос на регистры.* В исходном цикле требовалось совсем немного регистров для хранения активных значений. Нужны были регистры с плавающей точкой для `y(i)`, `x(j)` и `m(i,j)`. Нужны были регистры общего назначения для вычисления адресов элементом `x`, `y` и `m`. Для переменных цикла нужны были регистры, сохраняемые между итерациями. С другой стороны, в преобразованном цикле нужны регистры с плавающей точкой для всех 16 элементов `x`, 16 значений `m` и одного значения `y(i)`, а также все прежние регистры общего назначения.
- *Форма вычисления адреса.* В исходном цикле мы имели дело с тремя адресами, по одному для `y`, `x` и `m`. Поскольку в преобразованном цикле на каждой итерации используется больше ссылок, компилятор должен аккуратно выбрать форму вычислений адресов, чтобы избежать повторных вычислений и избыточного спроса на регистры. В худшем

случае коду придется вычислять полные адреса всех 16 элементов `x`, всех 16 элементов `m` и одного элемента `y`.

Если компилятор тщательно подойдет к выбору формы вычисления адресов, то сможет использовать единственный базовый адрес, или указатель, для `m` и еще один для `x`, и для каждого из них 16 постоянных смещений. Он сможет переписать цикл, так чтобы использовать один из этих указателей в проверке в конце цикла, устранив необходимость в еще одном регистре и исключив одно обновление (см. раздел 10.7.2). Планирование и оптимизация меняют всё.

Возможны и другие машинно-зависимые проблемы. Например, 17 загрузок и одно сохранение, 16 умножений, 16 сложений плюс вычисления адресов и операции управления циклом на каждой итерации необходимо тщательно планировать. Возможно, компилятору придется прибегнуть к организации программного конвейера для планирования некоторых операций загрузки, так чтобы они выполнялись на предыдущей итерации и начальные арифметические операции на каждой итерации отрабатывались своевременно (см. раздел 12.5).

8.2.3. Возможности для оптимизации

Как мы видели, задача оптимизации простого цикла может требовать учета сложных соображений. В общем случае оптимизирующие компиляторы пользуются возможностями, проистекающими из разных источников.

1. *Уменьшение накладных расходов из-за абстрагирования.* Как мы видели на примере `dmxpy`, структуры данных и типы, определяемые языками программирования, требуют вычисления адресов на этапе выполнения. Оптимизаторы применяют анализ и преобразования для снижения этих накладных расходов.
2. *Использование преимуществ частных случаев.* Часто компилятор может использовать знания о контексте, в котором выполняется операция, для специализации этой операции. Например, компилятор C++ иногда может определить, что при всех вызовах виртуальной функции используется одна и та же реализация, и в таком случае виртуальный вызов можно преобразовать в менее дорогой статический вызов.
3. *Сопоставление кода с системными ресурсами.* Если требования программы к ресурсам отличаются от располагаемых процессором, то компилятор может преобразовать код, приблизив его к имеющимся ресурсам. У преобразований, примененных в `dmxpy`, такой эффект имеется; они уменьшают количество операций доступа к памяти в расчете на одну операцию с плавающей точкой.

Это крупные вопросы, описанные в очень общем виде. При обсуждении конкретных методов анализа и преобразования в главах 9 и 10 мы приведем примеры, восполнив недостающие детали.

ИТОГИ РАЗДЕЛА

Большинство методов оптимизации, используемых компилятором, работают благодаря специализации кода общего назначения применительно к фактическому контексту. Некоторые преобразования своим успехом обязаны локальным эффектам, как в случае улучшения вычислений адресов элементов массива. Другие преобразования нуждаются в знании о более крупных участках кода.

Реализация любой оптимизации должна учитывать три основных вопроса. (1) Как компилятор может эффективно найти возможности для применения преобразования? (2) Безопасна ли оптимизация в этом контексте? (3) Сможет ли оптимизация улучшить код?

ВОПРОСЫ

1. Почему во фрагменте кода `dmxpy` программист развернул внешний, а не внутренний цикл? Как изменился бы результат, если бы он разворачивал внутренний цикл?
2. В показанном ниже фрагменте кода на C какие факты должен был бы установить компилятор, прежде чем улучшить код, заменив простую байт-ориентированную реализацию операций загрузки и сохранения?

```
MemCopy(char *source, char *dest, int length) {
 int i;
 for (i=1; i≤length; i++) {
  *dest++ = *source++;
 }
}
```

8.3. Область оптимизации

Область оптимизации
Участок кода, который анализирует и преобразует оптимизация, называется *областью* этой *оптимизации*.

Оптимизации работают на разных уровнях детализации, которые еще называются областями оптимизации. В предыдущем разделе мы рассматривали оптимизацию одной ссылки на элемент массива и всего вложенного цикла. Эти разные области предоставляли оптимизатору различные возможности. Изменив форму кода ссылки на массив, мы улучшили производительность только для одной этой ссылки. Переписывание вложенного цикла позволило улучшить производительность большего участка программы. В общем случае преобразования и поддерживающие их виды анализа ориентированы на одну из четырех областей: локальную, региональную, глобальную или межпроцедурную.

Локальные методы

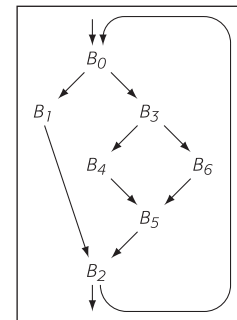
Локальные методы работают с одним простым блоком: последовательностью максимальной длины, не содержащей переходов. В ILOC-программе простой блок начинается помеченной операцией и заканчивается командой условного или безусловного перехода. В ILOC операция после условного или

безусловного перехода должна быть помечена, иначе на нее нельзя будет перейти; в других нотациях допускается «проваливание», так что помечать операцию после условного или безусловного перехода необязательно. Поведение линейного участка кода проще анализировать и понимать, чем участка, содержащего условные переходы и циклы.

Простой блок обладает двумя важными свойствами. Во-первых, предложения выполняются последовательно. Во-вторых, если выполнено какое-то предложение, то будет выполнен и весь блок, если только не возникнет исключение. Эти два свойства позволяют компилятору с помощью относительно простого анализа доказывать утверждения, более сильные, чем можно доказать для областей, содержащих условные или циклические управляющие конструкции (предложения if и циклы). Таким образом, локальные методы иногда позволяют выполнять такие улучшения, которые для более крупных областей невозможны. Но в то же время локальные методы способны улучшить только последовательности кода, целиком содержащиеся внутри одного простого блока.

Региональные методы

Региональные методы работают в областях, больших одного блока, но меньших полной процедуры. В примере графа потока управления (ГПУ) во врезке компилятор мог бы рассматривать весь цикл {B_0, B_1, B_2, B_3, B_4, B_5, B_6} как один регион. В некоторых случаях рассмотрение подмножества кода позволяет точнее проанализировать его и получить лучшее преобразование, чем было бы возможно для процедуры в целом. Например, компилятор иногда может доказать, что относительно вложенного цикла часто используемый указатель является инвариантом (т. е. принимает только одно значение), пусть даже он изменяется где-то в другом месте процедуры. Знание этого факта открывает возможность для некоторых оптимизаций; например, внутри цикла значение, на которое ссылается указатель, можно хранить в регистре.



Пример ГПУ

Компилятор может выбирать регионы разными способами. Регион может определяться какой-то управляющей конструкцией исходного кода, например вложенным циклом. Или компилятор мог бы рассмотреть подмножество блоков в регионе, образующее расширенный простой блок (РПБ). ГПУ из примера выше содержит три РПБ: {B_0, B_1, B_3, B_4, B_6}, {B_5} и {B_2}. Хотя два одноблочных РПБ не дают никаких преимуществ по сравнению с чисто локальным случаем, большой РПБ может предлагать дополнительные возможности для

Расширенный простой блок
набор блоков {B_0, B_1, ..., B_k}, в котором либо B_0 является входной вершиной, либо у B_0 есть несколько предшественников в ГПУ, а у всех остальных B_i – только один предшественник, принадлежащий набору.

Доминатор
Говорят, что в ГПУ вершина *x* доминирует над *y*, если любой путь из корня к *y* проходит через *x*.

оптимизации (см. раздел 8.5.1). Наконец, компилятор мог бы рассмотреть подмножество ГПУ, определенное каким-то теоретико-графовым свойством, например *отношением доминирования* или принадлежностью к компоненте сильной связности.

У региональных методов есть несколько преимуществ. Сосредоточение на регионе, меньшем полной процедуры, позволяет компилятору сконцентрировать усилия на часто исполняемых регионах, например теле цикла. (Ключевое для оптимизации наблюдение заключается в том, что тело цикла почти всегда исполняется чаще, чем окружающий код.) В случае региональных методов компилятор может применять разные стратегии оптимизации к разным регионам. Наконец, концентрация внимания на ограниченном участке кода иногда позволяет компилятору выводить более точную информацию о поведении программы, что, в свою очередь, может открыть возможности для улучшения.

Внутрипроцедурные методы

Эти методы, называемые также *глобальными*, рассматривают всю процедуру в качестве контекста. Обоснование простое: локально оптимальные решения могут иметь нежелательные последствия в более крупном контексте. Процедура дает компилятору естественную границу для анализа и преобразования. Процедуры – это абстракции, которые инкапсулируют и изолируют окружение выполнения. Во многих системах они также являются единицами раздельной компиляции.

Межпроцедурные и внутрипроцедурные методы

Мало найдется в теории компиляторов терминов, вызывающих такую путаницу, как *глобальный*. Глобальная оптимизация применяется к процедуре в целом. Но современный взгляд предполагает также наличие всеобъемлющей области оптимизации, по аналогии с глобальной переменной. Однако в случае оптимизации глобальный означает «относящийся к одной процедуре».

С ростом интереса к анализу и оптимизации через границы процедур сообществу понадобилась терминология, позволяющая различить *глобальный* анализ и анализ в более крупных областях. Термин «межпроцедурный» был введен для описания анализа, простирающегося от двух процедур до всей программы. Соответственно, для методов, применимых к одной процедуре, стали использовать термин «внутрипроцедурный». Но в английском написании эти слова (interprocedural и intraprocedural) очень похожи, что только увеличивает неразбериху.

Компания Perkin-Elmer Corporation ввела термин «универсальный» для описания оптимизации на уровне программы в целом. Однако этот термин не получил широкого распространения. Мы предпочитаем термин «общепрограммный» и пользуемся им там, где возможно. Он несет в себе правильный смысл, напоминая в то же время читателю, что «глобальный» не есть «универсальный».

Глобальные методы обычно строят представление процедуры, анализируют это представление и преобразуют соответствующий код. Если ГПУ содер-

жит циклы, то компилятор должен проанализировать всю процедуру, прежде чем поймет, какие факты имеют место при входе в каждый конкретный блок. Поэтому у большинства глобальных оптимизаций имеются раздельные фазы анализа и преобразования. В фазе анализа компилятор собирает факты и рассуждает о них. В фазе преобразования эти факты используются, чтобы установить безопасность и ожидаемую целесообразность преобразований. Благодаря наличию глобального представления эти методы могут обнаруживать возможности, невидимые локальным и региональным методам.

Межпроцедурные методы

Эти методы, иногда называемые *общепрограммными*, оперируют на уровне областей, больших одной процедуры. Любое преобразование, применяемое более чем к одной процедуре, мы считаем межпроцедурным. Переход от локальной области к глобальной открывает новые возможности, и то же самое справедливо для перехода от одной процедуры к нескольким. Но возникают и новые проблемы. Например, правила видимости имен ограничивают взаимодействие между выражениями в разных процедурах, а соглашения о передаче параметров затрудняют статический анализ через границы процедур.

Термин «общепрограммный» ясно намекает, что метод рассматривает весь код. Мы предпочитаем термин «межпроцедурный» для методов, которые анализируют несколько, но не все процедуры, и «общепрограммный» – для методов, исследующих всю программу.

Межпроцедурный анализ и оптимизация применяются, по крайней мере концептуально, к графу вызовов программы. Иногда анализу подвергается вся программа, а иногда – только ее часть. Два классических примера межпроцедурной оптимизации – встраивание, когда вызов заменяется копией кода вызываемой стороны, и межпроцедурное распространение констант, когда информация о константах распространяется по всей программе.

ИТОГИ РАЗДЕЛА

Компиляторы выполняют анализ и преобразование в разных областях, начиная с одиночных простых блоков (локальные методы) и кончая всей программой (общепрограммные методы). Набор возможностей зависит от области оптимизации; интуиция подсказывает, что чем шире область, тем больше возможностей. Однако анализ широких областей часто дает менее точные знания о поведении кода. Поэтому не существует простой связи между областью оптимизации и качеством кода. Было бы, конечно, хорошо, если бы оптимизация в большей области приводила к более качественному коду. Но, к сожалению, это не всегда верно.

ВОПРОСЫ

1. Простые блоки обладают тем свойством, что если выполнена хотя бы одна операция, то выполнены все операции, принадлежащие блоку, причем в указанном порядке (если только не произошло исключение). Сформулируйте более слабое свойство, имеющее место для расширенного простого блока.

2. Оптимизирующий компилятор PL/I был одним из первых компиляторов, выполнявших межпроцедурный анализ и оптимизацию. Однако эта возможность была ограничена набором процедур, входящих в один файл. Обоснуйте такое решение. Какие проблемы могли бы возникнуть у компилятора при попытке оптимизации в области, содержащей несколько файлов?

8.4. Локальная оптимизация

Локальные оптимизации принадлежат к числу простейших методов, доступных компилятору. Модель выполнения простого блока, подразумевающая линейное выполнение кода вдоль одного пути, дает довольно точный анализ для поддержки оптимизации. Поэтому такие методы на удивление эффективны.

Избыточность
Выражение *e* является *избыточным* в точке *p*, если оно уже было вычислено на каждом пути, ведущем в *p*.

В этом разделе представлены примеры двух локальных методов. Первый, *нумерация значений*, находит избыточные выражения в простом блоке и заменяет избыточные вычисления ссылками на ранее вычисленное значение. Второй, *балансировка деревьев по высоте*, реорганизует деревья выражений, так чтобы у планировщика появилось больше возможностей.

8.4.1. Локальная нумерация значений

```
a ← b + c
b ← a - d
c ← b + c
d ← a - d
```
Исходный блок

```
a ← b + c
b ← a - d
c ← b + c
d ← b
```
Переписанный блок

Рассмотрим простой блок из четырех предложений, показанный во врезке. Выражение, например `b + c` или `a - d`, является избыточным в блоке, если оно было ранее вычислено в этом блоке и в промежутке между вычислениями не было операций, переопределяющих какие-то аргументы. В примере выражение `b + c` в третьей операции не является избыточным, потому что вторая операция переопределяет `b`. Выражение `a - d` в четвертой операции избыточно, потому что ни `a`, ни `d` не переопределены между второй и четвертой операциями.

Консолидация
метод, который удаляет ненужное копирование, `a ← b`, путем объединения `a` и `b`.

Компилятор может переписать этот блок, так чтобы `a - d` вычислялось только один раз (показано во врезке). Второе вычисление `a - d` заменено копией `b`. Альтернативная стратегия – заменять последующие вхождения `d` на `b`. Однако при таком подходе требуется анализ, устанавливающий, что `b` не было переопределено перед последующим использованием `d`. На практике проще поручить оптимизатору вставить копию и дать возможность следующим проходам определить, нужна копия или нет (см. обсуждение консолидации копий в разделе 13.4.3).

Время жизни
В контексте оптимизации временем жизни значения является участок кода между его определением и использованием.

В блоке время жизни совпадает с интервалом между определением и последним использованием.

В общем случае замена избыточных вычислений ссылками на ранее вычисленные значения целесообразна, т. е. результирующий код работает быстрее исходного. Однако целесообразность не гарантируется. Замена `d ← a - d` на `d ← b` потенциально может продлить время жизни `b` и сократить время жизни `a`, `d` или той и другой – в каждом случае это зависит от того, где находится последнее использование значения. В зависимости от деталей переписывание может увеличить, уменьшить или оставить неизменным спрос на регистры. Замена избыточного вычисления ссылкой, скорее всего, окажется целесообразной, если только это не заставляет распределитель регистров выгружать дополнительные значения.

На практике оптимизатор не может надежно предсказать поведение распределителя регистров, отчасти потому что еще до распределения код может быть подвергнут дополнительным преобразованиям. Поэтому в большинстве алгоритмов устранения избыточности предполагается, что переписывание с целью избавиться от избыточности целесообразно.

```
a ← b × c
d ← b
e ← d × c
```

Эффект присваивания

В примере выше избыточное выражение было текстуально идентично первому вхождению. Но, конечно, присваивания может породить избыточное выражение, текстуально отличающееся от предшествующего. Рассмотрим блок, показанный во врезке. Присваивание `d ← b` заставляет выражение `d × c` породить то же значение, что `b × c`. Для распознавания этого случая компилятор должен отслеживать поток значений одного и того же имени. Методы, основанные на текстуальной идентичности, такие случае не распознают.

Присваивание `y(i)` на рис. 8.1 создаст несколько избыточных подвыражений вследствие применения формул для различных адресов.

Программисты будут с возмущением заявлять, что никогда не пишут код, содержащий избыточные выражения, как в примере выше. Но на практике находится много возможностей для устранения избыточности. При трансляции исходного кода в ПП имеется много мест, например вычисление адресов, где могут появиться избыточные выражения.

Для нахождения и устранения избыточности было разработано много методов. *Локальная нумерация значений* (ЛНЗ, *англ.* LVN) – одно из самых старых и самых мощных преобразований такого рода. Оно обнаруживает избыточности в простом блоке и переписывает блок, так чтобы избавиться от них. При этом предлагается простая и эффективная схема для других локальных оптимизаций, например свертки констант и упрощений с использованием алгебраических тождеств.

Алгоритм

Идея ЛНЗ проста. Алгоритм обходит простой блок и назначает номер каждому значению, которое в нем вычисляется. Номера выбираются так, что два выражения, *M* и *N*, имеют одинаковые номера тогда и только тогда, когда значения *M* и *N* доказуемо одинаковы для всех возможных выполнений блока.

На рис. 8.2 показан базовый алгоритм ЛНЗ. Он принимает блок, содержащий *n* бинарных операций вида $T_i \leftarrow L_i\ Op_i\ R_i$. Все операции исследуются по порядку. Для отображения имен, констант и выражений на различные номера значений используется хеш-таблица, первоначально пустая.

```
for i ← 1 to n, где блок включает n операций вида "Ti ← Li Opi Ri" do
  получить номера значений V1 и V2 для Li и Ri
  if в таблице нет записи для Ti then
    создать запись для Ti
  построить хеш-ключ k как ⟨V1, Opi, V2⟩
  if k уже присутствует в таблице then
    заменить операцию i на "Ti ← x", где x – имя, ассоциированное с ключом k
  else
    создать в таблице запись для k
    сгенерировать новый номер значения для k и сохранить его в записи таблицы для k
    сохранить имя Ti в записи таблицы для k
  сохранить номер значения k в записи таблицы для Ti
```

Рис. 8.2 Нумерация значений в одном блоке

Для обработки *i*-й операции ЛНЗ получает номера значений для L_i и R_i из хеш-таблицы. Если запись в таблице найдена, то ЛНЗ использует номер значения из этой записи. В противном случае создается новая запись и в нее помещается новый номер значения.

Важность порядка

Порядок, в котором написаны выражения, прямо влияет на способность оптимизатора анализировать и преобразовывать их. Рассмотрим применение локальной нумерации значений к двум разным записям присваивания `v ← a × b × c`:

```
t0 ← a × b        t0 ← b × c
 v ← t0 × c        v ← a × t0
```

В записи слева номера присваиваются значениям `a`, `b`, `a × b`, `c`, `(a × b) × c` и `v`, а в записи справа – значениям `b`, `c`, `b × c`, `a`, `a × (b × c)` и `v`. В зависимости от окружающего контекста предпочтителен может быть тот или иной порядок. Например, если `b × c` впоследствии встречается в блоке, а `a × b` не встречается, то правая запись порождает избыточность, а левая – нет.

В общем случае использование коммутативности, ассоциативности и дистрибутивности для изменения порядка выражений может изменять результаты оптимизации. Аналогичные явления можно наблюдать при свертке констант; если мы заменим `a` на `3`, а `c` на `5`, то никакое упорядочение не порождает константную операцию `3 × 5`, которую можно свернуть.

Поскольку число способов переупорядочить выражения слишком велико, компиляторы применяют эвристические методы, чтобы найти хороший порядок выражений. Например, компилятор IBM FORTRAN H генерировал вычисления адресов элементов массива в порядке, который обещал улучшить другие оптимизации. Другие компиляторы сортировали операнды коммутативных и ассоциативных операций в порядке, соответствующем уровню вложенности цикла, на котором они были определены. Поскольку возможных эвристических решений этой задачи много, необходимы эксперименты и настройка, чтобы отыскать то из них, которое в наибольшей степени соответствует конкретному языку, компилятору и стилю кодирования.

Если известны номера значений для L_i и R_i, которые мы обозначим V_1 и V_2, то ЛНЗ конструирует хеш-ключ из $\langle V_1, Op_i, V_2 \rangle$ и ищет этот ключ в таблице. Если соответствующая запись существует, то выражение избыточно и может быть заменено ссылкой на ранее вычисленное значение. В противном случае операция *i* является первым вычислением $L_i\ Op_i\ R_i$ в блоке, поэтому ЛНЗ создает запись для ее хеш-ключа и помещает в нее новый номер значения. Он также помещает номер значения хеш-ключа, не важно, нового или ранее существовавшего, в запись таблицы для T_i. Поскольку в ЛНЗ номера для построения хеш-ключа выражения используются номера значений, а не имена, алгоритм способен эффективно отслеживать поток значений в операциях копирования и присваивания, в частности в небольшом примере, озаглавленном «Эффект присваивания» выше. Обобщение ЛНЗ на выражения произвольной арности не вызывает затруднений.

Чтобы понять, как ЛНЗ вычисляет номера значений, рассмотрим наш пример, повторенный во врезке. Каждое имя аннотировано номером значения, присвоенным ему ЛНЗ (оно показано в виде верхнего индекса). В первой операции, когда таблица значений еще пуста, `b` и `c` получают новые номера значений: 0 и 1 соответственно. ЛНЗ конструирует текстовую строку «0+1», которая служит хеш-ключом выражения `b + c` и производит поиск в таблице. Поскольку записи с таким ключом нет, поиск завершается неудачно. Поэтому ЛНЗ создает новую запись для «0+1» и сопоставляет ей номер значения 2. Затем ЛНЗ создает запись для `a` и помещает в нее номер значения выражения, т. е. 2. Повторение этого процесса для каждой операции по порядку дает остальные номера значений, показанные во врезке.

```
a² ← b⁰ + c¹
b⁴ ← a² - d³
c⁵ ← b⁴ + c¹
d⁴ ← a² - d³
```
Пример с номерами значений

Номера значений показывают – совершенно правильно, – что два вхождения `b + c` порождают разные значения вследствие промежуточного переопределения `b`. Они также показывают, что оба вхождения `a - d` порождают одно и то же значение. В обоих вхождениях `a - d` номер значения `a` одинаковый, и то же верно для `d`. ЛНЗ обнаруживает этот факт и фиксирует его путем присваивания `b` и `d` одного и того же номера значения, 4. Это знание позволяет ЛНЗ переписать четвер-

```
a ← b + c
b ← a - d
c ← b + c
d ← b
```
Пример после перезаписи

тую операцию в виде d ← b, как показано во врезке. Последующие проходы могут консолидировать и устранить эту операцию копирования.

Обобщение алгоритма

ЛНЗ предлагает инфраструктуру для нескольких других локальных оптимизаций.

Коммутативные операции. Различные варианты коммутативного выражения, например a × b и b × a, должны получать один и тот же номер значения. При построении хеш-ключа для правой части текущей операции ЛНЗ может отсортировать операнды с помощью какой-нибудь удобной схемы, например по номеру значения. Это простое действие гарантирует, что варианты коммутативного выражения получат одинаковые номера значений.

Свертка констант. Если все операнды некоторой операции имеют известные постоянные значения, то ЛНЗ может выполнить операцию во время компиляции и подставить ответ прямо в код. ЛНЗ может помечать записи хеш-таблицы как константы и хранить их значения вместе с номерами значений. Обнаружив константное выражение, ЛНЗ может вычислить его и заменить выражения литеральной константой.

Алгебраические тождества. Алгоритм ЛНЗ может использовать алгебраические тождества для упрощения кода. Например, x + 0 и x должны получать одинаковые номера значений. К сожалению, ЛНЗ нуждается в специальном случае для каждого тождества. Последовательность проверок, по одной для каждого тождества, существенно замедлила бы ЛНЗ. Чтобы решить эту проблему, ЛНЗ должен организовать проверки в виде деревьев решений, зависящих от операторов. Поскольку для каждого оператора имеется всего несколько тождеств, такой подход позволяет снизить накладные расходы. На рис. 8.3 показаны некоторые тождества, которые могут быть обработаны в ЛНЗ.

a + 0 = a	a - 0 = a	a - a = 0	2 × a = a + a
a × 1 = a	a × 0 = 0	a ÷ 1 = a	a ÷ a = 1, a ≠ 0
a^1 = a	a^2 = a × a	a ≫ 0 = a	a ≪ 0 = a
a AND a = a	a OR a = a	MAX(a,a) = a	MIN(a,a) = a

Рис. 8.3 Алгебраические тождества для нумерации значений

NaN
Not a Number (не число) – предопределенная константа, которая представляет недопустимый или бессмысленный результат в стандарте IEEE арифметики с плавающей точкой.

Изобретательный разработчик найдет и другие тождества, в т. ч. зависящие от типа. Операция ИСКЛЮЧАЮЩЕЕ ИЛИ для двух одинаковых значений должна давать ноль соответствующего типа. Для чисел с плавающей точкой в формате IEEE имеются свои специальные случаи, связанные с явными представлениями ∞ и *NaN*; например, ∞ − ∞ = *NaN*; ∞ − *NaN* = *NaN* и ∞ ÷ *NaN* = *NaN*.

На рис. 8.4 показан алгоритм ЛНЗ с этими расширениями. Для обработки коммутативных операций он сортирует операнды по номерам их значений

перед построением хеш-ключа. Алгоритм вычисляет константные выражения и применяет алгебраические тождества. Даже с такими расширениями стоимость в расчете на одну операцию ПП остается очень низкой, потому что каждый шаг имеет эффективную реализацию.

```
for i ← 1 to n, где блок включает n операций вида "Ti ← Li Opi Ri" do
  получить номера значений V1 и V2 для Li и Ri
  if в таблице нет записи для Ti then
    создать запись для Ti
  if Opi коммутативна и V1 > V2 then
    переставить местами V1 и V2
  построить хеш-ключ k как ⟨V1, Opi, V2⟩
  if k уже присутствует в таблице then
    заменить операцию i на "Ti ← x", где x – значение константы
      или имя, ассоциированное с ключом k
  else
    создать в таблице запись для k
    сгенерировать новый номер значения для k и сохранить его в записи таблицы для k
    сохранить имя Ti в записи таблицы для k
    if V1 и V2 помечены как константы then
      поместить значение Li Opi Ri в c
      заменить операцию i на "Ti ← c"
      сохранить c в записях таблицы для k и Ti
    else if Li Opi Ri совпадает с одним из алгебраических тождеств then
      заменить операцию i упрощенной версией
  сохранить номер значения k в записи таблицы для Ti
```

Рис. 8.4 Локальная нумерация значений с расширениями

Роль именования

Выбор имен переменных и значений может ограничить эффективность нумерации значений. Посмотрим, что происходит, когда ЛНЗ применяется к блоку, показанному во врезке. Как и раньше, верхние индексы обозначают номера значений, присвоенные каждому имени и значению.

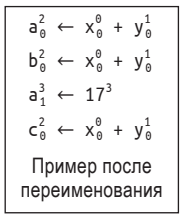
$a^2 \leftarrow x^0 + y^1$
$b^2 \leftarrow x^0 + y^1$
$a^3 \leftarrow 17^3$
$c^2 \leftarrow x^0 + y^1$

Пример с проставленными номерами значений

В первой операции ЛНЗ назначает 0 имени x, 1 – y и 2 – x + y и a. Во второй операции алгоритм обнаруживает, что x + y избыточно, поскольку имеет номер значения 2, поэтому можно заменить b ← x + y на b ← a. Третья операция избыточной не является. В четвертой операции ЛНЗ снова обнаруживает, что x + y имеет номер значения 2. Но переписать эту операцию в виде c ← a он не может, потому что номер значения a уже не равен 2.

Решить эту проблему ЛНЗ может двумя способами. Можно хранить отображение номеров значений на имена и на каждой операции обновлять это отображение. Предположим, что операция определяет t. ЛНЗ должен удалить t из списка для его старого номера значения и добавить t в список для его нового номера значения. В момент замены ЛНЗ может использовать любое

имя, которое в данный момент хранится в отображении для этого номера значения. Такой подход несколько удорожает обработку каждой операции и загромождает базовый алгоритм.



Пример после переименования

В качестве альтернативы компилятор может переписать код, так чтобы каждая операция определяла уникальное имя. Для обеспечения уникальности достаточно добавить нижний индекс к имени, как показано во врезке. При таких новых именах код определяет каждое значение ровно один раз. Таким образом, ни одно значение не переопределяется и не пропадает (*уничтожается*). Применив ЛНЗ к этому блоку, мы получим желаемый результат. Это доказывает, что вторая и четвертая операции избыточны; каждую можно заменить копией из a_0. (Заметим, что после переименования a_0 остается живым в более протяженном участке кода, чем первоначальное a.)

Однако теперь компилятор должен согласовывать эти индексированные имена с именами в окружающих блоках, чтобы сохранить семантику исходного кода. В нашем примере исходное имя a должно ссылаться на значение из индексированного имени a_1 в переписанном коде. Хорошая реализация отобразила бы новое a_1 в исходное a, b_0 – в исходное b, а c_0 – в исходное c; она также переименовала бы a_0, присвоив ему новое временное имя. Такое решение согласует пространство имен преобразованного блока с окружающим контекстом без введения копий.

Эта схема именования аппроксимирует одно свойство пространства имен, созданного для формы с единственным статическим присваиванием (SSA) (см. разделы 4.6.2 и 9.3). С точки зрения проектирования, простое решение заключается в том, чтобы применить ЛНЗ к коду, уже представленному в форме SSA.

Исключения и оптимизация

Некоторые аномальные условия, складывающиеся во время выполнения, могут приводить к возбуждению исключений, например ссылки на память за пределами выделенной области, неопределенные арифметические операции, такие как деление на ноль, и недопустимые операции. (Чтобы активировать точку останова, отладчик мог бы заменить интересующую команду недопустимой, а затем перехватить исключение.) Некоторые языки включают средства обработки исключений – как предопределенных, так и определенных программистом.

Обычно исключение вызывает передачу управления «обработчику исключения», специально предназначенному для реакции на исключение. Обработчик может решить проблему, повторно выполнить приведшую к ошибке операцию и вернуть управление блоку. Альтернативно он может передать управление в какое-то другое место или завершить выполнение.

Оптимизатор должен понимать, какие операции могут возбудить исключение, и учитывать влияние исключения на выполнение программы. Поскольку обработчик исключения может модифицировать значения переменных или передавать управление, компилятор должен относиться к операциям, возбуждающим исклю-

чение, с осторожностью. Например, любая операция, возбуждающая исключение, теоретически может завершить текущий простой блок. Такое отношение может серьезно ограничить способность оптимизатора улучшать код.

Чтобы оптимизировать содержащий исключения код, компилятору необходимо понять и смоделировать эффекты обработчиков исключений. Для этого ему необходим доступ к коду обработчиков исключений и понимание того, какие обработчики могут быть задействованы при выполнении конкретной операции, вызвавшей исключение.

Влияние косвенных присваиваний

Неоднозначная ссылка
Ссылка является неоднозначной, если компилятор не может соотнести ее с единственным адресом в памяти.

В предыдущем обсуждении предполагалось, что присваивания прямые и очевидные, как в случае `a ← b × c`. Но многие программы содержат косвенные присваивания, когда компилятор может не знать, какие значения модифицируются, потому что ссылка неоднозначна. Примерами могут служить: (1) присваивание по указателю, как, например, `*p=0;` в языке C; (2) присваивание элементу структуры или массива, как, например, `a(i,j) = 0` в FORTRAN; (3) присваивание с помощью параметра, передаваемого по ссылке. Косвенные присваивания усложняют нумерацию значений и другие оптимизации, потому что мешают компилятору точно понять поток значений.

Без ограничения общности можно предположить, что имена в наших примерах являются адресами в памяти. Рассмотрим, какое влияние косвенное присваивание типа `*p = 0` должно оказывать на состояние ЛНЗ. Если компилятор не знает адрес, хранящийся в `p`, то он должен предполагать, что любое значение, которое `p` *мог бы* адресовать, изменилось. В реализации ЛНЗ, где имена представляют адреса в памяти, неоднозначное присваивание сопоставило бы каждому имени его собственный новый номер значения.

Звучит ужасно, но показывает истинное влияние неоднозначного присваивания на множество фактов, которые может вывести компилятор. Компилятор может попытаться устранить неоднозначность ссылок. Для ссылок на основе указателей он может выполнить анализ, чтобы сузить множество переменных, которые, по мнению компилятора, может адресовать указатель. Существует ряд методов, помогающих компилятору понять паттерны доступа к элементам массива – тоже чтобы свести множество элементов, которые предположительно могут быть модифицированы присваиванием, к одному. Межцпроцедурный анализ псевдонимов может разрешить некоторые неоднозначности, вызванные взаимодействиями между соглашениями о передаче параметров и правилами видимости имен.

8.4.2. Балансировка дерева по высоте

Оптимизация выражения при наличии нескольких функциональных блоков демонстрирует влияние формы кода на способность компилятора улучшить

вычисление. У большинства процессоров есть несколько функциональных блоков; эти блоки могут выполнять независимые операции в каждом такте. Если компилятор может выразить вычисление, представив его в виде независимых операций, то планировщик и процессор смогут выполнить их конкурентно. Если же, напротив, компилятор привносит излишние последовательные ограничения, то планировщик и процессор, возможно, не сумеют найти достаточную степень конкурентности, чтобы занять все функциональные устройства.

$t_1 \leftarrow a + b$
$t_2 \leftarrow t_1 + c$
$t_3 \leftarrow t_2 + d$
$t_4 \leftarrow t_3 + e$
$t_5 \leftarrow t_4 + f$
$t_6 \leftarrow t_5 + g$
$t_7 \leftarrow t_6 + h$

Вычисление
a + b + c + d + e + f + g + h
слева направо

Рассмотрим выражение `a + b + c + d + e + f + g + h`. В коде во врезке представлено вычисление слева направо, эквивалентное обходу левоассоциативного дерева на рис. 8.5(a) в обратном порядке. Вычислению выражения справа налево соответствовало бы дерево на рисунке (b). Каждое дерево налагает на порядок выполнения ограничения более строгие, чем правила сложения. Левоассоциативное дерево вынуждает код вычислить `a + b` до прибавления `g` или `h`. Правоассоциативное дерево требует, чтобы вычисление `g + h` предшествовало прибавлению `a` или `b`. Сбалансированное дерево на рисунке (c) налагает меньше ограничений, но все равно ограничивает порядок вычислений сильнее, чем правила сложения.

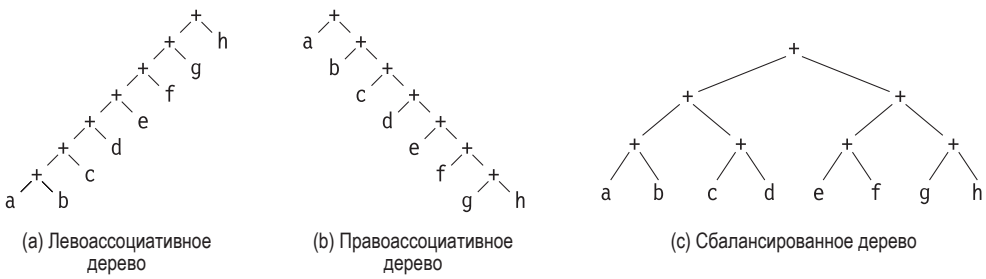


Рис. 8.5 Возможные формы дерева для выражения `a + b + c + d + e + f + g + h`

Если процессор может выполнять несколько операций сложения одновременно, то сбалансированное дерево должно приводить к более короткому плану вычисления. На рис. 8.6 показаны кратчайшие возможные планы для сбалансированного и левоассоциативного деревьев на компьютере с двумя однотактными сумматорами. Сбалансированное дерево можно выполнить за четыре такта, причем в четвертом такте один блок будет простаивать. С другой стороны, левоассоциативное дерево требует семи тактов, потому что сложения сериализованы. Один сумматор остается вообще незадействованным. Правоассоциативное дерево дает аналогичные результаты.

Параллелизм на уровне команд
Независимые операции можно выполнять в одном такте. Такой вид конкурентности называется параллелизмом на уровне команд.

Этот простой пример наводит на мысль о важной оптимизации: использовать коммутативность и ассоциативность для выяв-

ления дополнительного параллелизма при вычислении выражения. В оставшейся части этого раздела представлен алгоритм, который переписывает один блок с целью улучшить баланс на всем лесе деревьев выражений. Это преобразование раскрывает бóльшую степень *параллелизма на уровне команд* (instruction-level parallelism – ILP) планировщику команд.

1	$t_1 \leftarrow a + b$	–
2	$t_2 \leftarrow t_1 + c$	–
3	$t_3 \leftarrow t_2 + d$	–
4	$t_4 \leftarrow t_3 + e$	–
5	$t_5 \leftarrow t_4 + f$	–
6	$t_6 \leftarrow t_5 + g$	–
7	$t_7 \leftarrow t_6 + h$	–

(a) Левоассоциативное дерево

1	$t_1 \leftarrow a + b$	$t_2 \leftarrow c + d$
2	$t_3 \leftarrow e + f$	$t_4 \leftarrow g + h$
3	$t_5 \leftarrow t_1 + t_2$	$t_6 \leftarrow t_3 + t_4$
4	$t_7 \leftarrow t_5 + t_6$	–
5	–	–
6	–	–
7	–	–

(b) Сбалансированное дерево

Рис. 8.6 Планы выполнения для деревьев на рис. 8.5

Деревья-кандидаты

На первом шаге алгоритм должен идентифицировать деревья, которые могут стать кандидатами для балансировки. Дерево или поддерево-кандидат должно содержать только один вид операторов, например + или ×. Чтобы можно было менять порядок операндов, этот оператор должен быть ассоциативным и коммутативным. Для максимизации ILP дерево должно быть как можно больше. Если дерево слишком мало, то оптимизация ничего не даст.

Чтобы переписанный код гарантированно работал в окружающем его контексте, он должен сохранять значения, используемые вне выражения. Говорят, что значение *экспонировано*, если оно используется после блока, если оно используется более одного раза внутри блока или если оно встречается в качестве аргумента оператора другого типа.

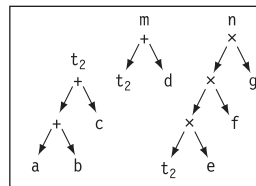
Экспонированные значения должны быть сохранены, т. е. переписанное вычисление должно порождать те же экспонированные значения, что исходное. Это ограничение может помешать алгоритму в реорганизации кода.

$t_1 \leftarrow a + b$
$t_2 \leftarrow t_1 + c$
$m \leftarrow t_2 + d$
$t_3 \leftarrow t_2 \times e$
$t_4 \leftarrow t_3 \times f$
$n \leftarrow t_4 \times g$

Пример простого блока

Во фрагменте кода во врезке вычисляется два больших выражения: $m \leftarrow (a + b + c + d)$ и $n \leftarrow (a + b + c) \times e \times f \times g$. Предположим, что m и n используются позднее, т. е. являются экспонированными значениями. t_2 также экспонировано; его необходимо сохранить для использования в вычислении t_3.

Тот факт, что t_2 экспонировано, влияет на код двояко. Во-первых, компилятор не может переписать $(a + b + c + d)$ как $(a + b) + (c + d)$, потому что последнее выражение никогда не вычисляет значение t_2. Во-вторых, вычисление t_2 должно предшествовать вычислению t_3, что опять-таки ограничивает порядок операций.



Деревья в примере

Алгоритм нахождения деревьев-кандидатов должен трактовать каждое экспонированное значение как корень дерева. В нашем примере это порождает лес, показанный во врезке. Заметим, что t_2 определено один раз, а используется дважды. Деревья для t_2 и `m` слишком малы, и потому балансировка их по высоте не дает никакого выигрыша. С другой стороны, дерево для `n` с четырьмя операндами сбалансировать можно.

Экспонирование сверху
Имя *x экспонировано сверху* в блоке *b*, если первое использование *x* в *b* относится к значению, вычисленному до входа в *b*.

Если определяющее ПП не является набором деревьев выражений, то компилятор может либо построить деревья выражений для блока в виде производного ПП, либо интерпретировать определяющее ПП как деревья выражений для управления алгоритмом. Последний подход требует некоторых простых структур данных для связывания каждого использования в блоке либо с определением в блоке, которое достигает его, либо с нотацией, обозначающей, что это использование *экспонировано сверху*.

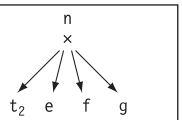
Высокоуровневый набросок алгоритма

Алгоритм балансировки дерева, показанный на рис. 8.7, состоит из двух этапов. На первом этапе ищутся корни деревьев-кандидатов выражений в блоке. Операторы, не являющиеся коммутативными и ассоциативными, игнорируются. Если значение *T* экспонировано и является результатом коммутативного и ассоциативного операторов, то алгоритм добавляет *T* в очередь корней, упорядоченную по арифметическому приоритету оператора.

```
// Все операции имеют вид "T_i ← L_i Op_i R_i"
// Этап 1: построить очередь Roots деревьев-кандидатов
Roots ← новая пустая очередь
for i ← 0 to n-1 do
   Rank(T_i) ← -1
   if Op_i коммутативна и ассоциативна и T_i экспонировано then
      пометить T_i как корень
      Enqueue(Roots, T_i, приоритет Op_i)

// Этап 2: удалить дерево из Roots и перебалансировать его
while (Roots непуста) do
   var ← Dequeue(Roots)
   Balance(var)
```

Рис. 8.7 Алгоритм балансировки дерева по высоте



n-арное дерево для n

На втором этапе корни берутся в порядке приоритета от наименьшего к наибольшему и создается сбалансированное дерево путем линеаризации дерева в один *n*-арный оператор с последующей перестройкой в сбалансированной форме. *n*-арное дерево для `n` из нашего сквозного примера показано во врезке.

Этап 1: нахождение деревьев-кандидатов

Этап 1 алгоритма прямолинеен. Алгоритм обходит все операции в блоке по порядку и для каждой проверяет, должна ли она стать корнем дерева-кандидата. Проверка опирается на сделанные ранее наблюдения. Корень дерева-кандидата должен быть результатом использования коммутативного и ассоциативного операторов для создания экспонированного значения.

Чтобы установить, является ли значение T_i, определенное операцией i, экспонированным, компилятор должен понять, где T_i используется. Компилятор может вычислить множества LiveOut для каждого блока и воспользоваться этой информацией, чтобы узнать, используется ли T_i после текущего блока (см. раздел 8.6.1). Попутно с вычислением этой начальной информации компилятор может также определить, сколько раз T_i используется внутри блока.

Затем проверку на корень дерева-кандидата можно выразить кратко. Для операции $T \leftarrow L\ Op\ R$ в блоке B T является корнем тогда и только тогда, когда

Op коммутативна и ассоциативна и
(T используется в B более одного раза или T ∈ LiveOut(B))

На этапе 1 инициализируются ранги всех T_i, определенных в блоке. Для каждого найденного корня алгоритм помечает имя как корень и добавляет его в приоритетную очередь корней. Очередь для нашего сквозного примера показана во врезке; предполагается, что приоритет + равен 1, а приоритет × равен 2.

```
(⟨t2,1⟩, ⟨m,1⟩,⟨n,2⟩)
```
Очередь корней для нашего примера

Этап 2: перестройка блока в сбалансированной форме

На этапе 2 берется очередь корней деревьев-кандидатов и для каждого корня строится приблизительно сбалансированное дерево. Этап 2 начинается циклом while, в котором для каждого корня дерева-кандидата вызывается функция *Balance*. Функции *Balance*, *Flatten* и *Rebuild* составляют реализацию этапа 2, как показано на рис. 8.8 и 8.9.

Balance вызывается для каждого корня дерева-кандидата в порядке возрастания приоритетов. Если дерево еще не балансировалось, то *Balance* вызывает *Flatten*, чтобы создать новую приоритетную очередь, в которой хранятся все операнды текущего дерева. По существу, очередь представляет собой одноуровневую n-арную версию дерева-кандидата. После того как *Flatten* создала очередь, *Balance* вызывает *Rebuild*, чтобы сгенерировать код приблизительно сбалансированного дерева.

С *Flatten* никаких трудностей не возникает. Она рекурсивно обходит дерево-кандидат, назначает каждому узлу ранг и добавляет узел в очередь. Листовым узлам *Flatten* назначает ранги непосредственно: 0 – константе и 1 – листу. Если *Flatten* встречает корень, то вызывает для него *Balance*, которая создает новую

Напомним, что на этапе 1 очередь *Roots* была ранжирована по приоритету арифметической операции, поэтому *Flatten* автоматически производит вычисления в правильном порядке.

очередь, линеаризует дерево-кандидат с этим корнем и перестраивает его. Корень сохраняет приоритет, назначенный ему этим рекурсивным вызовом.

```
Balance(root)      // Создать сбалансированное дерево по его корню T_i
  if Rank(root) ≥ 0 then
    return                                  // это дерево уже обрабатывалось

  q ← новая пустая очередь                  // линеаризовать, а затем перестроить дерево
  Rank(root) ← Flatten(L_i,q) + Flatten(R_i,q)
  Rebuild(root, q, Op_i)

Flatten(node,q)    // Flatten вычисляет ранг узла и строит очередь
  if node – константа then                  // константа является листом дерева
    Rank(node) ← 0
    Enqueue(q,node,Rank(node))
  else if node экспонирован сверху then     // экспонированный сверху узел является листом дерева
    Rank(node) ← 1
    Enqueue(q,node,Rank(node))
  else if node является корнем then         // корень другого дерева является листом
    Balance(node)                           // Balance устанавливает Rank(node)
    Enqueue(q,node,Rank(node))
  else                                      // внутренний узел для T_i ← L_i Op_i R_i
    Rank(node) ← Flatten(L_i,q) + Flatten(R_i,q)
  return Rank(node)
```

Рис. 8.8 Алгоритм балансировки дерева по высоте: Balance и Flatten

```
Rebuild(root, q, op)  // построить сбалансированное выражение
  while (q не пуста) do
  NewL ← Dequeue(q)    // получить левый операнд
  NewR ← Dequeue(q)    // получить правый операнд

  if NewL и NewR константы then
    // вычислить и свернуть результат
    NewT ← Fold(op, NewL,NewR)
    if q пуста then
      создать операцию “root ← NewT”
      Rank(root) = 0
    else
      Enqueue(q,NewT, 0)
      Rank(NewT) = 0

  else if q пуста then    // не константа, поименовать результат
    NewT ← root
  else
    NewT ← new name

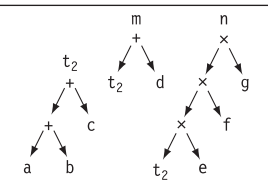
  создать операцию "NewT ← NewL op NewR"
  Rank(NewT) ← Rank(NewL) + Rank(NewR)
  if q не пуста then    // есть еще операции в q ⇒ добавить NewT в q
    Enqueue(q, NewT, Rank(NewT))
```

Рис. 8.9 Алгоритм балансировки дерева по высоте: Rebuild

Когда *Balance* вызывает *Rebuild*, очередь содержит все листья дерева-кандидата в порядке рангов. Промежуточные результаты, представленные внутренними узлами исходных деревьев выражений, отброшены.

Rebuild использует простой алгоритм для переписывания кода. Она в цикле удаляет из дерева два узла с наименьшими рангами. Если это константы, то функция вычисляет операцию и помещает результат обратно в очередь с приоритетом 0. Поскольку все константы имеют ранг 0, *Rebuild* свернет их в один константный терм. Если один или оба элемента имеют ненулевой ранг, то *Rebuild* создает операцию для комбинирования обоих элементов и назначает этой операции новый ранг. Наконец, она помещает новую операцию в очередь. Этот процесс продолжается, пока очередь не опустеет.

Balance, по сути дела, собирает вместе константы, чтобы *Rebuild* могла их скомбинировать.

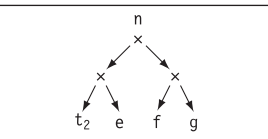


Деревья выражений

Деревья выражений и очередь *Roots*, порождаемые на этапе 1 для нашего сквозного примера, показаны во врезке. На этапе 2 каждый корень извлекается из очереди и передается функции *Balance*. И в `m`, и в t_2 операндов недостаточно, чтобы алгоритм создал сбалансированное дерево.

(⟨m,1⟩, ⟨t_2,1⟩, ⟨n,2⟩)
Очередь *Roots*

Когда на этапе 2 `n` извлекается из очереди *Roots*, функция *Flatten* строит очередь, которая содержит все четыре операнда, каждый ранга 1. Затем *Rebuild* создает операции для вычисления `g * f` и t_2 `* e`, помещает эти операции в очередь с рангом 2, после чего извлекает их и размещает под корнем, `n`. Перестроенное дерево для `n` показано во врезке.



Перестроенное дерево для `n`

Более полный пример

На рис. 8.10(a) показан второй пример. Константы сворачиваются, а избыточные вычисления исключаются, как мог бы сделать алгоритм ЛНЗ. Все значения `s`, `t`, `u`, `v`, `w`, `x`, `y` и `z` экспонированы сверху; по выходе из блока t_6, t_{10} и t_{11} остаются живы.

Блок содержит несколько перемежающихся вычислений. На рисунке (b) показаны все пять деревьев выражений, встречающихся в этом блоке. Каждое из значений t_3 и t_7 используется несколько раз, поэтому мы показываем для них отдельные деревья. Связи с t_3 и t_7 показаны серыми стрелками.

Применив этап 1 алгоритма балансировки деревьев к этому примеру, мы найдем пять корней: три значения, остающихся живыми после выхода, а также t_3 и t_7. Эти корни обведены рамками на рисунках (b) и (c). В конце этапа 1 *Roots* содержит: {⟨t_{11},1⟩, ⟨t_9,1⟩, ⟨t_3,1⟩, ⟨t_{10},2⟩, ⟨t_6,2⟩}.

Этап 2 алгоритма начинается с очереди *Roots*. Из нее извлекается узел, и для его обработки вызывается *Balance*. После возврата из *Balance* процесс повторяется, пока очередь *Roots* не опустеет.

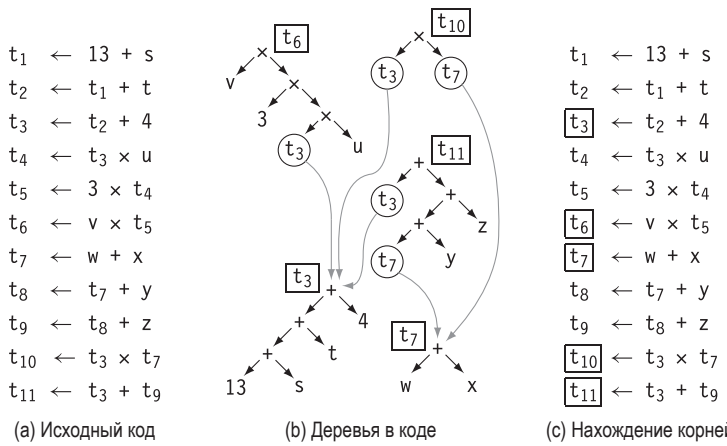


Рис. 8.10 Детальный пример балансировки дерева по высоте

В нашем примере алгоритм этапа 2 сначала вызывает *Balance* для t_{11}. *Balance* вызывает *Flatten*, которая строит очередь. Встретив корень t_3, *Flatten* вызывает *Balance*(t_3).

Будучи применена к t_3, *Balance* сначала вызывает *Flatten*, которая порождает очередь {⟨4,0⟩, ⟨13,0⟩, ⟨t,1⟩, ⟨s,1⟩}. *Balance* передает эту очередь *Rebuild*.

```
n0 ← 17 + t
t3 ← n0 + s
```

Rebuild извлекает из очереди элементы ⟨4,0⟩ и ⟨13,0⟩. Поскольку оба являются константами, они складываются, и в очередь помещается ⟨17,0⟩. Затем из очереди извлекаются ⟨17,0⟩ и ⟨t,1⟩; создается операция 17 + t; результату назначается новое имя, скажем n_0, и в очередь помещается элемент ⟨n_0,1⟩. Наконец, *Rebuild* извлекает из очереди ⟨n_0,1⟩ и ⟨s,1⟩ и создает операцию n_0 + s. Так как теперь очередь пуста, *Rebuild* сопоставляет результат корню дерева, t_3. Код показан во врезке.

Rebuild возвращается, *Balance* возвращается, и управление снова попадает вызову *Flatten* для t_{11}. *Flatten* продолжает работать, пока не встретит t_7, после чего рекурсивно вызывает *Balance*(t_7).

```
t7 ← x + w
```

Для t_7 *Flatten* порождает очередь {⟨x,1⟩, ⟨w,1⟩}. *Rebuild* извлекает из очереди ⟨x,1⟩ и ⟨w,1⟩, создает операцию x + w и, поскольку очередь пуста, сопоставляет результат корню, t_7, как показано во врезке.

```
n1  ← z + y
n2  ← n1 + t7
t11 ← n2 + t3
```

Rebuild возвращается, *Balance* возвращается, и управление возвращается *Flatten*(t_{11}). *Flatten* возвращает очередь {⟨z,1⟩, ⟨y,1⟩, ⟨t_7,2⟩, ⟨t_3,2⟩}. *Rebuild* создает операцию z + y и назначает ее значению имя n_1. Затем она создает операцию n_1 + t_7 и назначает ее значению имя n_2. Наконец, создает операцию n_2 + t_3 и сопоставляет ее значению корня дерева, t_{11}.

Rebuild возвращается, *Balance* возвращается, и управление снова попадает в цикл while. Вызов *Balance*(t_7) обнаруживает, что t_7 уже обработано. Затем вызывается *Balance*(t_3), которая обнаруживает, что t_3 уже обработано.

```
t10 ← t7 × t3
```

После этого в цикле while вызывается *Balance*(t_{10}). *Flatten* рекурсивно вызывает *Balance* для t_3 и t_7. В обоих случаях дерево

уже обработано. В этот момент *Flatten* возвращает очередь {⟨t_7,2⟩, ⟨t_3,2⟩}. *Rebuild* создает операцию $t_7 \times t_3$ и, поскольку очередь теперь пуста, сопоставляет ее корню дерева, t_{10}. *Rebuild* возвращается, *Balance* возвращается.

Наконец, на последней итерации цикла while вызывается *Balance*(t_6). *Flatten* встречает t_3, что заставляет ее вызвать *Balance*(t_3); т. к. t_3 уже обработано, вызов возвращается немедленно. *Flatten* возвращает очередь {⟨3,0⟩, ⟨v,1⟩, ⟨u,1⟩, ⟨t_3,2⟩}. Из этой очереди *Rebuild* создает операции, показанные во врезке. *Rebuild* возвращается, *Balance* возвращается, и цикл while завершается. Конечный результат показан на рис. 8.11.

```
n3 ← 3 × v
n4 ← n3 × u
t6 ← n4 × t3
```

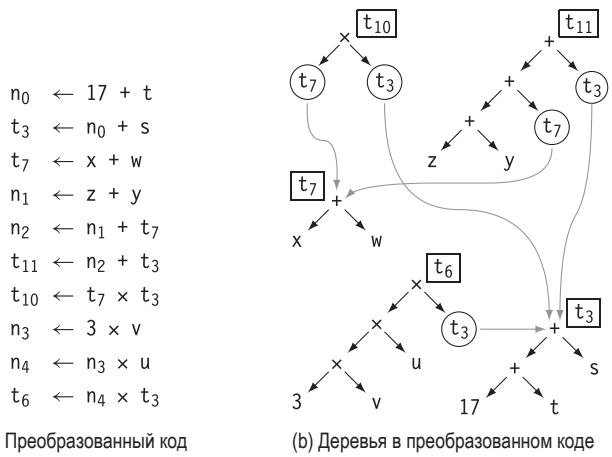


Рис. 8.11 Структура кода после балансировки

Различие между лесами деревьев выражений на рис. 8.10(b) и 8.11(b), на первый взгляд, не очевидно. Чтобы оценить реструктурированное дерево, сравним высоту дерева для t_6, включая дерево для t_3. В исходном коде высота дерева была равна 7, тогда как высота реструктурированного дерева равна 4. Аналогично высота исходного дерева для t_{11} была равна 5, а реструктурированного – 4.

И напоследок: читатель, возможно, заметил, что отдельные деревья на рис. 8.11(b) не выглядят сбалансированными. Алгоритм обеспечивает сбалансированность для всего блока, а не для отдельных деревьев. Этот эффект – простой и понятный пример различия между локальной оптимизацией и оптимизацией в более широком контексте.

Сбалансированность блока – результат всего одной строчки в коде *Flatten* на рис. 8.8. Для корня *x* *Flatten* помещает *x* в очередь с рангом, который был ранее назначен ему в *Balance*. Если бы мы изменили эту строчку, сделав ранг равным 1, то все поддеревья были бы сбалансированы. Рисунок, возможно, стал бы симпатичнее, но суммарная высота деревьев, ссылающихся на разделяемые поддеревья, оказалась бы больше, а степень параллелизма на уровне команд – меньше.

ИТОГИ РАЗДЕЛА

Локальная оптимизация производится на уровне одного простого блока кода. Блок переписывается на основании выведенной из него информации. По ходу дела необходимо сохранить взаимодействия блока с окружающим контекстом выполнения, в частности любые *наблюдаемые* значения, вычисленные в блоке.

Поскольку область ограничена одним блоком, локальные оптимизации могут полагаться только на свойства, справедливые для линейного кода. Например, локальная нумерация значений опирается на тот факт, что все операции в блоке выполняются в порядке, согласованном с линейным выполнением. Поэтому можно построить модель контекста, раскрывающую избыточности и константные выражения. Аналогично балансировка деревьев по высоте опирается на тот факт, что у блока есть только одна точка входа и одна точка выхода; это позволяет определить, какие подвыражения в блоке необходимо сохранить, а какие можно переупорядочить.

ВОПРОСЫ

1. Именование играет важную роль для эффективности ЛНЗ. Вчерне разработайте алгоритм, который переименовывает значения в простом блоке, так чтобы каждое определение в нем создавало уникальное имя, и позаботьтесь о том, чтобы каждое использование ссылалось на правильное новое имя.
2. Алгоритм балансировки деревьев по высоте назначает узлу *n* конечного выражения ранг, равный количеству неконстантных листьев, расположенных ниже него в конечном дереве. Как бы вы модифицировали алгоритм, чтобы он порождал ранги, соответствующие высоте *n* в дереве? Изменило бы это код, порождаемый алгоритмом?

8.5. Региональная оптимизация

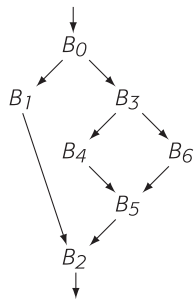
Неэффективность не ограничивается одиночными блоками. Код в одном блоке может содержать контекст, полезный для улучшения кода в другом блоке. Поэтому во многих оптимизациях исследуется контекст более широкий, чем один блок.

Региональные методы применяются к подмножествам графа потока управления, которые включают несколько блоков, но необязательно целую процедуру. Основное затруднение, возникающее при переходе от локальной оптимизации к региональной, – необходимость обрабатывать несколько путей в графе потока управления. Конструкция `if-then-else` вводит условно исполняемые пути. Циклы вводят циклический поток управления. Поскольку региональные методы исследуют больший контекст, чем локальные, они могут обнаружить возможности, невидимые локальным методам. Так как эти методы фокусируются на подмножестве ГПУ процедуры, зачастую они оказываются проще глобального подхода к той же проблеме.

В этом разделе рассматриваются два региональных метода в качестве иллюстрации. Первый, суперлокальная нумерация значений, обобщает ЛНЗ на несколько блоков. Второй, разворачивание цикла, применяется к одному вложенному циклу; разворачивание играло важную роль в обсуждении подпрограммы `dmxpy` в разделе 8.2.1.

8.5.1. Суперлокальная нумерация значений

Чтобы улучшить результаты ЛНЗ, компилятор может распространить ее область с одного простого блока на расширенный простой блок, или РПБ (см. раздел 8.3). Для обработки РПБ алгоритм должен произвести нумерацию значений на каждом пути через РПБ. Рассмотрим, к примеру, ГПУ на рис. 8.12(a). Он содержит три РПБ. Нетривиальным является РПБ (B_0, B_1, B_3, B_4, B_6). Два других РПБ тривиальны: (B_2) и (B_5). Региональная версия нумерации значений, которую мы будем называть *суперлокальной нумерацией значений* (СНЗ, *англ.* SVN), обрабатывает по одному РПБ за раз и распространяет информацию вниз по каждому пути в РПБ.



(a) Граф потока управления

B_0: $m_0 \leftarrow a_0 + b_0$
$n_0 \leftarrow a_0 + b_0$
$(a_0 > b_0) \rightarrow B_1, B_3$

B_1: $p_0 \leftarrow c_0 + d_0$
$r_0 \leftarrow c_0 + d_0$
$\rightarrow B_2$

B_2: $r_2 \leftarrow \phi(r_0, r_1)$
$y_0 \leftarrow a_0 + b_0$
$z_0 \leftarrow c_0 + d_0$
$\rightarrow exit$

B_3: $q_0 \leftarrow a_0 + b_0$
$r_1 \leftarrow c_0 + d_0$
$(a_0 > b_0) \rightarrow B_4, B_6$

B_4: $e_0 \leftarrow b_0 + 18$
$s_0 \leftarrow a_0 + b_0$
$u_0 \leftarrow e_0 + f_0$
$\rightarrow B_5$

B_5: $e_2 \leftarrow \phi(e_0, e_1)$
$u_2 \leftarrow \phi(u_0, u_1)$
$v_0 \leftarrow a_0 + b_0$
$w_0 \leftarrow c_0 + d_0$
$x_0 \leftarrow e_2 + f_0$
$\rightarrow B_2$

B_6: $e_1 \leftarrow a_0 + 17$
$t_0 \leftarrow c_0 + d_0$
$u_1 \leftarrow e_1 + f_0$
$\rightarrow B_5$

(b) Соответствующий низкоуровневый код

Рис. 8.12 Пример суперлокальной нумерации значений

Нетривиальный РПБ содержит три пути: $\langle B_0, B_1 \rangle$, $\langle B_0, B_3, B_4 \rangle$ и $\langle B_0, B_3, B_6 \rangle$. СНЗ рассматривает каждый из этих путей, как если бы он был линейным. Для обработки $\langle B_0, B_1 \rangle$ компилятор может применить ЛНЗ к B_0 и использовать результирующую хеш-таблицу как отправную точку для применения ЛНЗ к B_1. Такой же подход применим к путям $\langle B_0, B_3, B_4 \rangle$ и $\langle B_0, B_3, B_6 \rangle$ – нужно лишь обработать блоки каждого по порядку и перенести хеш-таблицы далее. Блоки, у которых несколько предшественников, например B_2 и B_5, начинают новые РПБ и потому не наследуют контекст от своих предшественников.

Эта схема позволяет достичь таких же результатов, как ЛНЗ, на более длинных путях благодаря рассмотрению пути в РПБ, как если бы это был один блок. Чтобы добавить примеру конкретики, рассмотрим код для каждого блока на рис. 8.12(b). Применение ЛНЗ к путям через РПБ открывает возможности, которые оставались скрытыми при рассмотрении одних лишь индивидуальных блоков.

- На пути $\langle B_0, B_1 \rangle$ ЛНЗ обнаруживает, что присваивания n_0 и r_0 избыточны. СНЗ обнаруживает те же избыточности.

- На пути $\langle B_0, B_3, B_4 \rangle$ ЛНЗ обнаруживает, что присваивание n_0 избыточно. СНЗ дополнительно находит, что присваивания q_0 и s_0 избыточны.
- На пути $\langle B_0, B_3, B_6 \rangle$ ЛНЗ обнаруживает, что присваивание n_0 избыточно. СНЗ дополнительно находит, что присваивания q_0 и t_0 избыточны.
- На путях B_5 и B_2 СНЗ вырождается в ЛНЗ.

```
m0 ← a0 + b0
n0 ← a0 + b0
q0 ← a0 + b0
r1 ← c0 + d0
e0 ← b0 + 18
s0 ← a0 + b0
u0 ← e0 + f0
```

Рассмотрение пути как одного блока

Трудность этого подхода в том, как сделать процесс эффективным. Очевидное решение – рассматривать каждый путь так, будто это один блок, притворившись, например, что код для $\langle B_0, B_3, B_4 \rangle$ выглядит как показанный во врезке. К сожалению, при таком подходе блок анализируется по одному разу для каждого включающего его пути. В случае расширенного блока (B_0, B_1, B_3, B_4, B_6) алгоритм проанализировал бы B_0 три раза, а B_3 дважды. Мы, конечно, хотим получить преимущества, проистекающие из рассмотрения более широкого контекста, но при этом желательно минимизировать накладные расходы на этапе компиляции. Ключ к разрешению конфликта лежит в учете древовидной структуры РПБ.

Чтобы сделать СНЗ эффективным, компилятор должен повторно использовать результаты блоков, встречающихся в роли префиксов на нескольких путях через РПБ. После обработки $\langle B_0, B_3, B_4 \rangle$ ему необходим механизм, который будет воссоздавать состояние в конце $\langle B_0, B_3 \rangle$, чтобы это состояние можно было повторно использовать для обработки B_6. Добиться этой цели компилятор может несколькими способами:

- он может запоминать состояние таблицы на границе каждого блока и восстанавливать его по мере необходимости;
- он может раскрутить действия блока, пройдя по блоку в обратном направлении и для каждой операции отменяя работу, проделанную на прямом проходе;
- он может использовать связку хеш-таблиц для реализации таблицы значений (см. раздел 4.5.1). При входе в блок создается новая таблица. Чтобы отменить действия блока, эта таблица блока удаляется.

СНЗ легко может оценить размер каждой таблицы по размерам блоков.

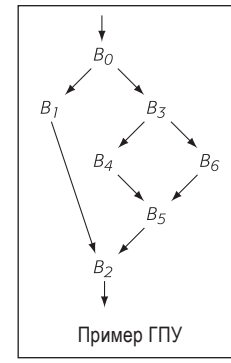
Хотя все три схемы работают, связка таблиц может дать простую и быструю реализацию, особенно если компилятор способен повторно использовать реализацию таблиц в фазе анализа.

На рис. 8.13 показан высокоуровневый набросок алгоритма СНЗ с использованием связки таблиц значений. Предполагается, что алгоритм ЛНЗ принимает блок и связку таблиц в качестве параметров. Для каждого блока *b* СНЗ выделяет память под таблицу значений *b*, связывает ее с таблицей значений предшествующего блока, как если бы это была объемлющая область, и вызывает ЛНЗ для блока *b* с этой новой таблицей. Когда ЛНЗ возвращается, СНЗ должен решить, как обрабатывать каждый из преемников *b*.

```
// Начать процесс
WorkList ← { начальный блок }
Empty ← новая таблица для Block
while (WorkList не пуст) do
   удалить Block из WorkList
   SVN(Block, Empty)
освободить Empty
```

```
// Алгоритм суперлокальной нумерации значений
SVN(Block, Table)
   t ← новая таблица для Block
   связать Table как объемлющую область для t
   LVN(Block, t)
   for каждого преемника s блока Block do
      if |preds(s)| = 1 then
         SVN(s, t)
      else if s еще не был обработан then
         добавить s в WorkList
   освободить t
```

Рис. 8.13 Алгоритм суперлокальной нумерации значений



Для *s*, преемника *b*, возможны два случая. Если у *s* только один предшественник, *b*, то СНЗ должен использовать контекст, аккумулированный при обработке *b*. Таким образом, СНЗ применяется к *s* с таблицей от *b*. Если у *s* несколько предшественников, то СНЗ должен начать обработку *s* с новым контекстом, поэтому СНЗ добавляет *s* в *WorkList*. Позже внешний цикл найдет его и вызовет СНЗ для *s* с пустой таблицей. Важно реализовать *Worklist* как множество, чтобы каждый блок обрабатывался только один раз (см. раздел B.2).

Остается одна трудность. Номер значения имени записывается в таблицу значений, ассоциированную с первой операцией в определящем его РПБ. Это может свести на нет наши усилия по использованию механизма областей видимости. В нашем примере ГПУ, если бы имя *x* было определено в каждом из B_0, B_4 и B_6, то номер его значения был бы записан в таблицу для B_0. В процессе обработки B_4 СНЗ записал бы новый номер значения из B_4 в таблицу для B_0. После того как СНЗ удалит таблицу для B_4 и создаст новую таблицу для B_6, номер значения из определения в B_4 останется.

Чтобы обойти это затруднение, компилятор может применять СНЗ к представлению, в котором каждое имя определено только один раз. В разделе 4.6.2 мы видели, что форма SSA обладает нужным свойством; каждое имя определено ровно в одной точке кода. Использование формы SSA гарантирует, что СНЗ записывает номер значения для некоторого определения в ту таблицу, которая соответствует блоку, содержащему это определение. Располагая формой SSA, СНЗ может отменить все действия блока, удалив его таблицу значений. Этот поступок возвращает набор таблиц в состояние при выходе из предшественника ГПУ блока. Как обсуждалось в разделе 8.4.1, использование формы SSA может также повысить эффективность ЛНЗ.

Применение алгоритма на рис. 8.13 к коду на рис. 8.12 порождает последовательность действий, показанную во врезке на следующей странице. Ал-

1. *Создать область для* B_0
2. *Применить ЛНЗ к* B_0
3. *Создать область для* B_1
4. *Применить ЛНЗ к* B_1
5. *Добавить* B_2 *в WorkList*
6. *Удалить область* B_1
7. *Создать область для* B_3
8. *Применить ЛНЗ к* B_3
9. *Создать область для* B_4
10. *Применить ЛНЗ к* B_4
11. *Добавить* B_5 *в WorkList*
12. *Удалить область* B_4
13. *Создать область для* B_6
14. *Применить ЛНЗ к* B_6
15. *Удалить область* B_6
16. *Удалить область* B_3
17. *Удалить область* B_0
18. *Создать область для* B_5
19. *Применить ЛНЗ к* B_5
20. *Удалить область* B_5
21. *Создать область для* B_2
22. *Применить ЛНЗ к* B_2
23. *Удалить область* B_2

Действия, предпринимаемые СНЗ в примере

горитм начинает работу с B_0 и переходит к B_1. В конце B_1 он посещает B_2, понимает, что у B_2 несколько предшественников, и добавляет его в рабочий список. Затем он возвращается и обрабатывает B_3 и B_4. В конце B_4 он добавляет B_5 в рабочий список. Потом он возвращается к B_3 и обрабатывает B_6. В этой точке управление возвращается циклу while, который вызывает СНЗ для двух одиночных блоков, находящихся в рабочем списке, B_5 и B_2.

СНЗ обнаруживает и устраняет избыточные вычисления, которых ЛНЗ обнаружить не может. В нашем примере он находит, что присваивания q_0, s_0 и t_0 избыточны из-за определений в более ранних блоках. ЛНЗ, будучи ограничен локальной областью, не может найти такие избыточности.

С другой стороны, у СНЗ есть свои ограничения. Он не может найти избыточности в B_5 и B_2. Читателю довольно беглого взгляда, чтобы увидеть, что присваивания в этих блоках избыточны. Но поскольку у этих блоков несколько предшественников, СНЗ не может перенести в них контекст. Так что эти возможности остались неиспользованными. Чтобы их обнаружить, алгоритму нужно больше контекста.

8.5.2. Разворачивание цикла

Разворачивание цикла – пожалуй, самое старое, самое простое и самое известное преобразование цикла. Чтобы развернуть цикл, компилятор повторяет его тело и немного изменяет логику, управляющую числом итераций. Для иллюстрации рассмотрим вложенный цикл из подпрограммы `dmxpy` в примере из раздела 8.2.

```
    do 60 j = 1, n2
       do 50 i = 1, n1
          y(i) = y(i) + x(j) * m(i,j)
50     continue
60  continue
```

Слияние циклов
Процесс объединения тел двух циклов в одно называется *слиянием* (fusion). Слияние безопасно, когда каждое определение и каждое использование в результирующем цикле имеют то же значение, какое было у них в исходных циклах.

Компилятор может развернуть как внутренний, так и внешний цикл. Результат разворачивания внутреннего цикла на четыре итерации показан на рис. 8.14(а). Разворачивание внешнего цикла на четыре итерации порождает четыре внутренних цикла; если затем компилятор объединит тела этих внутренних циклов – это преобразование

называется *слиянием циклов*, – то получится код, подобный показанному на рис. 8.14(b). Сочетание разворачивания внешнего цикла с последующим слиянием внутренних циклов часто называют *разворачиванием со слиянием* (unroll-and-jam).

```
  do 60 j = 1, n2
    nextra = mod(n1,4)
    if (nextra .ge. 1) then
      do 49 i = 1, nextra
        y(i) = y(i) + x(j) * m(i,j)
49       continue

  do 50 i = nextra + 1, n1, 4
    y(i)   = y(i) + x(j) * m(i,j)
    y(i+1) = y(i+1) + x(j) * m(i+1,j)
    y(i+2) = y(i+2) + x(j) * m(i+2,j)
    y(i+3) = y(i+3) + x(j) * m(i+3,j)
50  continue
60 continue
```

(a) Разворачивание внутреннего цикла на 4 итерации

```
  nextra = mod(n2,4)
  if (nextra .ge. 1) then
    do 59 j = 1, nextra
      do 49 i = 1, n1
        y(i) = y(i) + x(j) * m(i,j)
49       continue
59     continue
  do 60 j = nextra+1, n2, 4
    do 50 i = 1, n1
      y(i) = y(i) + x(j) * m(i,j)
      y(i) = y(i) + x(j+1) * m(i,j+1)
      y(i) = y(i) + x(j+2) * m(i,j+2)
      y(i) = y(i) + x(j+3) * m(i,j+3)
50     continue
60   continue
```

(b) Разворачивание внешнего цикла на 4 итерации, слияние внутренних циклов

Рис. 8.14 Разворачивание вложенного цикла в dmxpy

В каждом случае преобразованному коду необходим короткий предварительный цикл, чтобы число итераций в последующем развернутом цикле было кратно четырем. Если границы цикла известны на этапе компиляции, то компилятор сам может определить, нужен ли такой пролог.

В данном конкретном примере вложенного цикла разворачивание внутреннего и внешнего циклов дает разные результаты. Разворачивание внутреннего цикла уменьшает число выполняемых в нем последовательностей проверки и перехода. С другой стороны, разворачивание внешнего цикла с последующим слиянием внутренних циклов не только уменьшает число таких последовательностей, но и способствует повторному использованию `y(i)` и последовательному доступу к `x` и `m`. Увеличенная степень повторного использования изменяет отношение числа арифметических операций к числу операций с памятью в цикле; несомненно, автор `dmxpy` имел этот эффект в виду, когда вручную оптимизировал код, как показано на рис. 8.1. Как обсуждается ниже, каждый подход может давать и косвенные выгоды.

> Доступ к `m` является последовательным, потому что в FORTRAN массивы хранятся по столбцам.

Источники улучшений и ухудшений

Разворачивание цикла имеет прямые и косвенные последствия для кода, порождаемого компилятором для данного цикла. Окончательная производительность цикла зависит от тех и от других.

Прямым последствием является то, что разворачивание должно уменьшить число операций, необходимых для завершения цикла. Изменения потока управления уменьшают общее число последовательностей проверки и перехода. Разворачивание может создать возможности для повторного использования в теле цикла, уменьшив трафик в памяти. Наконец, если цикл содержит циклическую цепочку операций копирования, то разворачивание может устранить копирование, как показано в упражнении 8.6.

Однако разворачивание цикла может увеличить размер кода – как промежуточного, так и исполняемого. Рост ПП увеличивает время компиляции. Рост исполняемого кода мало на что влияет до тех пор, пока цикл помещается в кеш команд, а после этого деградация, скорее всего, перевесит прямые выгоды.

Компилятор может также разворачивать цикл ради косвенных эффектов. Основной побочный эффект разворачивания – увеличение числа операций в теле цикла. Это изменение может лечь в основу других оптимизаций, как то:

- увеличение степени параллелизма в теле цикла, возможно, позволит лучше планировать команды. При наличии большего числа независимых операций у планировщика больше шансов загрузить работой несколько функциональных блоков и компенсировать задержку многотактных операций, таких как условные переходы и доступ к памяти;
- разворачивание может переместить соседние операции доступа к памяти в одну итерацию, и тогда у компилятора появится шанс спланировать их вместе. Это может улучшить локальность или открыть возможность для операций с несколькими словами;
- разворачивание может вскрыть межитерационные избыточности, которые трудно обнаружить в исходном коде. В обеих версиях кода на рис. 8.14 адресные выражения повторно используются на всех итерациях исходного цикла. В развернутых циклах алгоритм локальной нумерации значений найдет и устранит эти избыточности, а в исходных не заметит их;
- развернутый цикл можно оптимизировать иначе, чем исходный. Например, увеличение количества появлений переменной в цикле может изменить веса в коде выбора регистров для выгрузки (см. раздел 13.4). Изменение порядка выгрузки регистров может весьма существенно повлиять на время выполнения цикла.

Заметим, однако, что в теле развернутого цикла может понадобиться больше регистров, чем в теле исходного. Если повышенный спрос на регистры приводит к увеличению количества операций выгрузки и восстановления, то результирующий трафик в памяти может перевесить выгоды разворачивания.

Эти косвенные взаимодействия гораздо труднее охарактеризовать и понять, чем прямые эффекты. Они могут привести к значительному улучшению производительности, но могут также стать причиной ее ухудшения. Трудность предсказания таких косвенных последствий побудила некоторых исследова-

телей выступить за адаптивный подход к выбору коэффициента разворачивания; в таких системах компилятор пробует несколько коэффициентов и измеряет характеристики производительности получившегося кода.

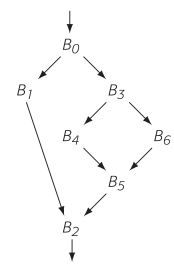
ИТОГИ РАЗДЕЛА

Оптимизации, ориентированные на области, большие блока, но меньшие всей процедуры, могут находить и использовать возможности, ускользнувшие от внимания чисто локального алгоритма. У таких региональных методов долгая история в литературе и на практике.

В некоторых случаях, например CH3, распространение на региональную область обеспечивает улучшенную оптимизацию за небольшую дополнительную плату. И CH3, и суперлокальное планирование команд (см. раздел 12.4.1) имеют эффективные реализации.

В других случаях региональные оптимизации применяют более агрессивные методы к небольшим участкам кода, где возможен большой выигрыш. К этому разряду относятся оптимизации циклов. Для многих оптимизаций циклов требуется тщательный анализ, чтобы доказать их безопасность. Поскольку очень многие программы проводят значительное время внутри циклов, авторы компиляторов пришли к выводу, что такие методы окупаются сторицей.

ВОПРОСЫ



1. Суперлокальная нумерация значений обобщает локальную нумерацию значений на расширенные простые блоки путем изобретательного использования хеш-таблиц. Рассмотрите проблемы, которые могут возникнуть при обобщении алгоритма балансировки деревьев по высоте на суперлокальную область оптимизации.
   a. Как ваш алгоритм будет обрабатывать один путь через РПБ, например $\langle B_0, B_3, B_4 \rangle$, в графе управления потоком, показанном справа?
   b. Какие трудности возникнут, если алгоритм попытается обработать путь $\langle B_0, B_3, B_6 \rangle$ после обработки пути $\langle B_0, B_3, B_4 \rangle$?
2. Разворачивание цикла всегда уменьшает связанные с циклом накладные расходы. Но, несмотря на это, компиляторы не разворачивают каждый цикл. Какие факторы должен учитывать компилятор, принимая решение о разворачивании конкретного цикла?

8.6. Глобальная оптимизация

Глобальная оптимизация применяется ко всей процедуре или методу. Поскольку эта область включает циклические управляющие конструкции, например циклы, эти методы, как правило, выполняют полный анализ, прежде чем переписывать код.

В этом разделе представлены два примера глобального анализа и оптимизации. Первый, нахождение неинициализированных, но живых переменных, строго говоря, не является оптимизацией, а применяет глобальный анализ данных для обнаружения полезной информации о потоке значений в процедуре. Мы воспользуемся этим обсуждением, чтобы ввести понятие *живучести*, которое играет важную роль во многих методах оптимизации, включая балансировку деревьев по высоте (раздел 8.4.2), построение формы

SSA (раздел 9.3) и распределение регистров (глава 13). Во втором примере, глобальной реорганизации кода, используется информация о профиле, собранная путем выполнения откомпилированного кода, для изменения размещения исполняемого кода в памяти.

8.6.1. Нахождение неинициализированных переменных с помощью анализа живучести

Если некоторая переменная *v* может использоваться в процедуре до того, как была инициализирована, говорят, что *v* потенциально не инициализирована в этой точке. Использование неинициализированных переменных всегда является признаком логической ошибки в компилируемой процедуре. Обнаружив такую ситуацию, компилятор должен уведомить о ней программиста.

Живая переменная
Говорят, что переменная *v жива* в точке *p*, если существует путь из *p* к месту использования *v*, на котором *v* не переопределяется.

Найти места потенциального использования неинициализированных переменных можно путем вычисления информации о *живучести*. Говорят, что переменная *v жива* в точке *p*, если в ГПУ существует путь из *p* к месту использования *v*, на котором *v* не переопределяется. Мы будем кодировать информацию о живучести, вычисляя для каждого блока *b* в процедуре множество LiveOut(*b*), содержащее все переменные, которые живы при выходе из *b*.

Концептуально компилятор может найти неинициализированные переменные, добавив искусственную входную вершину n_0 в ГПУ и ребра из n_0 в каждую фактическую входную вершину ГПУ. Тогда LiveOut(n_0) будет содержать в точности те переменные, которые могут использоваться до определения.

На практике, поскольку n_0 никак не влияет на множества других вершин ГПУ, компилятору нет нужды представлять n_0. Он может вычислить LiveOut(n_0), игнорируя существование n_0. После этого вычисления он может вычислить, какое значение имело бы LiveOut(n_0), если бы n_0 существовала. Такой подход должен быть немного быстрее, поскольку позволяет избежать итеративных вычислений LiveOut(n_0).

Анализ потока данных
один из видов анализа на этапе компиляции, позволяющий рассуждать о потоке значений во время выполнения.

Вычисление множеств LiveOut – пример *глобального анализа потока данных* – семейства методов, позволяющих на этапе компиляции рассуждать о потоке значений во время выполнения. Задачи анализа потока данных обычно ставятся как решение системы уравнений над множествами вершин и ребер графа.

Постановка задачи о потоке данных

Анализ живых переменных – классическая задача глобального анализа потока данных. Для каждой вершины *n* ГПУ процедуры компилятор вычисляет

множество LiveOut(n), которое содержит все переменные, являющиеся живыми при выходе из блока, соответствующего n.

Задачи о потоке данных часто ставятся в виде уравнения над множествами, ассоциированными с вершинами ГПУ. Для живых переменных следующее уравнение определяет множество LiveOut для вершины n:

$$\text{LiveOut}(n) = \bigcup_{m \in succ(n)} \left(\begin{array}{l} \text{UEVar}(m) = \cup \\ (\text{LiveOut}(n) \cap \overline{\text{VarKill}(m)}) \end{array} \right),$$

где $succ(n)$ – множество преемников n в ГПУ. До начала анализа следует выполнить инициализацию LiveOut(n) = ∅, ∀n.

Здесь UEVar(m) содержит экспонированные сверху переменные в m, т. е. переменные, которые используются в m до того, как они определены в m. VarKill(m) содержит переменные, определенные в m; надчерк над множеством VarKill(m) обозначает его логическое дополнение, т. е. множество всех переменных, не определенных в m. Компилятор должен вычислить UEVar(m) и VarKill для каждой вершины ГПУ и только после этого сможет решить уравнения LiveOut.

Уравнение описывает определение живучести интуитивно понятным способом. LiveOut(n) – это просто объединение переменных, которые живы в начале некоторого блока m, непосредственно следующего за n в ГПУ. Определение требует, чтобы значение было живо хотя бы на одном, а не на всех путях. Таким образом, для формирования LiveOut(n) вклады преемников n в ГПУ объединяются. Вклад одного преемника m вершины n равен

$$\text{UEVar}(m) \cup (\text{LiveOut}(m) \cap \overline{\text{VarKill}(m)}).$$

Переменная v жива при входе в m, если выполняется одно из двух условий. Она может быть использована в m до того, как определена в m: $v \in \text{UEVar}(m)$. Она может быть жива при выходе из m и пройти сквозь m неповрежденной, потому что в m она не переопределяется: $v \in \text{LiveOut}(m) \cap \overline{\text{VarKill}(m)}$. Объединение этих двух членов дает вклад m в LiveOut(n). Чтобы вычислить LiveOut(n), анализатор вычисляет вклад каждого преемника n, о чем свидетельствует обозначение $m \in succ(n)$, и применяет к ним операцию объединения.

Решение задачи о потоке данных

Чтобы вычислить множества LiveOut для процедуры и ее ГПУ, компилятор может воспользоваться трехшаговым алгоритмом.

1. *Построить ГПУ*. Компилятор перебирает блоки и строит граф (см. раздел 4.4.4). Каждая вершина представляет один блок.
2. *Собрать начальную информацию*. Для каждого блока b компилятор вычисляет UEVar(b) и VarKill(b), как показано на рис. 8.15(a).
3. *Решить систему уравнения для LiveOut*(b). На рис. 8.15(b) показан простой итеративный алгоритм нахождения неподвижной точки, решающий эту систему уравнений.

```
// каждый блок включает k операций
// вида "x ← y op z"
for каждого блока b do
  // Инициализировать множества для b
  UEVAR(b) ← ∅
  VARKILL(b) ← ∅
  for i ← 1 to k do
    if y ∉ VARKILL(b) then
      добавить y в UEVAR(b)
    if z ∉ VARKILL(b) then
      добавить z в UEVAR(b)
    добавить x в VARKILL(b)
```

(a) Сбор начальной информации

```
// предполагается, что ГПУ включает N блоков,
// пронумерованных от 0 до N - 1
for i ← 0 to N-1 do
  LIVEOUT( i ) ← ∅

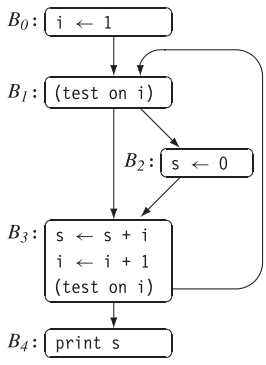
changed ← true
while (changed) do
  changed ← false
  for i ← 0 to N-1 do
    пересчитать LIVEOUT(i)
    if LIVEOUT(i) изменилось then
      changed ← true
```

(b) Решение системы уравнений

Рис. 8.15 Итеративный анализ живучести

Далее в этом разделе мы подробно разберем пример вычисления LiveOut. В разделе 9.2 мы рассмотрим итеративный анализ потока данных более углубленно.

Сбор начальной информации



Пример ГПУ

Для вычисления LiveOut анализатор должен знать множества UEVar и VarKill для каждого блока. Оба можно вычислить на одном проходе. Для каждого блока анализатор инициализирует эти множества, делая их пустыми. Затем он проходит по блоку от начала к концу и обновляет UEVar и VarKill, чтобы отразить воздействие каждой операции. На рис. 8.15(a) показаны детали этого вычисления для операций вида $x \leftarrow y \langle op \rangle z$.

Рассмотрим ГПУ, показанный во врезке. Он состоит из простого цикла, который содержит конструкцию `if-then`. Код абстрагирует детали сравнений и переходов. В таблице под ГПУ показаны начальные множества UEVar и VarKill.

	UEVAR	VARKILL
B_0	∅	{i}
B_1	{i}	∅
B_2	∅	{s}
B_3	{s,i}	{s,i}
B_4	{s}	∅

Начальные множества

Решение уравнений LiveOut

Зная множества UEVar и VarKill для каждой вершины ГПУ, компилятор применяет алгоритм на рис. 8.15(b), чтобы вычислить множества LiveOut для каждой вершины. Вначале все множества LiveOut пусты (равны ∅). Затем вычисляется множество LiveOut для каждого блока по порядку. Процесс продолжается, пока множества LiveOut не перестанут изменяться.

В таблице на рис. 8.16 показаны множества LiveOut на каждой итерации решателя. На итерации 0 показаны начальные значения. На итерации 1 вычисляется начальное приближение множеств LiveOut. Поскольку блоки при

этом обрабатываются в порядке возрастания меток, B_0, B_1 и B_2 получают значения, основанные исключительно на множествах UEVar их преемников в ГПУ. Достигнув B_3, алгоритм уже вычислил приближение LiveOut(B_1), поэтому значение, вычисленное им для B_3, отражает вклад нового значения LiveOut(B_1). LiveOut(B_4) пусто, как и подобает выходному блоку.

	LiveOut(*n*)				
Итерация	B_0	B_1	B_2	B_3	B_4
–	Ø	Ø	Ø	Ø	Ø
1	{i}	{s,i}	{s,i}	{s,i}	Ø
2	{s,i}	{s,i}	{s,i}	{s,i}	Ø
3	{s,i}	{s,i}	{s,i}	{s,i}	Ø

Рис. 8.16 Пример вычисления LiveOut

На второй итерации значение s добавляется в LiveOut(B_0) вследствие его присутствия в приближении LiveOut(B_1). Больше никаких изменений нет. Третья итерация не изменяет множества LiveOut, и алгоритм завершается.

Порядок обработки блоков этим алгоритмом влияет на значения промежуточных множеств. Если бы алгоритм посещал блоки в порядке убывания меток, то потребовался бы только один проход. Однако конечные значения множеств LiveOut не зависят от порядка вычислений. Итеративный решатель на рис. 8.15 вычисляет неподвижную точку системы уравнений LiveOut.

> Теория итеративного анализа потока данных гарантирует, что у этой системы уравнений всегда имеется единственное решение (см. также раздел 9.2).

Этот решатель задачи итеративного анализа потока данных является алгоритмом с неподвижной точкой. Он останавливается, потому что (1) множества LiveOut конечны и (2) повторное вычисление LiveOut для любого блока может породить либо то же самое, либо большее множество. Множества UEVar и VarKill постоянны на всех итерациях. Множества LiveOut увеличиваются монотонно, а т. к. их размер ограничен, алгоритм обязательно остановится.

Нахождение неинициализированных переменных

Вычислив множества LiveOut для всех вершин ГПУ, компилятор должен вычислить множество переменных, которые живы при входе в любую входную вершину ГПУ. Это множество эквивалентно LiveOut(n_0), где n_0 – искусственная входная вершина, из которой ребра ведут во все настоящие входные вершины.

В нашем примере ГПУ имеется только одна входная вершина, B_0, поэтому LiveOut(n_0) равно

$$\text{UEVar}(B_0) \cup (\text{LiveOut}(B_0) \cap \overline{\text{VarKill}(B_0)}) = \{ \text{s} \}.$$

Нетрудно видеть, что s не инициализирована на пути $\langle B_0, B_1, B_3 \rangle$. При таком подходе идентифицируются переменные, которые потенциально могут быть

использованы без предварительной инициализации. Компилятор должен распознать эту ситуацию и сообщить о ней программисту. Однако по нескольким причинам возможны ложноположительные тревоги.

```
 ...
 p = &x;
*p = 0;
 ...
 x = x+1;
```

- Если переменная *v* неоднозначна, то она могла бы быть инициализирована под другим именем. Анализ живучести не свяжет инициализацию с использованием. Такая ситуация может возникнуть, когда указателю присвоен адрес локальной переменной, как во фрагменте кода во врезке.

```
main() {
  int s, i = 1;
  while (i<=80) {
    if (i==1)
      s = 0;
    s = s + i++;
  }
  printf("%d",s);
}
```

- Если переменная *v* существует до вызова текущей процедуры, то, возможно, она была ранее инициализирована, но анализатор этого не видит. Такое возможно для статических переменных в текущей области оптимизации или для переменных, объявленных вне текущей области.
- Уравнения анализа живучести могут найти путь от точки входа в процедуру до места использования *v*, на котором *v* не определена. Если этот путь не реализуется во время выполнения, то *v* все равно появится в LiveOut(n_0), пусть даже ни при каком выполнении неинициализированное значение не используется. Например, в программе на C во врезке переменная `s` всегда инициализируется до использования, и тем не менее `s` ∈ LiveOut(n_0).

Пример цикла while во врезке иллюстрирует одно из фундаментальных ограничений анализа потока данных: предполагается, что все пути через ГПУ реализуются во время выполнения. Но в этом примере такое предположение излишне консервативно. Единственный путь в ГПУ, который ведет на использование неинициализированной переменной, начинается в точке входа `main` в цикл, обходит инициализацию `s` и выполняет инкремент `s`. Но по такому пути выполнение никогда не пойдет, потому что `i` должна иметь значение `1` на первой итерации цикла. Однако решатель уравнений LiveOut не сможет установить этот факт, потому что в нем не заложен механизм, позволяющий понять, что путь не реализуем.

Предположение о том, что все пути в ГПУ реализуемы, значительно уменьшает стоимость анализа. Но вместе с тем это предположение оказывается причиной потери точности вычисления множеств. Чтобы установить, что `s` инициализирована на первой итерации цикла while, компилятор должен был бы объединить анализ трассировки индивидуальных путей с какой-то формой распространения констант и с анализом живучести. Для решения этой задачи в общем виде понадобилось бы символическое вычисление частей кода в процессе анализа, что является гораздо более дорогостоящим предприятием.

Осложнение иного рода возникает в связи с вызовом процедур. Если процедура вызывает другую процедуру, то анализатор должен учитывать возможные побочные эффекты этого вызова. В результате вызова *s* какая-то гло-

бальная переменная *v* могла либо использоваться, либо модифицироваться. (Аналогичные проблемы возникают, когда *v* передается по ссылке или когда вызываемая сторона вложена в вызывающую.) В отсутствие более точной информации о вызываемой стороне анализатор должен предполагать, что любая переменная, которая может быть модифицирована, будет модифицирована, и что любая переменная, которая может быть использована, будет использована. Такие предположения безопасны в том смысле, что описывают худший случай.

Другие виды использования живых переменных

Компиляторы используют живучесть во многих других контекстах.

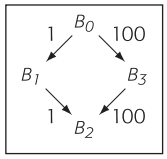
- Компилятор может использовать информацию о живучести, чтобы выявить бесполезные операции сохранения. Операция, сохраняющая однозначную скалярную переменную *v* в памяти, бесполезна, если *v* не является живой в месте сохранения. Бесполезные операции могут быть удалены.
- Живая информация может улучшить скорость и точность построения формы SSA; значение нуждается в φ-функции только в том блоке, где оно живо. Устранение неживых φ-функций может значительно уменьшить число вставляемых компилятором φ-функций.
- Информация о живых переменных крайне важна для глобального распределения регистров (см. раздел 13.4). Распределителю не нужно сохранять значения в регионах, где они не являются живыми. Информация о живучести лежит в основе рассуждений о том, какие значения хранить в регистрах.

В различных контекстах живучесть вычисляется для разных множеств имен. Мы обсудили вычисление LiveOut с неявным множеством имен переменных. В процессе глобального распределения регистров компилятор вычисляет множества LiveOut для тщательно сконструированного множества значений: они называются *интервалами жизни*.

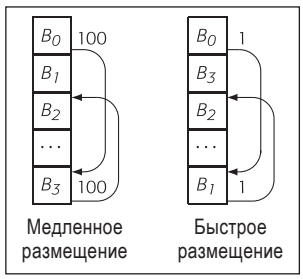
Интервал жизни
Интервалом жизни значения называется множество операций, расположенных между местами его определения и использования.

8.6.2. Глобальное размещение кода

Во многих процессорах стоимость условного перехода асимметрична; стоимость перехода по неявной ветви меньше, чем по выбранной. Для каждого условного перехода имеется два конечных простых блока, и компилятор может выбрать, какой из них будет лежать на неявной ветви, а какой – на выбранной. Оптимизация глобального размещения кода неявно опирается на два наблюдения: что стоимость условных переходов асимметрична и что частоты принятия некоторых решений в коде тоже неодинаковы.



Рассмотрим ГПУ, показанный во врезке. Путь (B_0, B_3) выполняется в 100 раз чаще, чем путь (B_0, B_1). Если стоимость условного перехода асимметрична, то компилятор должен использовать менее дорогую ветвь для (B_0, B_3). Если частоты выбора (B_0, B_1) и (B_0, B_3) примерно одинаковы, то выбор в этом примере не особо важен.



Во врезке показаны два разных размещения этого кода. В «медленном» размещении для реализации (B_0, B_1) используется неявная ветвь, а для реализации (B_0, B_3) – выбранная. В «быстром» размещении принято противоположное решение. Если неявная ветвь быстрее выбранной, то при «быстром» размещении менее дорогая ветвь будет использоваться в 100 раз чаще.

Компилятор может воспользоваться асимметрией стоимости условных переходов. Если компилятор знает относительные частоты переходов в процедуре, то может разместить код так, чтобы добиться более быстрого выполнения.

Для реализации глобального размещения кода компилятор переупорядочивает простые блоки процедуры, соблюдая два принципа.

- Во-первых, компилятор должен размещать наиболее вероятные пути выполнения на неявных ветвях. То есть по возможности непосредственно за блоком должен быть расположен его наиболее частый преемник.

  У такого размещения два преимущества. Чем больше доля переходов по более быстрому (неявному) пути, тем меньше общее время выполнения. Кроме того, размещение соседних блоков по соседним адресам виртуальной памяти должно улучшить локальность кеша команд.
- Во-вторых, компилятор должен переместить редко исполняемый (холодный) код в конец процедуры. Тогда часто исполняемый (горячий) код окажется рядом с другим горячим кодом и далеко от холодного кода.

Сбор данных профиля

Данные профиля играют важную роль в таких оптимизациях, как глобальное размещение кода и встраивание (см. раздел 8.7.1). К этой задаче есть несколько подходов:

- ***оснащение исполняемого файла.*** Компилятор может генерировать код подсчета некоторых событий, например входов в процедуры и выходов из них или выбранных ветвей переходов. В конце выполнения данные записываются во внешний файл и обрабатываются другим инструментом;
- ***прерывания таймера.*** Инструмент профилирования может прерывать выполнение программы через регулярные небольшие промежутки времени. Инструмент строит гистограмму мест программы, в которых происходили прерывания. А после завершения программы другой инструмент строит профиль;
- ***счетчики производительности.*** Большинство процессоров поддерживают аппаратные счетчики для регистрации конкретных событий, например выбранных ветвей переходов. Система времени выполнения может использовать счетчики для построения точных данных профиля.

Разные подходы порождают слегка различающиеся данные. Оснащенный исполняемый файл может измерять много свойств на этапе выполнения. Для системы на основе прерываний таймера накладные расходы могут быть ниже, но она находит только часто исполняемые предложения (а не пути, по которым до них дошло выполнение). Аппаратные счетчики точны и эффективны, но зависят от особенностей конкретного процессора.

Все описанные подходы работают. Каждый требует кооперации между компилятором и профилировщиком по таким вопросам, как форматы данных, размещение кода в памяти и методы отображения адресов на предложения исходного кода или операции ПП.

Такое размещение может уменьшить размер рабочего множества горячего кода, что, в свою очередь, улучшает локальность в кеше и в виртуальной памяти.

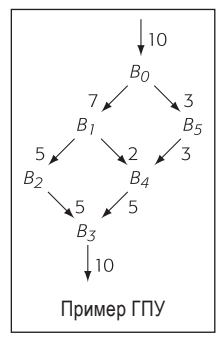
После размещения код должен выполнять более длинные последовательности операций без отвлечения – перехода по выбранной ветви, приостановки из-за отказа кеша команд или отказа страницы.

У размещения кода, как и у глобальных оптимизаций, имеются отдельные этапы анализа и преобразования. На этапе анализа вычисляются оценки части выполнения переходов. На этапе преобразования эти данные, выраженные в виде весов ребер ГПУ, используются для определения часто выполняемых путей. Затем на основе построенной модели изменяется порядок простых блоков.

Получение данных профиля

Для глобального размещения кода компилятору необходимы оценки относительных частот выполнения каждого ребра ГПУ. Эту информацию он может получить, выполнив профилирующий прогон кода: откомпилировать всю программу, выполнить ее под управлением профилировщика на репрезентативных данных и дать компилятору доступ к результирующему профилю. Компилятор сможет получить нужную информацию из модели выполнения программы; такие модели варьируются от простых до весьма изощренных, характеризуемых разной точностью.

Точнее, компилятору нужно знать счетчики выполнения для ребер ГПУ. ГПУ во врезке иллюстрирует, почему реберные счетчики для размещения кода важнее, чем счетчики блоков. Из счетчиков выполнения, показанных в виде меток на ребрах, мы видим, что каждый из блоков B_0 и B_3 выполняется десять раз. Путь $\langle B_0, B_1, B_2, B_3 \rangle$ выполняется чаще, чем любой другой путь в этом фрагменте ГПУ. Например, реберные счетчики позволяют предположить, что переход на B_2 в конце B_1 лучше сделать неявной ветвью, чем выбранной.



Пример ГПУ

Использование счетчиков выполнения для блоков, а не ребер, может приводить к неверным выводам; счетчик бло-

ка может объединять счетчики с нескольких путей. В ГПУ счетчики для обоих блоков B_2 и B_4 равны 5. Для ветвления в конце B_1 эти счетчики говорят, что пути в B_2 и B_4 одинаково важны, тогда как реберные счетчики показывают, что (B_1, B_2) выполняется чаще, чем (B_1, B_4). Алгоритм размещения кода ранжирует ребра по частоте выполнения; таким образом, повышенная точность реберных счетчиков напрямую влияет на качество результатов.

Построение горячих путей в ГПУ в виде цепочек

Чтобы определиться с размещением кода, компилятор находит *горячие пути* в ГПУ – пути, содержащие наиболее часто исполняемые ребра. Каждый путь – это цепочка, состоящая из одного или нескольких блоков. Алгоритм назначает приоритет каждому пути, приоритеты направляют процесс переупорядочения кода.

Компилятор может использовать жадный алгоритм для нахождения горячих путей, как показано на рис. 8.17. Вначале он создает для каждого блока вырожденную цепочку, содержащую только этот блок. Всем вырожденным цепочкам назначается высокий приоритет, например равный количеству ребер ГПУ или наибольшему целому числу.

```
E ← |edges|
for каждого блока b do
   создать вырожденную цепочку d для b
   priority(d) ← E
P ← 0
for каждого ребра ГПУ ⟨x,y⟩, x ≠ y, в порядке возрастания частот do
   if x является концом цепочки a и y является началом цепочки b then
      t ← priority(a)
      добавить b в конец a
      priority(a) ← min(t, priority(b), P++)
   else priority(b) ← min(priority(b), P++)
```

Рис. 8.17 Построение горячих путей

Алгоритм игнорирует петли вида, (x, x), потому что они не никак не влияют на решения о размещении.

Затем алгоритм перебирает ребра ГПУ и строит цепочки для моделирования горячих путей. Он рассматривает ребра в порядке убывания частоты выполнения. Для ребра (x, y) алгоритм объединяет цепочки, содержащие x и y, тогда и только тогда, когда x – последняя вершина в своей цепочке, а y – первая вершина в своей. Если хотя бы одно условие не выполнено, то цепочки, содержащие x и y, остаются без изменения.

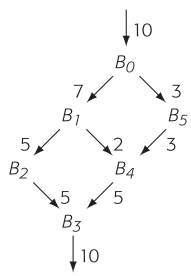
Если алгоритм объединяет цепочки, содержащие x и y, то он должен назначить новой цепочке подходящий приоритет. Этот приоритет вычисляется как минимум из приоритетов цепочек, содержащих x и y. Если x и y – вырожденные цепочки с высоким начальным приоритетом, то алгоритм делает приоритет новой цепочки равным числу уже рассмотренных ранее

объединений, которое мы обозначим P. Смысл такого выбора в том, чтобы поместить новую цепочку после цепочек, построенных из ребер с большей частотой, и до цепочек, построенных из ребер с меньшей частотой.

Прямой переход
Прямым называется переход, для которого конечный адрес больше начального. В некоторых ISA прямые переходы исполняются быстрее обратных.

Алгоритм останавливается, когда будут рассмотрены все ребра. Он порождает множество цепочек, моделирующих горячие пути в ГПУ. Каждая вершина принадлежит ровно одной цепочке. Ребра, принадлежащие цепочкам, выполняются чаще, чем ребра, соединяющие одну цепочку с другой. Приоритеты цепочек аппроксимируют порядок, который должен был бы максимизировать число исполняемых прямых переходов.

Для иллюстрации работы алгоритма рассмотрим его поведение применительно к примеру ГПУ из предыдущего раздела, изображенному на рис. 8.18(a). На рисунке (b) показано продвижение алгоритма в этом примере. Приоритеты цепочек представлены индексами. Величина E равна $|edges|$ (число ребер).



(a) Пример ГПУ

Ребро	Множество цепочек	P
—	$\langle B_0\rangle_E,\ \langle B_1\rangle_E,\ \langle B_2\rangle_E,\ \langle B_3\rangle_E,\ \langle B_5\rangle_E,\ \langle B_4\rangle_E$	0
(B_0,B_1)	$\langle B_0,B_1\rangle_0,\ \langle B_2\rangle_E,\ \langle B_3\rangle_E,\ \langle B_5\rangle_E,\ \langle B_4\rangle_E$	1
(B_2,B_3)	$\langle B_0,B_1\rangle_0,\ \langle B_2,B_3\rangle_1,\ \langle B_5\rangle_E,\ \langle B_4\rangle_E$	2
(B_4,B_3)	$\langle B_0,B_1\rangle_0,\ \langle B_2,B_3\rangle_1,\ \langle B_5\rangle_E,\ \langle B_4\rangle_E$	3
(B_1,B_2)	$\langle B_0,B_1,B_2,B_3\rangle_0,\ \langle B_5\rangle_E,\ \langle B_4\rangle_E$	4
(B_0,B_5)	$\langle B_0,B_1,B_2,B_3\rangle_0,\ \langle B_5\rangle_4,\ \langle B_4\rangle_E$	5
(B_5,B_4)	$\langle B_0,B_1,B_2,B_3\rangle_0,\ \langle B_5,B_4\rangle_4$	6
(B_1,B_4)	$\langle B_0,B_1,B_2,B_3\rangle_0,\ \langle B_5,B_4\rangle_4$	7

(b) Шаги алгоритма построения горячих путей для примера

Рис. 8.18 Пример применения алгоритма построения горячих путей

Если две цепочки имеют равные приоритеты, то решение о том, какую выбрать первой, приводит к различным размещениям в памяти. Например, если алгоритм поставит (B_4, B_3) раньше (B_2, B_3), то будут построены цепочки $\langle B_0, B_1, B_2\rangle_0$ и $\langle B_5, B_4, B_3\rangle_1$. Разные цепочки могут порождать разные размещения. Алгоритм в любом случае дает хорошие результаты, даже при неоптимальном упорядочении ребер равного веса.

Выполнение размещения кода

Множество цепочек, порождаемых алгоритмом поиска горячих путей, индуцирует частичный порядок на множестве простых блоков. Чтобы сгенерировать исполняемый код, компилятор должен выстроить все блоки в фиксированном линейном порядке. На рис. 8.19 показан алгоритм, который

вычисляет линейное размещение по множеству цепочек. В нем используются две простые эвристики: (1) размещать блоки цепочки по порядку, так чтобы неявные ветви реализовывали ребра, входящие в цепочки, и (2) размещать цепочки в порядке возрастания приоритетов.

```
t ← цепочка, началом которой является входная вершина ГПУ
WorkList ← { (t, priority(t)) }
while (Worklist ≠ ∅) do
  удалить цепочку с с наименьшим приоритетом из WorkList
  for каждого блока x в с в порядке следования в цепочке do
    поместить x в конец исполняемого кода
  for каждого блока x в с do
    for каждого ребра ⟨x, y⟩, где y еще не размещена do
      t ← цепочка, содержащая y
      if (t, priority(t)) ∉ WorkList then
        WorkList ← WorkList ∪ { (t, priority(t)) }
```

Рис. 8.19 Алгоритм размещения кода

Рабочий список *Worklist* может быть реализован в виде разреженного множества (см. приложение B.2.3).

Алгоритм представляет цепочку как пару (l, p), где l – список блоков, а p – приоритет цепочки. Каждая цепочка должна входить в рабочий список ровно один раз. В следующей таблице показано поведение алгоритма на множестве цепочек, выведенном на рис. 8.18(b):

Шаг	*WorkList*	Размещение кода
–	$\langle B_0, B_1, B_2, B_3\rangle_0$	
1	$\langle B_5, B_4\rangle_4$	B_0, B_1, B_2, B_3
2	∅	$B_0, B_1, B_2, B_3, B_5, B_4$

В первой строке показано начальное состояние; рабочий список начинается цепочкой, содержащей B_0. Алгоритм удаляет эту цепочку из рабочего списка и размещает все ее блоки по порядку. Затем он обрабатывает ребра, исходящие из уже размещенных блоков. При этом цепочка $\langle B_5, B_4\rangle$ добавляется в рабочий список. На второй итерации $\langle B_5, B_4\rangle$ удаляется из рабочего списка, а блоки B_5 и B_4 добавляются в конец размещения. Алгоритм не находит ребер, ведущих в еще не размещенные блоки, поэтому останавливается.

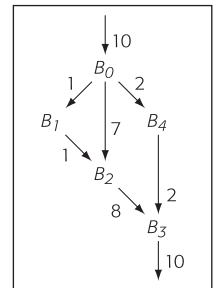
Если бы вместо этого алгоритм породил цепочки $\langle B_0, B_1, B_2\rangle_0$ и $\langle B_5, B_4, B_3\rangle_1$, то конечное размещение было бы другим.

Шаг	*WorkList*	Размещение кода
–	$\langle B_0, B_1, B_2\rangle_0$	
1	$\langle B_5, B_4, B_3\rangle_1$	B_0, B_1, B_2
2		$B_0, B_1, B_2, B_5, B_4, B_3$

В предположении, что оценки частот выполнения правильны, нет причин предпочесть одно размещение другому.

Последний пример

В качестве последнего примера рассмотрим ГПУ во врезке. Первый шаг алгоритма размещения – применить алгоритм поиска горячих путей. Результат получается таким:



Последний пример

Ребро	Множество цепочек	P
–	$\langle B_0\rangle_E, \langle B_1\rangle_E, \langle B_2\rangle_E, \langle B_3\rangle_E, \langle B_4\rangle_E$	0
$\langle B_2, B_3\rangle$	$\langle B_0\rangle_E, \langle B_1\rangle_E, \langle B_2, B_3\rangle_0, \langle B_4\rangle_E$	1
$\langle B_0, B_2\rangle$	$\langle B_0, B_2, B_3\rangle_0, \langle B_1\rangle_E, \langle B_4\rangle_E$	2
$\langle B_0, B_4\rangle$	$\langle B_0, B_2, B_3\rangle_0, \langle B_1\rangle_E, \langle B_4\rangle_2$	3
$\langle B_4, B_3\rangle$	$\langle B_0, B_2, B_3\rangle_0, \langle B_1\rangle_E, \langle B_4\rangle_2$	4
$\langle B_0, B_1\rangle$	$\langle B_0, B_2, B_3\rangle_0, \langle B_1\rangle_4, \langle B_4\rangle_2$	5
$\langle B_1, B_2\rangle$	$\langle B_0, B_2, B_3\rangle_0, \langle B_1\rangle_4, \langle B_4\rangle_2$	6

Алгоритм останавливается, построив одну цепочку с несколькими вершинами и две вырожденные. Он улавливает тот факт, что на $\langle B_0, B_2, B_3\rangle$ приходится больше всего выполнений и что B_4 и B_1 выполняются редко.

Алгоритм размещения сначала размещает $\langle B_0, B_2, B_3\rangle$. В процессе обработки ребер, исходящих из этих вершин, он добавляет оба вырожденных блока в рабочий список. На следующих двух итерациях сначала удаляется и размещается B_4, а затем B_1.

ИТОГИ РАЗДЕЛА

У оптимизаций, исследующих всю процедуру целиком, имеются возможности по улучшению кода, недоступные в меньших областях. Поскольку глобальная, или общепроцедурная, область включает циклические пути, глобальные оптимизации обычно требуют глобального анализа. Поэтому такие алгоритмы, как правило, работают в режиме, напоминающем офлайновый: они анализируют код, прежде чем преобразовать его.

В этом разделе описаны два разных вида анализа: глобальный анализ потока данных и профилирование во время выполнения. Анализ потока данных – это метод времени компиляции, который математически точно учитывает эффекты на всех возможных путях выполнения кода. С другой стороны, данные профиля отражают, что реально происходило при одном прогоне программы на одном наборе входных данных. Профилирование во время выполнения агрессивно: предполагается, что будущие прогоны будут иметь те же характеристики, что во время профилирования. Оба вида информации могут играть важную роль в процессе оптимизации.

ВОПРОСЫ

1. В некоторых ситуациях компилятору нужно знать, что переменная жива на *всех* путях, исходящих из блока, а не на каком-то *одном* пути. Перепишите уравнения LiveOut, так чтобы они вычисляли множество имен, которые используются раньше определения на любом пути из конца блока в выходную вершину ГПУ, n_f.

2. Чтобы собрать точные данные о реберных счетчиках, компилятор может оснастить каждое ребро ГПУ профилируемой процедуры. Изобретательная реализация может оснастить некоторое подмножество ребер и вывести счетчики для остальных. Придумайте схему, которая выводит данные о реберных счетчиках, не оснащая каждую ветвь. На какие принципы опирается ваша схема?

8.7. Межпроцедурная оптимизация

Как обсуждалось в главе 6, вызовы процедур образуют границы программной системы. У разделения программы на несколько процедур имеются плюсы и минусы, влияющие на способность компилятора генерировать эффективный код. Плюс – то, что таким образом ограничивается объем кода, рассматриваемого компилятором в каждый момент времени. Поэтому структуры данных этапа компиляции малы, а стоимость алгоритмов, выполняемых во время компиляции, ограничена вследствие ограниченного размера задачи.

Минусом же является то, что из-за вызовов процедур уменьшается точность информации, собираемой на этапе компиляции. Например, рассмотрим вызов `fie` из `fee`, при котором переменная *x* передается по ссылке. Даже если компилятор знает, что перед вызовом *x* имеет значение 15, воспользоваться этим фактом после вызова он не вправе, не будучи уверен, что вызов не может изменить *x*. Чтобы использовать значение *x* после вызова, компилятор должен доказать, что формальный параметр, соответствующий *x*, не изменяется ни процедурой `fie`, ни любой другой, вызываемой из `fie` прямо или косвенно.

Вызовы процедур также влекут прямые затраты на предварительную и заключительную последовательности, пролог и эпилог (см. раздел 6.5). Эти последовательности кода состоят из нескольких операций, и на выполнение каждой уходит какое-то время. Для передач управления между этими последовательностями требуются команды безусловного перехода. Но для любого конкретного вызова компилятор может адаптировать либо эти последовательности, либо тело вызываемой стороны к фактическому контексту и улучшить производительность.

Межпроцедурный анализ потока данных обсуждается в разделах 9.2.4 и 9.4.

Процедуры и вызовы процедур привносят неэффективности, с которыми внутрипроцедурная оптимизация справиться не может. Чтобы отчасти решить проблему, компилятор может анализировать и преобразовывать несколько процедур совместно, применяя методы межпроцедурного анализа и оптимизации. Эти методы равно важны в Algol-подобных и объектно-ориентированных языках.

В этом разделе представлены две межпроцедурные оптимизации: встраивание вызовов процедур и размещение процедуры в памяти с целью улучшить локальность кода. Поскольку для общепрограммной оптимизации необходимо, чтобы у компилятора был доступ к коду, подвергаемому анализу и преобразованию, решение применить общепрограммную оптимизацию влияет на всю структуру компилятора. Поэтому в последнем подразделе мы обсудим структурные проблемы, возникающие в системе, поддерживающей межпроцедурный анализ и оптимизацию.

8.7.1. Встраивание

В главах 6 и 7 мы видели, что код связи процедуры содержит много операций. Он выделяет память для записи активации, вычисляет все фактические па-

раметры, сохраняет состояние вызывающей стороны, создает окружение вызываемой стороны, передает управление от вызывающей стороны вызываемой и обратно, восстанавливает состояние вызывающей стороны и, если необходимо, возвращает значения от вызываемой стороны вызывающей. Все эти действия – часть накладных расходов, связанных с использованием языка высокого уровня; они необходимы для поддержки абстракций языка программирования, но при вычислении результатов без них иногда можно обойтись. Оптимизирующие компиляторы стараются уменьшить величину таких накладных расходов.

Встраивание
преобразование, которое заменяет место вызова копией тела вызываемой стороны, модифицированной с учетом переданных параметров.

В некоторых случаях компилятор может улучшить эффективность конечного кода, заменив место вызова копией тела вызываемой стороны, модифицированной с учетом контекста в месте вызова. Это преобразование, называемое *встраиванием*, позволяет компилятору исключить значительную часть кода, диктуемого соглашением о связях, и приспособить новую копию тела вызываемой стороны к контексту вызывающей. Поскольку встраивание перемещает код из одной процедуры в другую и изменяет граф вызовов программы, его смело можно называть межпроцедурным преобразованием.

Реализация встраивания естественно разделяется на две подзадачи: собственно преобразование и процедура решения о том, в каких местах вызова производить встраивание. Само преобразование сравнительно простое. Процедура решения – сложнее. От выбора в каждом случае напрямую зависит эффективность преобразования.

Преобразование

Чтобы выполнить встраивание, компилятор заменяет место вызова телом вызываемой стороны, внося изменения, необходимые для моделирования эффектов передачи параметров. На рис. 8.20(a) показаны две процедуры, `fee` и `fie`, каждая из которых вызывает третью процедуру, `foe`. На рис. 8.20(b) показан поток управления после встраивания вызова `fie` в `foe`. Компилятор создал копию `foe` и поместил ее внутрь `fie`, включил предварительную последовательность `fie` прямо в пролог ее копии в `foe` и точно так же включил эпилог в заключительную последовательность. Некоторые результирующие блоки можно объединить, подготовив тем самым почву для последующей оптимизации.

Конечно, компилятор должен использовать ПП, способное представить встраивание процедуры. Некоторые конструкции исходного языка могут создавать произвольные и необычные управляющие конструкции в результирующем коде. Например, вызывающая сторона с несколькими возвратами из разных точек может порождать граф потока управления (ГПУ), который трудно представить в ПП, близком к исходному коду.

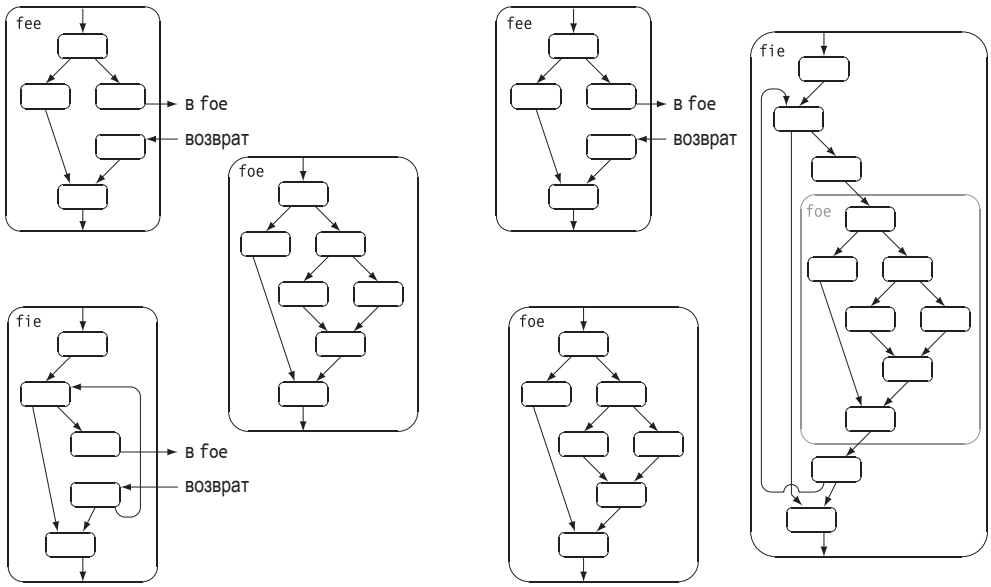


Рис. 8.20 Пример встраивания

Если компилятор применяет анализ потока данных, то рост пространства имен выливается в увеличение размеров множеств, используемых для решения задачи о потоке данных. В типичной задаче такого рода используется несколько множеств на каждую вершину ГПУ.

Реализуя преобразование, автор компилятора должен обращать внимание на увеличение числа локальных переменных. При простой реализации на вызывающей стороне будет создано по одной новой локальной переменной для каждой локальной переменной на вызываемой стороне. Если компилятор встраивает несколько процедур или одну процедуру в несколько мест вызова, то локальное пространство имен может сильно вырасти. И хотя рост пространства имен на корректности программы не сказывается, время компиляции преобразованного кода может значительно возрасти.

Процедура решения

Решить, в какие места вызова производить встраивание, трудно. Встраивание может повысить производительность, но, к сожалению, возможно и обратное. Чтобы принять взвешенное решение, компилятор должен учитывать широкий спектр характеристик вызывающей стороны, вызываемой стороны и места вызова. Также компилятор должен понимать свои собственные сильные и слабые стороны.

Основные источники улучшений в результате встраивания – прямое исключение операций и повышение эффективности других оптимизаций. Первый эффект достигается, когда можно устранить части последовательностей связи; например, код сохранения и восстановления регистров можно исклю-

чить, отдав такие решения на откуп распределителю регистров. Знания о вызывающей стороне иногда помогают доказать, что какой-то код на вызываемой стороне мертвый или бесполезный. Более точные знания могут также выявить константные значения или избыточности.

Основной источник ухудшений в результате встраивания – снижение эффективности оптимизации результирующего кода. Встраивание вызываемой стороны может увеличить размер кода и пространства имен. Может также повыситься спрос на регистры в окрестности места вызова. Устранение кода сохранения и восстановления регистров изменяет задачу, которую видит распределитель регистров. На практике любое из этих изменений может снизить эффективность оптимизации.

Изменение архитектуры, например увеличение числа регистров, может привести к повышению стоимости вызова процедуры, что, в свою очередь, может повысить привлекательность встраивания.

В каждом месте вызова компилятор должен решить, имеет ли смысл встраивание. Добавим к этому, что решение, принятое в одном месте вызова, может повлиять на решения в других местах. Например, если *a* вызывает *b*, а та вызывает *c*, то решение о встраивании *c* в *b* изменяет как характеристики процедуры, которая могла бы быть встроена в *a*, так и граф вызовов в оптимизируемой программе. Кроме того, встраивание приводит к увеличению размера кода, и эту проблему следует рассматривать для программы в целом; возможно, автор компилятора хотел бы ограничить общий рост размера кода.

Процедура принятия решения о встраивании проверяет различные критерии в каждом месте вызова:

- *размер вызываемой стороны*. Если размер вызываемой стороны меньше размера кода, относящегося к последовательностям связи, то встраивание должно уменьшать как размер кода, так и число выполняемых операций. Эта ситуация возникает на удивление часто;
- *размер вызывающей стороны*. Компилятор может налагать ограничения на общий размер любой процедуры, чтобы воспрепятствовать увеличению времени компиляции и уменьшению эффективности оптимизации;
- *динамический счетчик вызовов*. Улучшение часто выполняемого вызова дает больший выигрыш, чем такое же улучшение для редко выполняемого вызова. На практике компиляторы пользуются либо данными профилирования, либо простыми оценками, например десятикратной глубиной вложенности цикла, содержащего место вызова;
- *константные фактические параметры*. Если некоторые фактические параметры имеют известные постоянные значения, то преобразованный код может встроить эти константы прямо в тело вызываемой стороны;
- *статический счетчик вызовов*. Часто компиляторы подсчитывают количество мест, из которых вызывается процедура. Любую процедуру, вызываемую из единственного места, можно встроить, не увеличивая размер кода;

- *счетчик параметров*. Число параметров может служить оценкой затрат на вызов процедуры, поскольку компилятор должен генерировать код для вычисления и сохранения каждого фактического параметра;
- *вызовы из процедуры*. Подсчет числа вызовов из процедуры – простой способ обнаружить листья графа вызовов. Листовые процедуры часто являются хорошими кандидатами на встраивание;
- *глубина вложенного цикла*. Места вызова в циклах исполняются чаще, чем места вызова вне циклов. Они также подрывают способность компилятора планировать цикл как целостную единицу (см. раздел 12.4);
- *доля времени выполнения*. Вычисление доли времени выполнения, проведенного в каждой процедуре, на основе данных профиля может отвратить компилятор от встраивания процедур, не оказывающих заметного влияния на производительность.

По ходу встраивания у компилятора может возникнуть необходимость в обновлении этих метрик. Например, встраивание *b* в *a* изменяет как *статический счетчик вызовов* для *b*, так и метрику *вызовы из процедуры* для *a*.

На практике компиляторы заранее вычисляют некоторые или все метрики, а затем применяют набор эвристик, чтобы определить, в каких местах вызова производить встраивание. На рис. 8.21 показана типичная эвристика. В ней используется несколько пороговых параметров с именами от t_0 до t_4. От конкретных значений этих параметров во многом зависит поведение эвристики; например, значение t_3 должно быть больше размера стандартных предварительной и заключительной последовательностей. Оптимальные настройки некоторых параметров почти всегда зависят от конкретной программы.

Встраивать в любое место вызова, удовлетворяющее одному из следующих условий:

(1) на вызываемую сторону приходится более t_0 процентов времени выполнения и
(a) вызываемая сторона не содержит других вызовов или
(b) статический счетчик вызовов равен 1 или
(c) вызываемой стороне передается больше t_1 константных параметров;

(2) на место вызова приходится более t_2 всех вызовов и
(a) размер вызываемой стороны меньше t_3 или
(b) после встраивания размер процедуры будет меньше t_4.

Рис. 8.21 Типичные эвристики для принятия решения о встраивании

8.7.2. Размещение процедуры в памяти

Метод глобального размещения кода, описанный в разделе 8.6.2, предполагал изменение порядка блоков в рамках одной процедуры. В общепрограммной постановке аналогичная задача – изменить порядок процедур в исполняемом файле.

Для данного графа вызовов в программе, аннотированного частотами выполнения каждого места вызова, требуется изменить порядок разме-

щения процедур с целью уменьшить размер рабочего множества в виртуальной памяти и ограничить потенциальные конфликты в кеше команд из-за вызовов.

Отображение виртуальных адресов в физические может усложнить задачу, но ассоциативность кеша должна сгладить эту проблему.

Принцип простой. Если процедура *p* вызывает процедуру *q* и компилятор размещает их рядом в памяти, то они не должны вызывать конфликтов в кеше команд, если только *size*(*p*) + *size*(*q*) не превышает размера кеша. Размещение рядом друг с другом также повышает вероятность того, что *p* и *q* окажутся на одной и той же странице виртуальной памяти, что приведет к уменьшению размера рабочего множества и количества страничных отказов. Таким образом, алгоритм пытается максимизировать число вызовов, при которых вызывающая и вызываемая стороны размещены в памяти рядом.

Напомним, что в графе вызовов программы имеется вершина для каждой процедуры и ребро (*x*, *y*) для каждого вызова *y* из *x*.

Для вычисления размещения алгоритм рассматривает граф вызовов программы как множество ограничений на относительное размещение процедур в исполняемом коде. Каждое ребро графа вызовов, (*p*, *q*), определяет желательное соседство в исполняемом коде. К сожалению, компилятор не может удовлетворить все эти пожелания. Например, если *p* вызывает *q*, *r* и *s*, то компилятор не может разместить все три рядом с *p*. В алгоритме применяется жадный приближенный метод нахождения хорошего размещения, но не делается попытки найти оптимальное размещение.

Между размещением процедур и глобальным размещением кода, обсуждавшимся в разделе 8.6.2, есть тонкие различия. Алгоритм глобального размещения улучшает код, стремясь к тому, чтобы горячие пути были реализованы неявными ветвями. Поэтому алгоритм построения цепочек на рис. 8.17 игнорирует те ребра ГПУ, которые не ведут из конца одной цепочки в начало другой. С другой стороны, когда алгоритм размещения процедур строит цепочки процедур, он может использовать соединяющие процедуры ребра, которые расположены в серединах цепочек, потому что его цель – просто разместить процедуры рядом друг с другом и таким образом уменьшить размеры рабочих множеств и интерференцию в кеше команд. Если *p* вызывает *q* и расстояние между *p* и *q* меньше размера кеша, то размещение достигло своих целей. Следовательно, у алгоритма размещения процедур в некотором смысле больше свободы, чем у алгоритма размещения блоков.

Алгоритм размещения процедур включает два этапа: анализ и преобразование. Этап анализа работает с графом вызовов программы. Он в цикле выбирает две вершины графа вызовов, соединенные ребром, и объединяет их. Порядок объединения определяется данными о частотах выполнения, измеренными или оцененными. От этого порядка зависит конечное размещение. Этап преобразования прямолинейный: нужно просто разместить процедуры в порядке, выбранном на этапе анализа.

При объединении ребер вес нового ребра делается равным сумме весов, из которых оно получено.

На рис. 8.22 показан жадный алгоритм, работающий на этапе анализа. Он строит размещение, рассматривая ребра графа вызовов в порядке убывания частоты выполнения. На первом шаге строится граф вызовов, его ребра аннотируются частотами, и все ребра, соединяющие две вершины, объединяются в одно. На последнем шаге инициализации строится приоритетная очередь ребер графа вызовов, упорядоченная по весам.

```
// Инициализация
построить мультиграф вызовов G
инициализировать приоритетную очередь Q   // упорядочить Q по убыванию

for каждого ребра (x,y) ∈ G do            // аннотировать ребра весами
  if (x = y) then                         // петли не интересны
    удалить (x,y) из G
  else
    weight((x,y)) ← оценка частоты выполнения для (x,y)

for каждой вершины x ∈ G do
  list(x) ← { x }                         // инициализировать списки размещения
  if существует несколько ребер из x в y then
    объединить их и их веса
  for каждого ребра (x,z) ∈ G do          // поместить каждое ребро в Q
    Enqueue(Q, (x,z), weight((x,z)))

// Итеративное уменьшение графа
while Q не пуста do
  (x,y) ← Dequeue(Q)                      //взять ребро с наибольшим приоритетом

  for каждого ребра (y,z) ∈ G do          // переместить начало из y в x
    ReSource( (y,z), x)

  for каждого ребра (z,y) ∈ G do          // переместить конец из y в x
    ReTarget((z,y), x)
  дописать list(y) в конец list(x)        // обновить список размещения
  удалить y и связанные с ней ребра из G  // очистить G
```

Рис. 8.22 Этап анализа в алгоритме размещения процедур

Во второй части алгоритма с каждой вершиной графа ассоциируется упорядоченный список процедур. Эти списки индуцируют линейный порядок на множестве именованных процедур. Когда алгоритм останавливается, списки будут определять полный порядок на множестве процедур в каждой вершине, которым можно будет воспользоваться для размещения процедур в исполняемом коде.

Для направления процесса в нужную сторону алгоритм использует веса ребер графа вызовов. Он в цикле выбирает из приоритетной очереди ребро с наибольшим весом, скажем (x, y), и объединяет его начало x и конец y. Затем он должен обновить граф вызовов, отразив это изменение.

1. Для каждого ребра (y, z) алгоритм вызывает функцию *ReSource*, которая заменяет (y, z) на (x, z) и обновляет приоритетную очередь. Если (x, z) уже существует, то *ReSource* объединяет счетчики выполнения (y, z) и существующего (x, z).
2. Для каждого ребра (z, y) алгоритм вызывает функцию *ReTarget*, которая заменяет (z, y) на (z, x) и обновляет приоритетную очередь. Если (z, x) уже существует, то *ReTarget* объединяет счетчики выполнения (z, y) и существующего (z, x).

Чтобы гарантировать размещение y после x, алгоритм добавляет $list(y)$ в конец $list(x)$. Наконец, он удаляет y и связанные с ней ребра из графа вызовов.

Алгоритм останавливается, когда приоритетная очередь оказывается пустой. В конечном графе имеется одна вершина для каждой компоненты связности исходного графа. Если все вершины достижимы из точки входа в программу, то конечный граф будет состоять из одной вершины. Если некоторые вершине недостижимы – либо потому, что не вызываются ни на каком пути выполнения программы, либо потому, что пути к ним замаскированы неоднозначными вызовами, – то конечный граф будет содержать несколько вершин. В любом случае компилятор и компоновщик могут использовать списки, ассоциированные с вершинами конечного графа, чтобы определить относительное размещение процедур.

Пример

Чтобы увидеть, как работает алгоритм размещения процедур, рассмотрим пример графа вызовов на панели 0 рис. 8.23. Ребро из P_5 в себя показано серым цветом, потому что в алгоритме оно используется только для изменения частот выполнения. Петля не может повлиять на размещение, потому что ее начало совпадает с концом.

На панели 0 показано состояние алгоритма перед началом итеративного уменьшения. С каждой вершиной ассоциирован тривиальный список, содержащий только ее собственное имя. В приоритетной очереди находятся все ребра, кроме петли, упорядоченные по частоте выполнения.

На панели 1 показано состояние алгоритма после первой итерации цикла while. Алгоритм объединил P_6 с P_5 и обновил список для P_5 и приоритетную очередь.

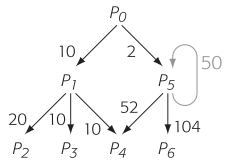
На панели 2 алгоритм объединил P_4 с P_5. Он присоединил ребро (P_1, P_4) к P_5 и изменил имя соответствующего ребра в очереди. Он также удалил P_4 из графа и обновил список для P_5.

Итерации продолжаются аналогично. На панели 4 показана ситуация, когда алгоритм объединил ребра. Объединив вершину P_5 с P_1, он присоединил (P_0, P_5) к P_1. Поскольку (P_0, P_1) уже существовало, он просто сложил их веса и обновил приоритетную очередь, удалив (P_0, P_5) и изменив вес ребра (P_0, P_1).

По завершении всех итераций граф схлопнулся в одну вершину, P_0. В этом примере построено размещение, начинающееся с входной вершины, но так произошло только из-за весов ребер, а не вследствие внутреннего устройства алгоритма.

Начальное состояние

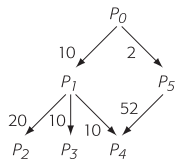
0.



	P_0	P_1	P_2	P_3	P_4	P_5	P_6
List(x):	$\{P_0\}$	$\{P_1\}$	$\{P_2\}$	$\{P_3\}$	$\{P_4\}$	$\{P_5\}$	$\{P_6\}$

Q: $\{(P_5,P_6)_{104}, (P_5,P_4)_{52}, (P_1,P_2)_{20}, (P_1,P_4)_{10}, (P_1,P_3)_{10}, (P_0,P_1)_{10}, (P_0,P_5)_2\}$

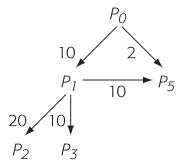
1.



	P_0	P_1	P_2	P_3	P_4	P_5
List(x):	$\{P_0\}$	$\{P_1\}$	$\{P_2\}$	$\{P_3\}$	$\{P_4\}$	$\{P_5,P_6\}$

Q: $\{(P_5,P_4)_{52}, (P_1,P_2)_{20}, (P_1,P_4)_{10}, (P_1,P_3)_{10}, (P_0,P_1)_{10}, (P_0,P_5)_2\}$

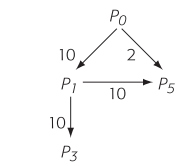
2.



	P_0	P_1	P_2	P_3	P_5
List(x):	$\{P_0\}$	$\{P_1\}$	$\{P_2\}$	$\{P_3\}$	$\{P_5,P_6,P_4\}$

Q: $\{(P_1,P_2)_{20}, (P_1,P_5)_{10}, (P_1,P_3)_{10}, (P_0,P_1)_{10}, (P_0,P_5)_2\}$

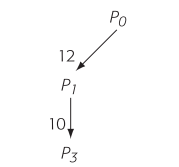
3.



	P_0	P_1	P_3	P_5
List(x):	$\{P_0\}$	$\{P_1,P_2\}$	$\{P_3\}$	$\{P_5,P_6,P_4\}$

Q: $\{(P_1,P_5)_{10}, (P_1,P_3)_{10}, (P_0,P_1)_{10}, (P_0,P_5)_2\}$

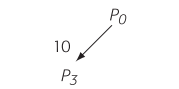
4.



	P_0	P_1	P_3
List(x):	$\{P_0\}$	$\{P_1,P_2,P_5,P_6,P_4\}$	$\{P_3\}$

Q: $\{(P_0,P_1)_{12}, (P_1,P_3)_{10}\}$

5.



	P_0	P_3
List(x):	$\{P_0,P_1,P_2,P_5,P_6,P_4\}$	$\{P_3\}$

Q: $\{(P_0,P_3)_{10}\}$

6.



	P_0
List(x):	$\{P_0,P_1,P_2,P_5,P_6,P_4,P_3\}$

Q: $\emptyset$

Рис. 8.23 Шаги алгоритма размещения процедур

8.7.3. Прагматические аспекты межпроцедурной оптимизации

Создание компилятора, выполняющего анализ и оптимизацию сразу двух или более процедур, фундаментально изменяет связь между компилятором и порождаемым им кодом. В традиционных компиляторах единицей

Единица компиляции
Часть программы, предъявляемая компилятору, часто называется *единицей компиляции*.

компиляции является одна процедура, один класс или один файл кода; результирующий код зависит только от содержимого единицы компиляции. Но коль скоро компилятор использует знания об одной процедуре для оптимизации другой, корректность результирующего кода зависит от состояния обеих процедур.

Рассмотрим влияние встраивания на действительность оптимизированного кода. Предположим, что компилятор встраивает `fie` в `fee`. Любое последующее изменение `fie` вынуждает перекомпилировать `fee` – эта зависимость является результатом решения об оптимизации, а не какой-то связи, существующей в исходном коде.

Если компилятор собирает и использует межпроцедурную информацию, то могут возникнуть похожие проблемы. Например, `fee` может вызывать `fie`, которая вызывает `foe`; предположим, что компилятор опирается на тот факт, что вызов `fie` не изменяет известное постоянное значение глобальной переменной *x*. Если впоследствии программист изменит `foe`, так что она будет модифицировать *x*, то результаты предыдущих компиляций `fee` и `fie` могут стать недействительными, потому что изменились предположения, на которых была основана оптимизация. Следовательно, изменение `foe` может повлечь за собой обязательную перекомпиляцию других процедур в программе.

Чтобы сделать межпроцедурную оптимизацию практичной, управлять перекомпиляцией и предоставить необходимый доступ к исходному коду, исследователи предложили ряд подходов: использовать более крупные единицы компиляции, интегрировать компилятор в среду разработки, выполнять компиляцию на этапе компоновки и поручить межпроцедурную оптимизацию JIT-компилятору.

Увеличение единиц компиляции

Самое простое решение практических проблем, связанных с межпроцедурной оптимизацией, – увеличить размер единиц компиляции. Если компилятор умеет выполнять анализ и оптимизацию только в пределах одной единицы компиляции, то проблема перекомпиляции вообще не возникает. Анализ и оптимизация ограничиваются только кодом, компилируемым вместе, а значит, не образуется никаких зависимостей между единицами компиляции, и доступ к информации о других единицах не нужен. Такой подход был принят в оптимизирующем компиляторе IBM PL/I; качество кода улучшалось, когда логически связанные процедуры помещались в один файл.

Разумеется, при таком подходе возможности межпроцедурной оптимизации ограничены. Ожидается, что программист будет создавать более крупные единицы компиляции и группировать процедуры, вызывающие друг друга. То и другое может натолкнуться на практические проблемы, когда над системой работает несколько программистов. Тем не менее с практической

точки зрения эта организация приклекательна, потому что требует лишь минимального отхода от традиционных моделей компиляции.

Интегрированные среды разработки

Если компилятор встроен в интегрированную среду разработки (IDE), то доступ к коду он может получать по мере необходимости через IDE. Среда может уведомлять компилятор об изменениях исходного кода, а компилятор уже сам определит, нужна ли перекомпиляция. Эта модель передает владение исходным и откомпилированным кодом от разработчика к IDE. Кооперация между IDE и компилятором служит залогом того, что будут предприняты необходимые действия, гарантирующие согласованную и корректную оптимизацию.

Оптимизация на этапе компоновки

Автор компилятора может поручить межпроцедурную оптимизацию компоновщику, который имеет доступ ко всему *статически* скомпонованному коду. Чтобы в полной мере воспользоваться преимуществами этого подхода, компоновщику, возможно, стоит выполнить и последующую глобальную оптимизацию. Поскольку результаты оптимизации на этапе компоновки записываются только в исполняемый файл, а этот файл уничтожается при следующей компиляции, проблема перекомпиляции при такой стратегии не возникает. Почти наверняка объем анализа и оптимизации при этом будет больше, чем при других подходах, но зато мы получаем простоту и очевидную корректность.

JIT-компиляция

Оптимизатор горячих трасс легко может построить трассы, пересекающие вызовы процедур (см. раздел 14.3). Оптимизаторы горячих методов могут включить встраивание, чтобы уменьшить стоимость динамической диспетчеризации (см. раздел 14.4).

Среда, в которой оптимизация производится на этапе выполнения, сталкивается с другими возможностями и ограничениями (см. главу 14). На этапе выполнения доступен весь код приложения, никаких проблем с доступом нет. JIT-компилятор может производить оптимизацию через границы процедур. В конце выполнения система отбрасывает JIT-компилированный код, поэтому вопрос о перекомпиляции может встать только для изменений, внесенных во время выполнения, – изменений, которые могут быть обработаны путем аккуратной деоптимизации (см. обсуждение на стр. 771).

Однако JIT-компиляторы работают в условиях временных ограничений, которые почти наверняка не дадут применить такие классические виды общепрограммного анализа, как вычисление сводной информации или межпроцедурное распространение констант. Системы оптимизации на этапе выполнения обычно управляются данными профилирования; они с большей

охотой находят и оптимизируют конкретные «горячие» вызовы, чем обнаруживают, что какая-то часть дерева вызовов свободна от побочных эффектов. Несмотря на эти ограничения, основанные на JIT-компиляции системы могут получить выигрыш от избирательного применения межпроцедурных оптимизаций. В этих системах действительно применяются межпроцедурные методы – и именно потому, что это целесообразно.

ИТОГИ РАЗДЕЛА

Анализ и оптимизация через границы процедур могут открыть новые возможности для улучшения кода. Примерами могут служить адаптация соглашений о вызове процедур (предварительная и заключительная последовательность, пролог, эпилог) к специфике места вызова или размещение процедур в памяти с целью улучшения локальности. Для распознавания и использования возможностей межпроцедурной оптимизации было предложено много методов; встраивание – один из самых хорошо известных и широко применяемых.

Компилятор, применяющий межпроцедурные методы, должен позаботиться о том, чтобы сгенерированные исполняемые файлы были основаны на непротиворечивом представлении программы в целом. Использование знаний об одной процедуре для модификации кода другой может привносить тонкие зависимости между кодом в процедурах, далеко отстоящих друг от друга; компилятор должен обнаруживать и учитывать такие зависимости. Для смягчения этих последствий было предложено и продемонстрировано на реальных системах несколько стратегий.

ВОПРОСЫ

1. Предположим, что процедура *a* вызывает *b* и *c*. Если компилятор встраивает вызов *b*, то какая возможна экономия памяти для кода и данных? Какая дополнительная экономия памяти для данных возможна при встраивании *c*?
2. Что произойдет при размещении процедуры, для которой оценки частоты выполнения на всех входящих ребрах нулевые? Куда алгоритм должен поместить такую процедуру? Влияет ли обработка такой процедуры на время выполнения? Может ли компилятор удалить ее как бесполезную?

8.8. Итоги и перспективы

В типичном компиляторе оптимизатор содержит набор методов, стремящихся улучшить производительность откомпилированного кода. Цель большинства оптимизаций – повысить скорость работы, но некоторые нацелены на другие характеристики программы, например размер кода или энергопотребление. В этой главе были представлены разнообразные методы, применимые к различным областям оптимизации – от простых блоков до всей программы.

Оптимизации повышают производительность за счет адаптации общих схем трансляции к конкретным деталям рассматриваемого кода. Преобразования, выполняемые оптимизатором, стремятся устранить накладные расходы, введенные для поддержки таких абстракций исходного языка, как структуры данных, управляющие конструкции и проверка ошибок. Они пытаются распознать специальные случаи, для которых имеются эффектив-

ные реализации, и переписать код с учетом открывающихся возможностей сэкономить. Они стараются согласовать потребность программы в ресурсах с ресурсами, доступными целевому процессору, в том числе функциональными блоками, емкостью и полосой пропускания каждого уровня иерархии памяти (регистров, кеша, буферов ассоциативной трансляции и оперативной памяти) и распараллеливанием на уровне команд.

Прежде чем применять преобразование, оптимизатор должен определить, безопасно ли предполагаемое переписывание кода, т. е. сохраняет ли оно оригинальную семантику кода. Обычно для доказательства безопасности оптимизатор должен проанализировать код. В этой главе мы рассмотрели несколько подходов к доказательству безопасности: от построения таблицы значений снизу вверх в алгоритме локальной нумерации значений до вычисления множеств LiveOut с целью обнаружения неинициализированных переменных.

Определив, что применять преобразование безопасно, оптимизатор должен решить, сможет ли переписывание улучшить код. Некоторые методы, например локальная нумерация значений, просто считают, что любое выполняемое ими переписывание целесообразно. В других методах, например встраивании, требуются сложные процедуры принятия решений, определяющие, когда преобразование может улучшить код.

В этой главе приведено введение в предмет оптимизации кода компилятором. В ней мы познакомились с терминологией и проблематикой оптимизации. В главе нет раздела «Дополнительные темы», но интересующийся читатель найдет дополнительный материал по статическому анализу в главе 9, а по оптимизирующим преобразованиям – в главе 10.

Замечания к этой главе

Вопросам оптимизации кода посвящена обширная литература, имеющая долгую историю. Более глубокое рассмотрение читатель найдет в специальных книгах [21, 277, 279]. Было бы хорошо и приятно, если бы дисциплина оптимизации кода развивалась логическим и дисциплинированным образом, начиная с локальных методов, переходя от них к регионам, затем к целым процедурам и, наконец, ко всей программе. Но в реальности развитие происходило не так гладко. Например, первый компилятор Fortran [28] умел выполнять локальную и глобальную оптимизацию – первую применительно к деревьям выражений, вторую для распределения регистров. Интерес к региональным методам, например оптимизации циклов [260], и межпроцедурным методам, например встраиванию, также прослеживается уже в ранних работах [17].

Локальная нумерация значений и ее обобщения на алгебраические упрощения и свертку констант обычно приписываются Балке, опубликовавшему в конце 1960-х годов работы [17, 94], хотя ясно, что Ершов получил ана-

логичные результаты в гораздо более ранней системе [150]. Флойд также упоминал потенциал локального устранения избыточности и применения коммутативности [160]. Обобщение на РПБ в алгоритме суперлокальной нумерации значений выглядит естественно и, без сомнения, вновь и вновь открывалось авторами многих компиляторов. Наше изложение следует работе Симпсона [59].

Алгоритм балансировки деревьев по высоте принадлежит Ханту и Маккинли [105]; в нем используется функция ранжирования, основанная на кодах Хаффмана, но можно без труда адаптировать его к другим метрикам. Классический алгоритм балансировки деревьев команд принадлежит Баеру и Бове [30]. На практике алгоритм балансировки деревьев лучше всего применять непосредственно перед планированием.

Разворачивание – простейшая оптимизация вложенного цикла. Оно имеет долгую историю в литературе [17]. Несколько авторов изучали вопрос о выборе коэффициента разворачивания [130, 337]. Использование разворачивания для устранения операций копирования регистров, как в вопросе 2 к разделу 8.5, заимствовано из работы Кеннеди [225]. Разворачивание может иметь тонкие и неожиданные эффекты [118].

Идеи, лежащие в основе анализа живучести, существуют с тех пор, как компиляторы стали автоматически выделять память для значений [250]. Бэтти первым определил анализ живучести во внутреннем техническом отчете IBM [16]. Лоури и Мэдлок обсуждают «активные» (busy) переменные [260] и использование этой информации для устранения мертвого кода и рассуждений об интерференции (см. главу 13). Постановка задачи о живучести как задачи о глобальном потоке данных появилась к 1971 году [14, 224]. Она играет ключевую роль во многих оптимизациях (см., например, разделы 9.3 и 13.4).

Алгоритмы размещения кода в глобальной и общепрограммной областях взяты из работы Петтиса и Хансена [293]. Последующие исследования этой проблемы были посвящены сбору более качественных данных профилирования и улучшению размещения [171, 194]. Более поздние работы включают рассмотрение выравнивания ветвей (branch alignment) [72, 369] и размещение кода в памяти [85, 100, 171].

Межпроцедурная оптимизация обсуждалась в литературе на протяжении десятилетий [17, 334]. Хотя встраивание как преобразование выполняется без особых хитростей, его целесообразность была предметом многих исследований [34, 108, 129, 309]. Холл сообщает об одном исследовании, в котором увеличение пространства имен привело к уменьшению эффективности оптимизации [108]. Все сценарии, упомянутые нами в разделе 8.7.3, изучались в реальных системах [114, 334, 354]. Межпроцедурный анализ имеет долгую и богатую историю [19, 35, 37, 334]. Анализ перекомпиляции является предметом углубленного изучения в работах Бэрка и Торкзон [70, 347]. Дополнительные ссылки на работы по межпроцедурному анализу см. в замечаниях к главе 9. Межпроцедурные аспекты JIT-компиляции более глубоко обсуждаются в главе 14.

Упражнения

К разделу 8.4

1. Примените алгоритм на рис. 8.4 к блокам B_0 и B_1.

```
t1 ← a + b
t2 ← t1 + c
t3 ← t2 + d
t4 ← b + a
t5 ← t3 + e
t6 ← t4 + f
t7 ← a + b
t8 ← t4 - t7
9 ← t8 * t6
```

Блок B_0

```
t1 ← a × b
t2 ← t1 × 2
t3 ← t2 × c
t4 ← 7 + t3
t5 ← t4 + d
t6 ← t5 + 3
t7 ← t4 + e
t8 ← t6 + f
t9 ← t1 + 6
```

Блок B_1

2. Спроектируйте алгоритм преобразования блока трехадресного кода в лес деревьев выражений. Требуется, чтобы каждое экспонированное значение было корнем собственного дерева. Можете предполагать, что множества LiveOut и LiveIn для блока уже известны.

Экспонированное значение определено на стр. 424.

3. Рассмотрите простой блок *b*, например B_0 или B_1 из упражнения 8.1. В нем имеется *n* операций, пронумерованных от 1 до *n*.

- **a.** Для имени *x* множество Uses(*x*) содержит индексы в *b* всех операций, в которых *x* используется в качестве операнда. Разработайте алгоритм вычисления множества Uses для каждого имени, встречающегося в блоке *b*.
- **b.** Примените свой алгоритм к блокам B_0 и B_1 из упражнения 8.1.
- **c.** Для ссылки на *x* в операции *i* блока *b* Def(*x*, *i*) определяется как индекс в *b* операции, которая определяет значение *x*, видимое в операции *i*. Разработайте алгоритм вычисления Def(*x*, *i*) для каждой ссылки на *x* в *b*. Если *x* экспонировано сверху в *i*, то Def(*x*, *i*) должно быть равно −1.
- **d.** Примените свой алгоритм к блокам B_0 и B_1 из упражнения 8.1.

4. Примените алгоритм балансировки деревьев по высоте из раздела 8.4.2 к блокам B_0 и B_1 из упражнения 8.1. Можете предполагать, что LiveOut(B_0) = {t_3, t_9} и LiveOut(B_1) = {t_7, t_8, t_9}. Имена от `a` до `f` в блоках экспонированы сверху.

К разделу 8.5

5. Для графа потока управления на рис. 8.24:

- **a.** Найдите расширенные простые блоки и перечислите их различные пути.
- **b.** Примените к каждому блоку локальную нумерацию значений (ЛНЗ).

c. Примените к РПБ суперлокальную нумерацию значений (СНЗ). Обратите внимание на улучшения, которые удалось найти в дополнение к ЛНЗ.

6. Рассмотрите следующее вычисление пятиточечного шаблона:

```
do 20 i = 2, n-1, 1
    t1 = A(i,1)
    t2 = A(i,2)
    do 10 j = 2, m-1, 1
        t3 = A(i,j+1)
        A(i,j) = 0.2 × (t1 + t2 + t3
                 + A(i-1,j) + A(i+1,j))
        t1 = t2
        t2 = t3
10     continue
20 continue
```

На каждой итерации цикла выполняется две операции копирования.

a. Разворачивание цикла может устранить операции копирования. Какой коэффициент разворачивания необходим для устранения всех операций копирования в этом цикле?

b. В общем случае, если цикл содержит несколько повторений операций копирования, то как можно вычислить коэффициент разворачивания, необходимый для исключения их всех?

К разделу 8.6

7. Live(p) для некоторой точки p – это множество имен, живых в p. LiveOut(b) – это просто множество Live в конце блока b.

a. Разработайте алгоритм, который принимает на входе блок b и его множество LiveOut и порождает на выходе множество Live для каждой операции в блоке.

b. Примените свой алгоритм к блокам B_0 и B_1 из упражнения 8.1. Можете предполагать, что LiveOut(B_0) = {t_3, t_9} и LiveOut(B_1) = {t_7, t_8, t_9}.

8. Вычислите множества LiveOut для каждого блока в графе потока управления на рис. 8.24.

9. На рис. 8.17 показан алгоритм нахождения горячих путей в ГПУ.

a. Придумайте альтернативное построение горячих путей, в котором учитывается разрешение неоднозначности для ребер с равными весами.

b. Постройте два примера, в которых ваш алгоритм дает лучшее размещение кода, чем алгоритм, приведенный в книге. Используйте алгоритм размещения кода на рис. 8.19 с цепочками, построенными вашим алгоритмом и алгоритмом в книге.

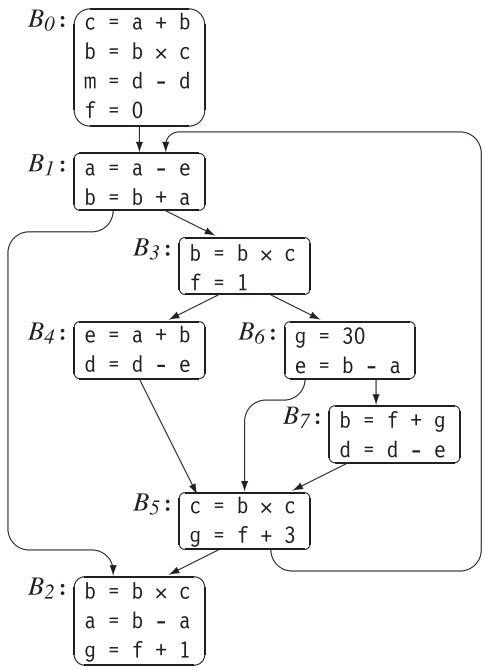


Рис. 8.24 Граф потока управления для вопроса 5

К разделу 8.7

10. Рассмотрите следующий фрагмент псевдокода. В нем показаны процедура `fee` и два ее места вызова.

```
static int A[1000,1000], B[1000], sum;
    ...
x = A[i,j] + y;
call fee(i,j,1000);
    ...
call fee(1,1,0);
    ...
fee(int row; int col; int ub)
    int i;
    sum = A[row,col];
    for (i=0; i<ub; i++)
        sum = sum + B[i];
```

a. Какого выигрыша вы ожидаете от встраивания `fee` в каждое место вызова? Оцените долю кода `fee`, которая останется после встраивания и последующей оптимизации.

b. На основе опыта, приобретенного в части a, набросайте высокоуровневый алгоритм оценки выигрыша от встраивания в конкретное место вызова. Ваш метод должен учитывать воздействие на обе стороны: вызывающую и вызываемую.

11. В предыдущей задаче особенности места вызова и окружающего его контекста определяли, до какой степени оптимизатор может улучшить встроенный код. На верхнем уровне опишите процедуру оценки улучшений, возможных в результате встраивания в конкретное место вызова. (Имея такое средство оценивания, компилятор мог бы осуществлять встраивание в места вызова с максимальной оценкой выигрыша и останавливаться по достижении некоторого порогового размера процедуры или программы в целом.)

12. Когда алгоритм на рис. 8.22 рассматривает ребро (p, q), он всегда помещает p раньше q.

- **a.** Модифицируйте алгоритм, так чтобы он рассматривал размещение конца ребра раньше его начала.
- **b.** Приведите пример, когда такой алгоритм размещает две процедуры ближе друг к другу, чем оригинальный. Можете предполагать, что размеры всех процедур одинаковы.

13. В примере на рис. 8.23 есть три ребра с одинаковыми весами: (P_0, P_1), (P_1, P_3) и (P_1, P_4). Покажите, как работал бы алгоритм, если бы выбирал эти ребра в другом порядке.

Глава 9

Анализ потока данных

АННОТАЦИЯ

Компиляторы анализируют программу в форме ПП, чтобы найти возможности улучшения кода и доказать безопасность и целесообразность преобразований, которые могли бы улучшить код. Анализ потока данных – классическая техника анализа программ на этапе выполнения. Он позволяет компилятору рассуждать о потоке значений в программе на этапе выполнения.

В этой главе рассматривается итеративный анализ потока данных, основанный на простом алгоритме с неподвижной точкой. Отправляясь от базового анализа потока данных, мы построим форму с единственным статическим присваиванием (SSA), проиллюстрируем ее применение и познакомимся с основами межпроцедурного анализа.

КЛЮЧЕВЫЕ СЛОВА

Анализ потока данных, доминирование, форма с единственным статическим присваиванием, распространение констант

9.1. Введение

В главе 8 мы видели, что оптимизация – это процесс анализа и преобразования программы с целью улучшить ее поведение во время выполнения. Прежде чем улучшить код, компилятор должен найти в программе точки, в которых изменение кода с большой вероятностью приведет к улучшению, *и* доказать, что изменение кода в этой точке безопасно. Обе задачи требуют более глубокого понимания кода, чем обычно дает фаза анализа. Чтобы собрать информацию, необходимую для выявления возможностей оптимизации и обоснования этих оптимизаций, компиляторы применяют ту или иную форму статического анализа.

В общем случае статический анализ включает рассуждения на этапе компиляции о потоке значений на этапе выполнения. В этой главе рассматриваются методы, которыми компиляторы пользуются для анализа программ в целях оптимизации.

Концептуальная дорожная карта

Компиляторы пользуются статическим анализом, чтобы определить, в каких местах применение оптимизирующих преобразований безопасно и целесообразно. В главе 8 мы видели, что оптимизации применяются к разным областям, от локальных до межпроцедурных. В общем случае преобразованию необходима аналитическая информация, охватывающая по меньшей мере область преобразования, т. е. для локальной оптимизации нужна по меньшей мере локальная информация, а для общепроцедурной, или глобальной, оптимизации – глобальная информация.

Статический анализ обычно начинается с анализа потока управления; компилятор строит граф, представляющий поток управления в коде. Затем компилятор анализирует, как именно значения перемещаются по коду. Собранную информацию он использует, чтобы найти возможности для улучшения и доказать безопасность преобразований. Анализ потока данных был разработан для ответа на эти вопросы.

Форма с единственным статическим присваиванием (SSA) – это промежуточное представление, в котором результаты анализа потока управления и потока данных объединены в единую разреженную структуру данных. Она доказала свою полезность для анализа и преобразования и стала стандартным ПП, которое используется в исследовательских и рабочих компиляторах.

Краткий обзор

В главе 8 мы познакомились с предметом анализа и преобразования программ, рассмотрели локальные, региональные, глобальные и межпроцедурные методы. Алгоритм нумерации значений прост, хотя и позволяет достичь нетривиальных результатов; он находит избыточные выражения, упрощает код, основанный на алгебраических тождествах, и распространяет известные константные значения. С другой стороны, нахождение неинициализированной переменной концептуально просто, но от компилятора требуется проанализировать всю процедуру и протрассировать определения и случаи использования переменных.

Точка соединения
В ГПУ точкой соединения называется вершина, у которой несколько предшественников.

Разница в сложности этих двух задач обусловлена тем, с какими видами потоков управления они сталкиваются. Локальная и суперлокальная нумерация значений имеет дело с подмножествами графа потока управления (ГПУ), которые образуют деревья (см. разделы 8.4.1 и 8.5.1). Для анализа всей процедуры компилятор должен рассуждать о полном ГПУ, включая циклы и *точки соединения*, что усложняет анализ. В общем случае методы, обрабатывающие только ациклические подмножества ГПУ, приспособлены к онлайновым решениям, тогда как методы, имеющие дело с циклами в ГПУ, требуют офлайновых решений – анализ должен быть полностью завершен, прежде чем можно будет приступать к переписыванию кода.

Статический анализ
анализ, выполняемый на этапе компиляции или компоновки.

Динамический анализ
анализ, выполняемый на этапе выполнения, может быть JIT-компилятором или специальным самомодифицируемым кодом.

Статический анализ, или анализ времени выполнения, – это собрание методов, которые компиляторы используют для доказательства безопасности и целесообразности потенциального преобразования. Статический анализ на уровне отдельных блоков или деревьев блоков, как правило, не вызывает трудностей. Эта глава посвящена глобальному анализу, когда ГПУ может содержать циклы и точки соединения. В ней упоминается несколько проблем межпроцедурного анализа, когда требуется работать с графом вызовов всей программы или еще с каким-то родственным графом.

В простых случаях статический анализ может давать точные результаты – компилятор может точно знать, что произойдет при выполнении кода. Если компилятор может вывести точную информацию, то сможет определить, что код вычисляет известное постоянное значение, и заменить вычисление выражения или функции непосредственной загрузкой результата. С другой стороны, если код читает значение из внешнего источника, включает хоть какое-то управление потоком или встречает неоднозначные ссылки на память, например указатели, ссылки на массив или параметры, передаваемые по ссылке, то статический анализ значительно усложняется, а его результаты становятся менее точными.

Мы начнем эту главу с классических задач анализа потока данных. Мы сосредоточимся на итеративном алгоритме их решения, потому что он простой, надежный и его легко понять. В разделе 9.3 представлен алгоритм построения SSA-формы процедуры. Построение опирается на результаты анализа потока данных. В разделе «Дополнительные темы» рассматривается понятие приводимости графа потока, описывается структура данных, которая ведет к быстрой версии вычисления доминатора, и приводится введение в межпроцедурный анализ потока данных.

Несколько слов о времени

Компилятор анализирует программу, чтобы определить, можно ли безопасно применить к ней улучшающие преобразования. *Статический анализ* либо доказывает утверждения о потоках управления и значений во время выполнения, либо аппроксимирует эти утверждения. Однако производится анализ на этапе компиляции. В классическом AOT-компиляторе анализ имеет место до выполнения кода.

В некоторых системах применяется компиляция во время выполнения, обычно в контексте JIT-компилятора (см. главу 14). В этом случае анализ и преобразования производятся на этапе выполнения, поэтому стоимость оптимизации включается во время выполнения программы. Эти затраты приходится нести при каждом выполнении программы.

9.2. Итеративный анализ потока данных

Прямая задача
задача, в которой факты в вершине n вычисляются на основе фактов, известных для предшественников n в ГПУ.

Обратная задача
задача, в которой факты в вершине n вычисляются на основе фактов, известных для преемников n в ГПУ.

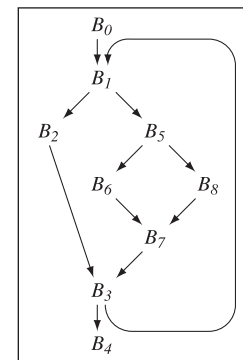
Компиляторы используют анализ потока данных – набор методов для рассуждения на этапе компиляции о потоке значений во время выполнения, – чтобы отыскать возможности для оптимизации и доказать безопасность конкретных преобразований. На примере анализа живучести в разделе 8.6.1 мы видели, что анализ потока данных принимает вид системы уравнений, определенных на множествах, ассоциированных с вершинами и ребрами графа, представляющего анализируемый код. Проблема анализа живучести формулируется как задача о глобальном потоке данных на графе потока управления (ГПУ) процедуры.

В этом разделе мы подробно изучим задачи о глобальном потоке данных и их решения. Нас будет интересовать один конкретный метод решения: итеративный алгоритм с неподвижной точкой. Его преимуществами являются простота, скорость и надежность. Сначала мы рассмотрим простую прямую задачу о доминаторах в графе потока. В качестве более сложного примера мы вернемся к вычислению множеств LiveOut, обратной задаче о потоке данных.

9.2.1. Доминирование

Доминирование
Для графа потока с входной вершиной b_0 говорят, что вершина b_i *доминирует* над вершиной b_j (записывается в виде $b_i \gg b_j$), если b_i принадлежит всем путям из b_0 в b_j. По определению, $b_i \gg b_i$.

Во многих методах оптимизации необходимо рассуждать о структурных свойствах анализируемой программы и ее ГПУ. Основным инструментом, которым компилятор пользуется для рассуждения о форме и структуре ГПУ, является понятие *доминирования*. Оно применяется, чтобы находить циклы и понимать, как код размещен в памяти. Доминирование играет важную роль в построении SSA-формы.



Пример ГПУ

Для вычисления информации о доминировании было предложено много алгоритмов. В этом разделе мы опишем простую задачу о потоке данных, в которой требуется аннотировать каждую вершину ГПУ b_i множеством $\mathrm{Dom}(b_i)$. Это множество содержит имена всех вершин, доминирующих над b_i.

Чтобы придать конкретики понятию доминирования, рассмотрим вершину B_6 ГПУ, показанного во врезке. Каждый путь из входной вершины B_0 в B_6 включает B_0, B_1, B_5 и B_6. Так что $\mathrm{Dom}(B_6)$ – это $\{B_0, B_1, B_5, B_6\}$. В таблице во врезке показаны все множества Dom для ГПУ.

n	DOM	IDOM
B_0	{0}	–
B_1	{0,1}	0
B_2	{0,1,2}	1
B_3	{0,1,3}	1
B_4	{0,1,3,4}	3
B_5	{0,1,5}	1
B_6	{0,1,5,6}	5
B_7	{0,1,5,7}	5
B_8	{0,1,5,8}	5

DOM и IDOM для примера ГПУ

Для любой вершины ГПУ n одна вершина $m \in \text{Dom}(n)$, $m \neq n$, будет ближе к n, чем любая другая вершина $x \in \text{Dom}(n)$, $x \neq n$. Эта вершина m называется *непосредственным доминатором* n и обозначается IDom(n). По определению, входная вершина графа потока не имеет непосредственного доминатора.

Следующие уравнения определяют множества Dom и лежат в основе метода их вычисления:

$$\text{Dom}(n) = \{n\} \cup \left(\bigcup_{m \in preds(n)} \text{Dom}(m) \right).$$

Для задания начальных значений компилятор полагает:

$\text{Dom}(n_0) = \{n_0\}$
$\text{Dom}(n) = N, \forall n \neq n_0,$

где N – множество всех вершин ГПУ. Для произвольного графа потока, т. е. ориентированного графа с одним входом и одним выходом, эти уравнения описывают множество Dom каждой вершины. В каждой точке соединения уравнения вычисляют пересечения множеств Dom вдоль каждого входящего пути. Поскольку они определяют Dom(n) как функцию предшественников n, обозначаемую *preds*(n), информация распространяется вперед вдоль ребер ГПУ. Таким образом, уравнения описывают прямую задачу о потоке данных.

Для решения этих уравнений компилятор может использовать тот же трехшаговый процесс, что для анализа живучести в разделе 8.6.1. Он должен (1) построить ГПУ, (2) собрать начальную информацию для каждого блока, (3) решить уравнения и найти множества Dom для каждого блока. Для Dom шаг 2 тривиален; для вычисления нужно знать только номера вершин.

На рис. 9.1 показан циклический итеративный решатель уравнений доминирования. Он рассматривает вершины в порядке их имен в ГПУ: B_0, B_1, B_2 и т. д. Алгоритм инициализирует множество Dom для каждой вершины, а затем в цикле пересчитывает их, пока они не перестанут изменяться.

```
Dom(0) ← { 0 }
for i ← 1 to |N| - 1 do
   Dom(i) ← N

changed ← true
while (changed) do
   changed ← false

   for i ← 1 to |N| - 1 do
      temp ← { i } ∪ (⋂_{j∈preds(i)} Dom(j))

      if temp ≠ Dom(i) then
         Dom(i) ← temp
         changed ← true
```

Рис. 9.1 Итеративный решатель задачи о доминировании

На рис. 9.2 показано, как изменяются значения в множествах Dom в процессе вычисления. Первый столбец содержит номер итерации; на итерации 0 показаны начальные значения. На итерации 1 вычисляются правильные множества Dom для всех вершин, в которые ведет единственный путь из B_0, но для вершин B_3, B_4 и B_7 множества Dom завышены. На итерации 2 меньшее множество Dom для B_7 исправляет множество для B_3, которое, в свою очередь, уменьшает Dom(B_4). Аналогично множество для B_8 исправляет множество для B_7. На итерации 3 алгоритм достиг неподвижной точки.

0	{0}	*N*	*N*	*N*	*N*	*N*	*N*	*N*	*N*
1	{0}	{0,1}	{0,1,2}	{0,1,2,3}	{0,1,2,3,4}	{0,1,5}	{0,1,5,6}	{0,1,5,6,7}	{0,1,5,8}
2	{0}	{0,1}	{0,1,2}	{0,1,3}	{0,1,3,4}	{0,1,5}	{0,1,5,6}	{0,1,5,7}	{0,1,5,8}
3	{0}	{0,1}	{0,1,2}	{0,1,3}	{0,1,3,4}	{0,1,5}	{0,1,5,6}	{0,1,5,7}	{0,1,5,8}

Рис. 9.2 Итерации алгоритма вычисления доминирования

К этой процедуре решения возникает три важных вопроса. Во-первых, всегда ли алгоритм останавливается? Он продолжает работать, пока множества Dom не перестанут изменяться, поэтому доказательство остановки не очевидно. Во-вторых, действительно ли он находит правильные множества Dom? Ответ критически важен, если мы собираемся использовать множества Dom для оптимизации. Наконец, насколько быстро работает решатель? Авторы компиляторов должны избегать алгоритмов, работающих медленно без особой на то необходимости.

Остановка

Итеративное вычисление множеств Dom останавливается, потому что множества, аппроксимирующие Dom, монотонно уменьшаются в процессе вычисления. Алгоритм инициализирует Dom(n_0) значением {0}, а множества Dom всех остальных вершин – значением *N*, множеством всех вершин. Множество Dom не может стать меньше {0} или больше *N*. Внимательное рассмотрение цикла while показывает, что никакое множество Dom, скажем Dom(n_i), не может расти при переходе к следующей итерации. Оно либо уменьшается, если уменьшается множество Dom одного из его предшественников, либо не изменяется.

Цикл while останавливается после итерации, на которой ни одно множество Dom не изменилось. Поскольку множества Dom могут только уменьшаться и их размер ограничен, цикл while должен рано или поздно остановиться. В момент остановки он нашел неподвижную точку этого конкретного вычисления Dom.

Корректность

Напомним определение доминирования. Вершина n_i доминирует над n_j, если любой путь из входной вершины n_0 в n_j содержит n_i. Доминирование – это свойство путей в ГПУ.

Dom(n_j) содержит i тогда и только тогда, когда $i \in$ Dom(n_k) для всех $k \in$ *preds*(j) или когда $i = j$. Алгоритм вычисляет Dom(n_j) как j плюс пересечение множеств Dom всех предшественников n_j. Как это локальное вычисление на отдельных ребрах связано со свойством доминирования, определенным на всех путях в ГПУ?

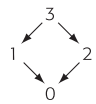
Оператор встречи
В теории анализа потока данных *оператор встречи* ($\wedge$) используется для комбинирования фактов в точке соединения в ГПУ.

В уравнениях Dom оператор встречи – это оператор пересечения множеств.

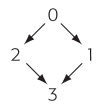
Множества Dom, вычисленные итеративным алгоритмом, образуют решение уравнений доминирования, являющееся неподвижной точкой. Теория итеративного анализа потока данных, изложение которой выходит за рамки книги, заверяет нас, что для этой системы уравнений неподвижная точка существует и единственна [221]. Эта «общепутевая» формулировка Dom описывает неподвижную точку системы уравнений, которая называется *встречей на всех путях*. Единственность гарантирует, что неподвижная точка, найденная итеративным алгоритмом, совпадает с этим решением.

Эффективность

Нумерация в обратном порядке
сопоставление вершинам графа меток, совпадающих с номером их посещения при обходе графа в обратном порядке.



Обратный порядок



Реверсивный обратный порядок

Компилятор может вычислить номера в порядке РОП, если вначале положит счетчик равным $|N| - 1$ и будет уменьшать его при посещении и пометке каждой вершины.

Поскольку решение уравнений Dom в виде неподвижной точки для конкретного ГПУ единственно, оно не зависит от порядка вычисления этих множеств решателем. Следовательно, автор компилятора вправе выбрать порядок вычислений, улучшающий время работы анализатора.

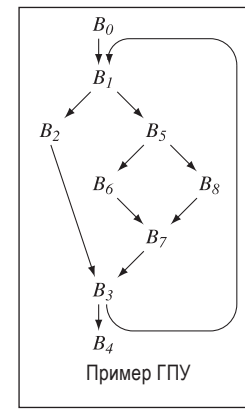
Обход графа в *реверсивном обратном порядке* (*РОП*) особенно эффективен для прямых задач о потоке данных. Если предположить, что номера вершин в обратном порядке изменяются от $|N| - 1$, то номер вершины в порядке РОП равен просто $|N| - 1$ минус номер этой вершины в обратном порядке. Здесь N – множество вершин графа.

Обход в порядке РОП сначала посещает всех возможных предшественников вершины в четко определенном порядке, а затем саму вершину. (В циклическом графе предшественник вершины может быть также ее

преемником.) У обратного порядка свойство противоположно: перед тем как посетить вершину n, посещаются все ее возможные преемники. У большинства интересных графов имеется несколько РОП-нумераций вершин; с точки зрения итеративного алгоритма, все они эквивалентны.

Для решения прямой задачи о потоке данных, такой как вычисление Dom, итеративный алгоритм должен использовать порядок РОП, вычисленный для ГПУ. Для обратной задачи, например вычисления множества Live-Out, алгоритм должен использовать РОП, вычисленный для *обращенного* ГПУ, т. е. ГПУ, у которого направление всех ребер изменено на противоположное. (Возможно, компилятору понадобится добавить единственную выходную вершину, чтобы у обращенного ГПУ была единственная входная вершина.)

Чтобы увидеть влияние упорядочения, рассмотрим, как РОП-проход влияет на наш пример вычисления Dom. Ниже и во врезке приведена одна из РОП-нумераций вершин нашего ГПУ:

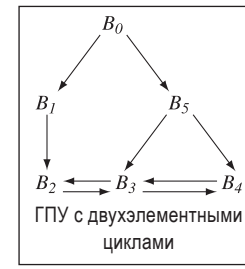


Пример ГПУ

	B_0	B_1	B_2	B_3	B_4	B_5	B_6	B_7	B_8
РОП(n)	0	1	6	7	8	2	4	5	3

Посещение вершин в этом порядке порождает последовательность итераций и значений, показанную на рис. 9.3. Обходя граф в порядке РОП, алгоритм точно вычисляет множества Dom для этого графа на первой итерации и останавливается после второй итерации. Порядок РОП позволяет алгоритму остановиться после двух проходов по графу вместо трех. Заметим, однако, что алгоритм не всегда точно вычисляет множества Dom на первом проходе, как показывает следующий пример.

В качестве второго примера рассмотрим другой ГПУ, показанный во врезке. В нем имеется два цикла с несколькими вершинами: (B_2, B_3) и (B_3, B_4). Важно, что (B_2, B_3) включает вершины из путей (B_0, B_1, B_2) и (B_0, B_5, B_3), а (B_3, B_4) – вершины из путей (B_0, B_5, B_3) и (B_0, B_5, B_4). Из-за этого свойства граф оказывается неприводимым, что затрудняет анализ потока данных с помощью некоторых алгоритмов (см. обсуждение приводимости в разделе 9.5.1).



ГПУ с двухэлементными циклами

	Dom(n)								
	B_0	B_1	B_2	B_3	B_4	B_5	B_6	B_7	B_8
0	{0}	N	N	N	N	N	N	N	N
1	{0}	{0,1}	{0,1,2}	{0,1,3}	{0,1,3,4}	{0,1,5}	{0,1,5,6}	{0,1,5,7}	{0,1,5,8}
2	{0}	{0,1}	{0,1,2}	{0,1,3}	{0,1,3,4}	{0,1,5}	{0,1,5,6}	{0,1,5,7}	{0,1,5,8}

Рис. 9.3 Итерации при вычислении доминирования в реверсивном обратном порядке

Чтобы применить итеративный алгоритм, нам нужна РОП-нумерация. Одна такая нумерация для этого ГПУ имеет вид:

	B_0	B_1	B_2	B_3	B_4	B_5
РОП(n)	0	2	3	4	5	1

При работе в этом порядке алгоритм выполняет следующие итерации:

	Dom(n)					
	B_0	B_1	B_2	B_3	B_4	B_5
0	{0}	N	N	N	N	N
1	{0}	{0,1}	{0,1,2}	{0,3}	{0,4}	{0,5}
2	{0}	{0,1}	{0,2}	{0,3}	{0,4}	{0,5}
3	{0}	{0,1}	{0,2}	{0,3}	{0,4}	{0,5}

Для вычисления множеств Dom этому алгоритму нужно две итерации. На последней итерации он понимает, что достиг неподвижной точки.

Для вычисления доминирования нужна только структура графа. Поведение кода в любом блоке ГПУ игнорируется. Поэтому саму задачу можно рассматривать как форму анализа потока управления. В большинстве задач о потоке данных имеют место рассуждения о поведении кода и о потоке данных между итерациями. В качестве примера такого рода вычисления мы вернемся к анализу живучести переменных.

9.2.2. Анализ живучести переменных

В разделе 8.6.1 мы использовали результаты анализа живучести для нахождения неинициализированных переменных. Компиляторы пользуются информацией о живучести для многих других целей, например распределения регистров и построения некоторых вариантов SSA-формы. Мы сформулировали задачу анализа живучести как задачу о глобальном потоке данных с помощью уравнения

$$\text{LiveOut}(n) = \bigcup_{m \in succ(n)} \left(\begin{array}{l} \text{UEVar}(m) \cup \\ (\text{LiveOut}(m) \cap \overline{\text{VarKill}(m)}) \end{array} \right),$$

где $succ(n)$ обозначает множество преемников n в ГПУ. В процессе анализа необходимо инициализировать $\text{LiveOut}(n) = \emptyset, \forall n$.

Сравнение уравнений для LiveOut и Dom вскрывает различия между задачами:

- LiveOut – обратная задача о потоке данных; LiveOut(n) – функция от информации, известной при входе в каждый из преемников n в ГПУ. Напротив, Dom – прямая задача о потоке данных;

- LiveOut ищет будущие использования на *каком-нибудь пути* в ГПУ; таким образом, информация о нескольких путях комбинируется с помощью оператора объединения. Dom же ищет предшественников, принадлежащих *всем путям* из входной вершины; следовательно, информация о нескольких путях комбинируется с помощью оператора пересечения;
- LiveOut рассуждает об эффектах операции. Множества UEVar(n) и VarKill(n) кодируют эффекты выполнения блока, ассоциированного с n. Напротив, в уравнениях Dom применяются только имена вершин. LiveOut использует больше информации и потребляет больше памяти.

Названия множеств в уравнениях потока данных

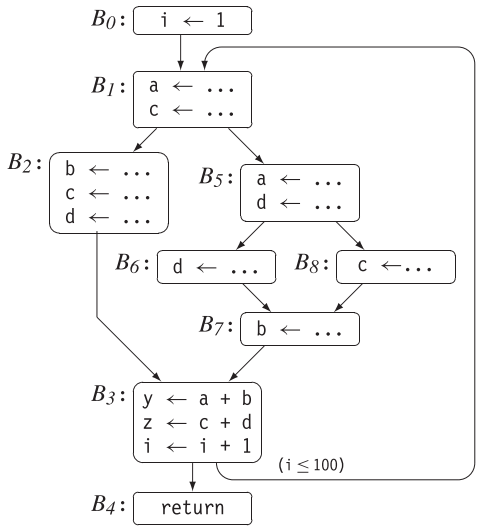
При записи уравнений потока данных для классических задач мы по-другому назвали множества, содержащие локальную информацию. В оригинальных статьях использовались интуитивно более понятные названия. К сожалению, эти названия конфликтуют друг с другом в разных задачах. Например, в задачах о *доступных выражениях*, *живых переменных*, *достигающих определениях* и *предсказуемых выражениях* используется понятие *уничтоженного* (kill) множества. Но эти четыре задачи определены над тремя разными областями: выражения (AvailOut и AntOut), точки определения (Reaches) и переменные (LiveOut). Поэтому использование одного названия множества, например Kill или Killed, может создать путаницу.

В принятых нами названиях закодирована как область определения, так и указание на семантику множества. Так, VarKill(n) – это множество переменных, уничтоженных в блоке n, тогда как ExprKill(n) – множество выражений, уничтоженных в том же блоке. Аналогично UEVar(n) – множество экспонированных сверху переменных в n, а UEExpr(n) – множество экспонированных сверху выражений. И хотя эти имена несколько громоздки, они позволяют провести явное различие между понятием уничтожения в задаче о *доступных выражениях* (ExprKill) и в задаче о *достигающем определении* (DefKill).

Несмотря на различия, процесс решения уравнений LiveOut такой же, как уравнений Dom. Компилятор должен: (1) построить ГПУ; (2) вычислить начальные значения множеств (см. рис. 8.15(a) на стр. 440) и (3) применить итеративный алгоритм (см. рис. 8.15(b)). Эти шаги аналогичны шагам решения уравнений Dom.

Чтобы разобраться в проблемах, которые могут возникать при решении уравнений LiveOut, рассмотрим код на рис. 9.4(a). В нем конкретизирован пример ГПУ, который мы использовали на протяжении всей этой главы. На рисунке (b) показаны множества UEVar и VarKill для каждого блока.

На рис. 9.5 показан ход работы итеративного решателя для примера на рис. 9.4(a) с применением того же РОП, который использовался при вычислении Dom. Хотя уравнения для LiveOut сложнее, чем для Dom, доказательства остановки, корректности и эффективности похожи.



(a) Код для примера анализа живучести

	UEVar	VarKill
B_0	∅	{i}
B_1	∅	{a,c}
B_2	∅	{b,c,d}
B_3	{a,b,c,d,i}	{y,z,i}
B_4	∅	∅
B_5	∅	{a,d}
B_6	∅	{d}
B_7	∅	{b}
B_8	∅	{c}

(b) Начальные множества для примера

Рис. 9.4 Примеры кода для анализа живучести

	LiveOut(n)								
	B_0	B_1	B_2	B_3	B_4	B_5	B_6	B_7	B_8
0	∅	∅	∅	∅	∅	∅	∅	∅	∅
1	∅	∅	{a,b,c,d,i}	∅	∅	∅	∅	{a,b,c,d,i}	∅
2	∅	{a,i}	{a,b,c,d,i}	{i}	∅	∅	{a,c,d,i}	{a,b,d,c,i}	{a,c,d,i}
3	{i}	{a,i}	{a,b,c,d,i}	{i}	∅	{a,c,d,i}	{a,c,d,i}	{a,b,c,d,i}	{a,c,d,i}
4	{i}	{a,c,i}	{a,b,c,d,i}	{i}	∅	{a,c,d,i}	{a,c,d,i}	{a,b,c,d,i}	{a,c,d,i}
5	{i}	{a,c,i}	{a,b,c,d,i}	{i}	∅	{a,c,d,i}	{a,c,d,i}	{a,b,c,d,i}	{a,c,d,i}

Рис. 9.5 Итерации решателя уравнений живучести в порядке (B_0, B_1, B_5, B_0, B_6, B_7, B_2, B_3, B_4)

Остановка

Напомним, что в Dom множества монотонно уменьшаются.

Итеративный анализ живучести останавливается, потому что множества монотонно увеличиваются, но имеется максимальный конечный размер. Всякий раз, как алгоритм решает уравнение LiveOut в некоторой вершине ГПУ, множество LiveOut либо не изменяется, либо становится больше. Множества LiveOut никогда не уменьшаются. Когда алгоритм достигает состояния, в котором ни одно множество LiveOut не изменяется, он останавливается. Найдена неподвижная точка.

В коде на рис. 9.4 V равно {a, b, c, d, i, y, z}.

Множества LiveOut конечны. Каждое множество LiveOut либо совпадает с V, множеством анализируемых имен, либо является собственным подмножеством V. В худшем случае множество LiveOut будет увеличиваться на одно

имя на каждой итерации; при таком поведении алгоритм остановится после $n \cdot |V|$ итераций, где n – число вершин в ГПУ.

Это свойство – комбинация монотонности и конечности множеств – гарантирует остановку. Часто его называют *свойством конечной убывающей цепочки*. В случае Dom множества монотонно уменьшаются, а их размер меньше или равен количеству вершин в ГПУ. В случае LiveOut множества монотонно увеличиваются, а их размер ограничен сверху количеством анализируемых имен. В обоих случаях остановка гарантирована.

Корректность

Итеративный анализ живучести корректен тогда и только тогда, когда он находит все переменные, удовлетворяющие условию живучести, в конце каждого блока. Напомним определение: переменная v называется *живой* в точке p, если существует путь из p в точку использования v, вдоль которого v не переопределяется. Таким образом, живучесть определяется в терминах путей в ГПУ. Должен существовать путь из p в точку использования v, на котором нет ни одного определения v. Такой путь мы называется v-чистым.

LiveOut(n) должно содержать v тогда и только тогда, когда v жива в конце блока n. Для построения LiveOut (n) итеративный решатель вычисляет вклад в LiveOut(n) каждого преемника n в ГПУ. Вклад преемника m в LiveOut(n) определяется правой частью уравнения LiveOut: UEVar(m) ∪ (LiveOut(m) ∩ $\overline{\text{VarKill}(m)}$). Решатель комбинирует вклады преемников с помощью оператора объединения, потому что $v \in$ LiveOut(n), если v жива на *каком-нибудь* пути, исходящем из n.

Как это локальное вычисление на отдельных ребрах связано с живучестью, определенной на всех путях? Множества LiveOut, вычисляемые решателем, – это решение уравнений живучести в форме неподвижной точки. И снова теория итеративного анализа потока данных утверждает, что уравнения живучести имеют единственное такое решение [221]. Единственность гарантирует, что все решения, являющиеся неподвижными точками, одинаковы, а стало быть, совпадают с решением встречи на всех путях, следующим из определения.

Эффективность

Для обратной задачи решатель должен использовать обход в порядке РОП обращенного ГПУ. В итеративном вычислении на рис. 9.5 использовался РОП самого ГПУ. Для нашего примера один из РОП обращенного ГПУ имеет вид

Возникает искушение подумать, что реверсивный обратный порядок на обращенном ГПУ эквивалентен реверсивному прямому порядку на самом ГПУ. В упражнении 3.b приведен контрпример.

	B_0	B_1	B_2	B_3	B_4	B_5	B_6	B_7	B_8
РОП(n)	8	7	6	1	0	5	4	2	3

Посещение вершин в этом порядке дает итерации, показанные на рис. 9.6. Теперь алгоритм останавливается после трех итераций вместо пяти, которые понадобились при обходе ГПУ в порядке РОП. Сравнивая эту таблицу с предыдущим вычислением, легко понять, почему так вышло. На первой итерации алгоритм правильно вычислил множества LiveOut для всех вершин, кроме B_3. Для B_3 понадобилась вторая итерация из-за обратного ребра, ведущего из B_3 в B_1. Третья итерация нужна, чтобы алгоритм распознал, что достиг неподвижной точки. Поскольку неподвижная точка единственна, компилятор вправе использовать этот более эффективный порядок.

	LiveOut(n)								
	B_0	B_1	B_2	B_3	B_4	B_5	B_6	B_7	B_8
0	∅	∅	∅	∅	∅	∅	∅	∅	∅
1	{i}	{a,c,i}	{a,b,c,d,i}	∅	∅	{a,c,d,i}	{a,c,d,i}	{a,b,c,d,i}	{a,c,d,i}
2	{i}	{a,c,i}	{a,b,c,d,i}	{i}	∅	{a,c,d,i}	{a,c,d,i}	{a,b,c,d,i}	{a,c,d,i}
3	{i}	{a,c,i}	{a,b,c,d,i}	{i}	∅	{a,c,d,i}	{a,c,d,i}	{a,b,c,d,i}	{a,c,d,i}

Рис. 9.6 Итерации решателя уравнений живучести с применением порядка РОП на обращенном ГПУ

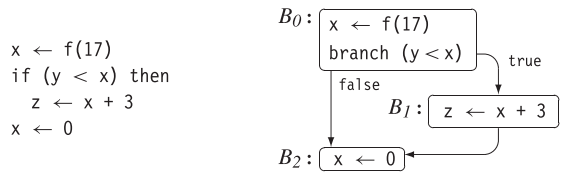
Такая схема применима ко многим задачам о потоке данных. На первой итерации вычисляются множества, которые правильны с точностью до эффектов циклов. На последующих итерациях алгоритм разбирается с информацией о циклах.

9.2.3. Ограничения на анализ потока данных

Существуют ограничения на то, что компилятор может узнать из анализа потока данных. В одних случаях причиной ограничений являются предположения, лежащие в основе анализа, в других – особенности анализируемого языка. Для принятия обоснованных решений автор компилятора должен понимать, на что способен анализа потока данных, а на что – нет.

При вычислении LiveOut(n) итеративный алгоритм использует множества LiveOut, UEVar и VarKill для каждого преемника n в ГПУ. Это действие неявно предполагает, что в процессе выполнения каждый из этих преемников может быть достигнут; на практике же один или несколько могут оказаться недостижимыми.

Рассмотрим фрагмент кода на рис. 9.7 и соответствующий ему ГПУ. Определение x в B_0 живо при выходе из B_0, потому что x используется в B_1. Определение x в B_2 уничтожает значение, установленное в B_0. Если B_1 не может быть выполнен, то значение x из B_0 не является живым после сравнения с y, и x ∉ LiveOut(B_0). Если компилятор может доказать, что y всегда меньше x, то B_1 никогда не будет выполнен. Компилятор может устранить B_1 и заменить проверку и условный переход в B_0 безусловным переходом на B_2. В этой точке если вызов f не имеет побочных эффектов, то компилятор может также устранить B_0.



(a) Простая конструкция if-then

(b) Соответствующий граф потока управления

Рис. 9.7 Поток управления ограничивает точность анализа потока данных

Однако в уравнениях LiveOut берется объединение по всем преемникам блока, а не только по исполняемым. Следовательно, анализатор вычисляет:

$$\text{LiveOut}(B_0) = (\text{UEVar}(B_1) \cup (\text{LiveOut}(B_1) \cap \overline{\text{VarKill}(B_1)})) \cup (\text{UEVar}(B_2) \cup (\text{LiveOut}(B_2) \cap \overline{\text{VarKill}(B_2)})).$$

Анализ потока данных предполагает, что все пути в ГПУ реализуемы. Таким образом, информация, которая по ним вычисляется, обобщает возможные события потока данных в предположении, что каждый путь может быть выбран. Это ограничивает точность результирующей информации; мы говорим, что информация точна «с точностью до символического выполнения». При таком предположении $x \in \text{LiveOut}(B_0)$ и обе вершины, B_0 и B_1, должны быть сохранены.

Статический и динамический анализы

Само существование статического анализа поднимает вопрос: а как насчет динамического анализа? По определению, статический анализ пытается на этапе компиляции оценить, что будет происходить на этапе выполнения. Во многих ситуациях компилятор не может сказать, что произойдет, пусть даже ответ был бы очевиден, если бы были известны одно или несколько значений во время выполнения.

Рассмотрим, к примеру, следующий фрагмент кода на C:

```
x = y * z + 12;
*p = 0;
q = y * z + 13;
```

Он содержит избыточное выражение `y * z` тогда и только тогда, когда `p` не является адресом `y` или `z`. На этапе компиляции значение `p` и адреса `y` и `z` могут быть неизвестны. Но на этапе выполнения они известны и могут быть проверены. Проверка этих значений во время выполнения могла бы исключить повторное вычисление `y * z` там, где анализ на этапе компиляции не способен ответить на этот вопрос.

Однако стоимость проверок `p == &y` и `p == &z` с последующим действием по результатам, скорее всего, превышает стоимость повторного вычисления `y * z`. Чтобы динамический анализ имел смысл, он априори должен быть целесообразным, т. е. экономия должна быть больше затрат на анализ. Иногда такое случается, но чаще – нет. С другой стороны, стоимость статического анализа можно разложить на несколько выполнений кода, так что в общем случае он предпочтительнее.

Другой путь проникновения неточности в результаты анализа потока данных связан с обработкой массивов, указателей и вызовов процедур. Ссылка на массив, например A[i,j], относится к одному элементу A. Однако без анализа, устанавливающего значения i и j, компилятор не может сказать, к какому именно элементу A имеет место обращение. Поэтому компиляторы традиционно рассматривали ссылку на элемент A как ссылку на весь A. Следовательно, использование A[i,j] считается как использование A, а определение A[m,n] считается как определение A.

Однако автору компилятора следует воздерживаться от слишком сильных выводов. Поскольку информация о массивах неточна, компилятор должен интерпретировать ее консервативно. Следовательно, если цель анализа – определить, где значение не живо, – т. е. *его следует уничтожить*, – то определение A[i,j] не уничтожает значение A. Если же цель – определить, где значение *могло бы не выжить*, то определение A[i,j] *могло бы* определить любой элемент A.

Анализ, применяемый для прослеживания возможных значений указателей, обходится дороже, чем классические задачи, такие как Dom и Live.

Указатели добавляют еще один уровень неточности к результатам статического анализа. Явные арифметические операции над указателями дополнительно ухудшают ситуацию. Если компилятор не трассирует значения указателей, то должен интерпретировать присваивание указательной переменной как потенциальное определение любой переменной, на которую теоретически может быть направлен указатель. Типобезопасность может ограничивать множество объектов, на которые может указывать указатель; если указатель определен как указывающий на объект типа *t*, то использовать его можно только для модификации объектов типа *t*. Без анализа значений указателя или гарантий, даваемых механизмом типобезопасности, присваивание переменной по указателю вынуждает анализатор предполагать, что модифицирована может быть любая переменная. На практике это часто препятствует компилятору хранить значение доступной по указателю переменной в регистре до и после присваивания по указателю. Если компилятор не может доказать, что указатель, упомянутый в присваивании, не может ссылаться на адрес памяти, соответствующий некоторому значению, значит, это значение нельзя безопасно хранить в регистре.

Сложность анализа указателей заставляет многие компиляторы избегать хранения значений в регистрах, если они могут быть целью указателя. Обычно некоторые переменные можно исключить из этого правила, например локальные переменные, для которых адрес никогда не берется явно. Альтернатива – выполнить анализ потока данных с целью устранить неоднозначность ссылок через указатели и тем самым сократить множество возможных переменных, на которые может ссылаться указатель в каждой точке кода. Если программа может передавать указатели в качестве параметров или использовать их как глобальные переменные, то такое устранение неоднозначности становится межпроцедурной задачей.

И последний источник неточности – вызовы процедур. Чтобы разобраться в потоке данных в текущей процедуре, компилятор должен знать, что вызываемая сторона может делать с каждой переменной, доступной как ей, так и вызывающей стороне. Вызываемая сторона может, в свою очередь, вызывать другие процедуры, у которых могут быть свои побочные эффекты.

Если компилятор не вычисляет точную сводную информацию о каждом вызове процедуры, то должен оценить поведение в худшем случае. Конкретные допущения зависят от задачи и языка, но общее правило – предполагать, что вызываемая сторона использует и модифицирует каждую переменную, до которой может дотянуться. Поскольку лишь немногие процедуры модифицируют и используют каждую переменную, это правило обычно дает завышенную оценку области воздействия вызова, что вносит дополнительную неточность в результаты анализа.

9.2.4. Другие задачи о потоке данных

Компиляторы используют анализ потока данных, чтобы доказать безопасность применения преобразования в конкретных ситуациях. Поэтому было поставлено много задач о потоках данных для конкретных оптимизаций.

Доступные выражения

Доступность
Выражение *e доступно* в точке *p*, если на любом пути из точки входа в процедуру к *p* выражение *e* вычисляется и ни один из его операндов не переопределяется.

Чтобы найти избыточные выражения, компилятор может вычислить информацию о *доступности* выражений. В процессе этого анализа каждая вершина ГПУ n аннотируется множеством AvailIn(n), содержащим имена выражений в процедуре, которые доступны при входе в блок n. Уравнения AvailIn имеют вид

$$\text{AvailIn}(n) = \bigcap_{m \in preds(n)} \left(\begin{array}{l} \text{DEExpr}(m) \cup \\ (\text{AvailIn}(m) \cap \overline{\text{ExprKill}(m)}) \end{array} \right)$$

с такими начальными значениями множеств AvailIn:

$$\text{AvailIn}(n_0) = \emptyset,$$
$$\text{AvailIn}(n) = \{\textit{все выражения}\}, \forall n \neq n_0.$$

Эти уравнения можно эффективно решить с помощью стандартного итеративного решателя потока данных. Поскольку это прямая задача, решатель должен использовать порядок РОП обхода ГПУ.

В этих уравнениях DEExpr(n) – множество экспонированных вниз выражений в n. Выражение $e \in$ DEExpr(n), если в блоке n вычисляется e и ни один из операндов e не переопределен между последним вычислением e в n и концом n. ExprKill(n) содержит все такие выражения, которые уничтожаются

вследствие переопределения в n. Выражение уничтожается, если хотя бы один из его операндов переопределен в блоке.

Выражение e доступно при входе в n, если оно доступно на выходе из каждого предшественника n в ГПУ. Уравнение говорит, что выражение e доступно на выходе из некоторого блока m, если выполняется одно из двух условий: либо e экспонируется вниз в m, либо оно доступно при входе в m и не уничтожается в m.

Множества AvailIn используются для глобального устранения избыточности, которое иногда называют *глобальным устранением общих подвыражений*. Быть может, самый простой способ решить эту задачу – вычислить множества AvailIn для каждого блока и использовать их в качестве начальных данных для локальной нумерации значений (см. раздел 8.4.1). Отложенное перемещение кода – более сильная форма устранения избыточности, в которой также используется доступность (см. раздел 10.3.1).

Достигающие определения

Достижимость
Определение d переменной x *достигает* операции u, если в u используется значение x и существует путь из d в u, на котором x не переопределяется.

В некоторых случаях компилятору нужно знать, где был определен операнд. Если к операции ведут несколько путей в ГПУ, то значение операнда могут предоставлять несколько определений. Чтобы найти множество определений, достигающих блока, компилятор может вычислить *достигающие определения*. Компилятор аннотирует каждую вершину ГПУ n множеством Reaches(n), которое содержит все определения, достигающие начала блока, соответствующего n. Областью определения Reaches является множество точек определения в процедуре – множество присваиваний.

Компилятор вычисляет множество Reaches(n) для каждой вершины ГПУ n, пользуясь уравнением

$$\text{Reaches}(n) = \bigcup_{m \in preds(n)} \left(\begin{array}{l} \text{DEDef}(m) \cup \\ (\text{Reaches}(m) \cap \overline{\text{DefKill}(m)}) \end{array} \right)$$

с такими начальными значениями множеств Reaches:

$$\text{Reaches}(n) = \emptyset, \forall n.$$

Здесь DEDef(m) – множество экспонированных вниз определений в m: таких, что определенное в них имя далее не переопределяется в m. DefKill(m) содержит *все* точки определения, маскируемые другим определением того же имени в m; $d \in$ DefKill(m), если d определяет некоторое имя v и m содержит определение, которое также определяет v. Таким образом, $\overline{\text{DefKill}(m)}$ содержит те точки определения, которые переживают прохождение через блок m.

DEDef и DefKill определены на множестве точек определения, но для вычисления каждого из них необходимо сначала отобразить имена (перемен-

ных и сгенерированных компилятором временных переменных) на точки определения. Следовательно, сбор начальной информации для вычисления достигающих определений обходится дороже, чем для вычисления живых переменных.

Предсказуемые выражения

Предсказуемость
Выражение *e* называется *предсказуемым* в точке *p*, если (1) *e* вычисляется на каждом пути, исходящем из *p*, и (2) вычисление *e* в *p* дало бы такой же результат, как первое вычисление на каждом из этих путей.

В некоторых ситуациях компилятор может переместить выражение выше в ГПУ и заменить несколько экземпляров выражения на разных путях одним. Эта оптимизация, называемая *поднятием* (hoisting), уменьшает размер кода. Она не изменяет количество вычислений выражения.

Чтобы найти безопасные возможности для поднятия, компилятор может вычислить множество *предсказуемых выражений* в конце каждого блока. Выражение *e* является предсказуемым в конце блока *b*, если следующее вычисление *e* на каждом пути, исходящем из *b*, дало бы такой же результат. Уравнения требуют, чтобы *e* вычислялось на каждом пути, исходящем из *b*.

AntOut, множество выражений, предсказуемых в конце блока, можно вычислить, решив обратную задачу о потоке данных на ГПУ. Задача о предсказуемости ставится на множестве выражений.

Реализация каркасов потока данных

Уравнения для многих задач о глобальном потоке данных на удивление похожи. Например, в уравнениях доступных выражений, живучести переменных, достигающих определений и предсказуемых выражений, функции распространения имеют вид

$f(x) = c_1\ op_1\ (x\ op_2\ c_2)$,

где c_1 и c_2 – константы, выведенные из кода, а op_1 и op_2 – стандартные теоретико-множественные операции типа $\cup$ и $\cap$. Это сходство проявляется и в описании задач; оно открывает возможность для разделения кода в реализациях анализатора.

Автор компилятора легко может абстрагировать детали, отличающие задачи друг от друга, и реализовать один параметризованный анализатор. Анализатору необходимы функции для вычисления c_1 и c_2, реализации операторов и указание на направление задачи (прямая или обратная). В ответ он порождает желаемые множества.

Эта стратегия реализации поощряет повторное использование кода. Она скрывает низкоуровневые детали решателя, а также создает ситуацию, в которой автор компилятора может с выгодой потратить время и силы на оптимизацию реализации. Например, схема, которая реализует $f(x) = c_1\ op_1\ (x\ op_2\ c_2)$ в виде одной функции, может оказаться производительнее схемы, которая реализует $f_1(x) = c_1\ op_1\ x$ и $f_2(x) = x\ op_2\ c_2$, а затем вычисляет $f(x)$ как $f_1(f_2(x))$. Единый каркас позволяет всем клиентским преобразованиям получить выигрыш от улучшения представления множеств и реализаций операторов.

Уравнения, определяющие AntOut, имеют вид:

$$\text{AntOut}(n) = \bigcup_{m \in succ(n)} \left(\begin{array}{l} \text{UEExpr}(m) \cup \\ (\text{AntOut}(m) \cap \overline{\text{ExprKill}(m)}) \end{array} \right)$$

с такими начальными значениями множеств AntOut:

$\text{AntOut}(n_f) = \emptyset$,
$\text{AntOut}(n) = \{$*все выражения*$\}$, $\forall n \neq n_f$.

Здесь UEExpr(m) – множество экспонированных сверху выражений, таких, которые используются в m до уничтожения. Множество ExprKill(m) содержит все выражения, уничтожаемые определением m; оно также встречается в уравнениях для нахождения доступных выражений.

Результаты анализа предсказуемости используются при отложенном перемещении кода – для уменьшения времени выполнения, и при поднятии кода – чтобы уменьшить размер откомпилированного кода. Оба преобразования обсуждаются в разделе 10.3.

Межпроцедурные сводные задачи

При анализе одной процедуры компилятор должен учитывать последствия каждого вызова процедуры. В отсутствие более точной информации о вызове компилятор должен предположить худшее о вызываемой стороне и о любых процедурах, которые она вызывает. Эти предположения могут значительно снизить точность информации о глобальном потоке данных. Например, компилятор должен предполагать, что вызываемая сторона модифицирует любую переменную, к которой может получить доступ; это предположение, по сути дела, останавливает распространение сведений обо всех глобальных переменных, переменных уровня модуля и параметрах, передаваемых по ссылке, далее места вызова.

Чтобы ограничить такие последствия, компилятор может вычислять сводную информацию о каждом месте вызова. Классические задачи о сводке вычисляют множество переменных, которые могут быть модифицированы в результате вызова или использованы в качестве результата вызова. Затем компилятор может использовать вычисленные множества вместо предположений для худшего случая.

Потоконезависимость
В этой формулировке MayMod игнорируется поток управления внутри процедур. Говорят, что эта формулировка *потоконезависима*.

Задача о *межпроцедурной потенциальной модификации* аннотирует каждое место вызова множеством имен, которые могут быть модифицированы вызываемой стороной и вызываемыми из нее процедурами. Потенциальная модификация – одна из самых простых задач межпроцедурного анализа, но она может серьезно повлиять на качество информации, порождаемой другими видами анализа, например глобального распространения констант. Задача о потенциальной модифи-

кации ставится в виде системы уравнений потока данных на графе вызовов программы; ее цель – аннотировать каждую процедуру множеством MayMod.

$$\text{MayMod}(p) = \text{LocalMod}(p) \cup (\cup_{e=(p,q)} unbind_e(\text{MayMod}(q))).$$

После инициализации MayMod(p) содержит все имена, которые локально модифицируются в p и видимы вне p. Оно вычисляется как множество имен, определенных в p, за вычетом имен, строго локальных по отношению к p.

Функция $unbind_e$ отображает одно множество имен в другое. Для ребра графа вызовов $e = (p, q)$ и множества имен s $unbind_e(s)$ отображает каждое имя в s из пространства имен q в пространство имен в месте вызова, используя привязки в месте вызова, соответствующем e. По существу, она проецирует s из пространства имен q в пространство имен p.

Зная набор множеств LocalMod и граф вызовов, итеративный решатель находит решение этих уравнений в форме неподвижной точки. Он не сможет достичь такой же скорости, как при анализе глобального потока данных. Для достижения почти линейной сложности на этой задаче требуется более сложный каркас (см. примечания к этой главе).

Множества MayMod, вычисленные по этим уравнениям, – это обобщенные сводные множества. То есть MayMod(q) содержит имена переменных, которые могут быть модифицированы в результате вызова q, выраженные в пространстве имен q. Чтобы использовать эту информацию в конкретном месте вызова q, компилятор должен вычислить множество $S = unbind_e(\text{MayMod}(q))$, где $e = (p, q)$ – ребро графа вызовов, соответствующее данному вызову. Затем компилятор должен добавить в S все имена, которые внутри p являются псевдонимами имен, содержащихся в S.

Компилятор также может вычислить множество переменных, на которые возможна ссылка в результате выполнения вызова процедуры, т. е. решить задачу о *межпроцедурной потенциальной ссылке*. Уравнения, которые нужно решить для аннотирования каждой процедуры p множеством MayRef(p), аналогичны уравнениям MayMod.

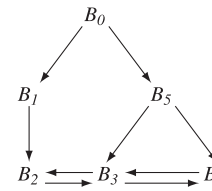
ИТОГИ РАЗДЕЛА

Принцип работы итеративного анализа потока данных – повторяющееся вычисление уравнений в каждой вершине некоторого графа до тех пор, пока множества, определяемые уравнениями, не перестанут изменяться, т. е. не будет найдена неподвижная точка. Во многих задачах о потоке данных неподвижная точка единственна, что гарантирует нахождение правильного решения вне зависимости от порядка вычислений, а кроме того, имеет место свойство конечной убывающей цепочки, гарантирующее остановку алгоритма вне зависимости от порядка вычислений. Эти два свойства позволяют автору компилятора выбирать порядок вычислений, при котором алгоритм быстро сходится. Поэтому итеративный анализ надежен и эффективен.

В литературе описано много разных задач о потоке данных. В этом разделе мы рассмотрели в качестве примеров задачи о доминировании, анализе живучести, доступности, предсказуемости и межпроцедурном обобщении. У всех них, кроме межпроцедурных задач, имеются прямолинейные эффективные решения с помощью итеративного алгоритма. Чтобы избежать решения нескольких задач по отдельности, компиляторы часто обращаются к унифицированному каркасу, например форме SSA, описанной в следующем разделе.

ВОПРОСЫ

1. Вычислите множества Dom для ГПУ во врезке, перебирая вершины в порядке $\{B_4, B_2, B_1, B_5, B_3, B_0\}$. Объясните, почему при таком порядке количество итераций отличается от показанного на стр. 475.
2. Неоднозначные вызовы могут осложнить процесс построения компилятором графа вызовов точно так же, как неоднозначные переходы осложняют построение ГПУ. Какие особенности языка могли бы привести к неоднозначному месту вызова, т. е. такому, где компилятор не уверен, как процедура будет вызвана в итоге?



9.3. Форма с единственным статическим присваиванием

Со временем авторы компиляторов сформулировали много разных задач о потоке данных. Если бы для каждого преобразования был нужен свой анализ, то затраты на реализацию, отладку и сопровождение аналитических проходов оказались бы неоправданно велики. Чтобы ограничить количество видов анализа, желательно использовать один и тот же анализ для нескольких преобразований.

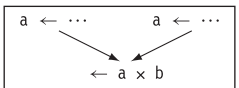
В некоторых компиляторах, например LLVM/CLANG, форма SSA является определяющим ПП.

Одна из возможных стратегий такого «универсального» анализа заключается в построении ПП, которое называется формой с единственным статическим присваиванием (SSA) (см. также раздел 4.6.2). В SSA поток данных и поток управления закодированы непосредственно в ПП. Многие классические скалярные оптимизации были переосмыслены и теперь работают с кодом в форме SSA.

Код в форме SSA подчиняется двум правилам:

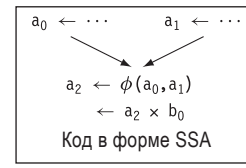
1) каждое вычисление в процедуре определяет уникальное имя;
2) каждое использование в процедуре ссылается на одно имя.

Первое правило устраняет эффект «уничтожений» из кода; любое выражение доступно в любой точке после его вычисления. (Впервые мы повстречались с этим эффектом в алгоритме локальной нумерации значений.) Эффект второго правила тоньше. Оно гарантирует, что компилятор по-прежнему может представить код лаконично и корректно; использование можно записать с помощью одного имени, а не длинного списка всех достигающих его определений.



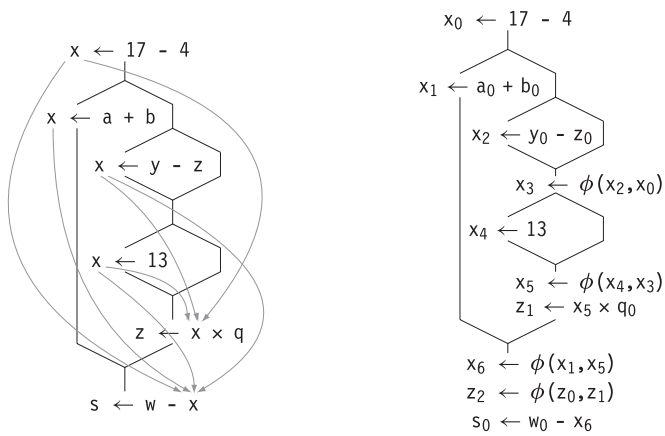
Исходный код

Рассмотрим простой пример во врезке. Если компилятор переименует два определения a в a_0 и a_1, то какое имя следует употребить при использовании a в выражении a × b? Ни a_0, ни a_1 работать в a × b не могут. (В этом примере предполагается, что b было определено раньше.)



Для управления этим пространством имен процесс построения SSA вставляет специальный вид операции копирования, φ-функцию, в начало блока, где встречаются пути потока управления, как показано во врезке. В момент вычисления φ-функция читает аргумент, соответствующий ребру, по которому управление попало в блок. Следовательно, если мы пришли в блок слева, то φ-функция читает a_0, а если справа, то a_1. Выбранный аргумент присваивается a_2. Таким образом, вычисление $a_2 \times b_0$ дает то же значение, что вычисление $a \times b$ в коде до преобразования в SSA-форму.

На рис. 9.8 показан более пространный пример. Рассмотрим различные использования переменной x во фрагменте кода на рисунке (a). Кривые серые линии показывают, какие определения могут достигнуть каждого использования x. На рисунке (b) тот же фрагмент представлен в форме SSA. Переменные переименованы, уникальность имен в каждом определении обеспечивается с помощью индексов. Мы предполагаем, что a_0, b_0, w_0, y_0 и z_0 определены раньше.



(a) Исходный фрагмент кода

(b) После преобразования x и z в SSA-форму

Рис. 9.8 SSA. Кодирование потока управления в потоке данных

Код на рисунке (b) включает все φ-функции, необходимые для согласования имен, сгенерированных по правилу 1, с потребностью в уникальных именах в точках использования. Трассировка потока значений обнаружит, что одни и те же значения распространяются по тем же путям, что и в исходном коде.

И еще два момента, касающихся φ-функций, нуждаются в пояснении. Во-первых, φ-функции, по определению, выполняются конкурентно. Когда управление входит в блок, все φ-функции этого блока параллельно читают предназначенные им аргументы. Затем все они параллельно определяют свои целевые имена. Эта семантика конкурентного выполнения позволяет

алгоритму построения SSA-формы игнорировать порядок вставки φ-функций в блок.

Во-вторых, по соглашению, мы записываем аргументы φ-функции слева направо, чтобы не расходиться с расположением входящих ребер слева направо на печатной странице. Но внутри компилятора у ПП нет естественного понятия «слева направо» для перечисления ребер, входящих в блок. Поэтому в реализации необходимы дополнительные механизмы, чтобы отслеживать соответствие между аргументами φ-функции и ребрами ГПУ.

9.3.1. Наивный метод построения SSA-формы

Оба алгоритма построения SSA-формы, которые мы представим, основаны на одной и той же схеме: (1) вставить необходимые φ-функции и (2) переименовать переменные и временные значения в соответствии с обоими правилами, определяющими форму SSA. Самый простой метод построения реализует оба шага следующим образом.

«Наивный» алгоритм вставляет больше φ-функций, чем необходимо. Он добавляет φ-функцию для каждого имени в каждой точке соединения.

1. *Вставка φ-функций.* В начало каждого блока, имеющего несколько предшественников в ГПУ, вставить φ-функцию, например $x \leftarrow \varphi(x,x)$, для каждого имени x, определенного в текущей процедуре. φ-функция должна иметь один аргумент для каждого предшествующего блока в ГПУ. Этот процесс вставляет φ-функции во все места, где они необходимы. Но многие из этих функций лишние.
2. *Переименование.* Алгоритм вставки φ-функций гарантирует, что φ-функция для x будет находиться в каждой точке соединения в ГПУ, которой достигают два или более определения x. Алгоритм переименования переписывает все имена в подходящие SSA-имена. На первом шаге к имени добавляется уникальный индекс в каждой точке определения. В этот момент с каждым определением связано уникальное SSA-имя. Компилятор может вычислить достигающие определения (см. раздел 9.2.4) и определить, какое SSA-имя достигает каждого использования. Автор компилятора должен изменить семантику DefKill таким образом, чтобы определение одного SSA-имени уничтожало не только это SSA-имя, но и все SSA-имена с тем же базовым именем. Смысл в том, чтобы остановить распространение SSA-имени в любой φ-функции, аргументом которой оно является. После такого изменения ровно одно определение – одно SSA-имя – достигает каждого использования.

Базовое имя
Для SSA-имени x_2 базовым именем является x, а его версия равна 2.

Компилятор выполняет проход по коду, чтобы переписать имя в каждой точке использования достигающим этой точки SSA-именем. При этом переписываются все использования, в т. ч. и в аргументах φ-функции. Если одно и то же SSA-имя достигает φ-функции по раз-

ным путям, у соответствующих аргументов φ-функции будут одинаковые SSA-имена.

Компилятор должен установить соответствие между входящими ребрами в ГПУ и аргументами φ-функции, чтобы каждый аргумент можно было переименовать, присвоив ему правильное SSA-имя. Концептуально эта задача проста, но требует кое-какого внутреннего учета.

φ-функция $x_j \leftarrow \varphi(x_i, x_i)$ избыточна.

φ-функция, значение которой не живо, считается мертвой.

Этот наивный алгоритм строит SSA-форму с соблюдением обоих правил. Каждому определению сопоставляется уникальное имя; в каждой точке использования употребляется имя из отдельного определения. Но хотя алгоритм строит правильную SSA-форму, он может вставлять избыточные или мертвые φ-функции. Эти лишние φ-функции могут создавать проблемы. Компилятор зря расходует на них память, а на их обход тратится время. Они могут снизить точность некоторых видов анализа программы в форме SSA.

Мы будем называть такой вариант *максимальной SSA-формой*. Чтобы построить SSA-форму с меньшим числом φ-функций, потребуется дополнительная работа; в частности, компилятор должен проанализировать код и определить, где в ГПУ сходятся потенциально различные значения. Это вычисление опирается на информацию о доминировании, описанную в разделе 9.2.1.

Различные варианты SSA-формы

В литературе предложено несколько вариантов SSA-формы. Различаются они критериями вставки φ-функций и для заданной программы могут порождать разные множества φ-функций.

В случае *минимальной SSA-формы* φ-функция вставляется в каждой точке соединения, где встречаются два определения одного и того же исходного имени. Это минимальное число, совместимое с определением SSA. Однако некоторые из этих φ-функций могут быть мертвы; определение ничего не говорит о том, должны ли значения быть живы в точке встречи.

В случае *усеченной SSA-формы* в алгоритм вставки φ-функции добавляется проверка живучести, чтобы избежать добавления мертвых φ-функций. По ходу дела нужно вычислять множества LiveOut, что увеличивает стоимость построения усеченной SSA-формы.

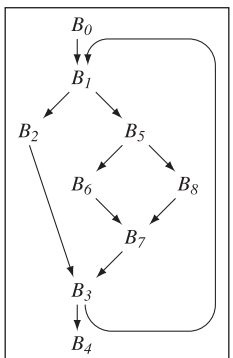
Полуусеченная SSA-форма – компромисс между минимальной и усеченной формами. До вставки φ-функций алгоритм исключает имена, которые не остаются живы после пересечения границы блока. Это может сузить пространство имен и уменьшить число φ-функций без затрат на вычисление множеств LiveOut. Алгоритм на рис. 9.11 вычисляет полуусеченную SSA-форму.

Разумеется, количество φ-функций зависит от конкретной программы, преобразуемой в форму SSA. Для некоторых программ сокращение, достигаемое в результате усечения и полуусечения SSA, существенно. Уменьшение SSA-формы может ускорить компиляцию, потому что в этом случае проходы, на которых обрабатывается SSA-форма, оперируют программой, содержащей меньше операций – и меньше φ-функций.

В следующих подразделах подробно описан алгоритм построения *полуусеченной SSA-формы* – версии с меньшим числом φ-функций. В разделе 9.3.2 вводится понятие *границ доминирования* и показано, как они вычисляются; границы доминирования направляют вставку φ-функций. В разделе 9.3.3 представлен алгоритм вставки φ-функций, а в разделе 9.3.4 – эффективный алгоритм переименования. В разделе 9.3.5 обсуждаются осложнения, которые могут возникнуть при трансляции из SSA-формы.

9.3.2. Границы доминирования

Основная проблема максимальной SSA-формы заключается в том, что она содержит слишком много φ-функций. Чтобы уменьшить это число, компилятор должен более точно определить, где они нужны. Ключ к вставке φ-функции – в понимании того, какие имена нуждаются в φ-функции в каждой точке соединения. Для эффективного решения этой проблемы компилятор может поставить вопрос по-другому. Он может для каждого блока i определить множество блоков, которым φ-функция понадобится вследствие определения в блоке i. В этом вычислении критическую роль играет доминирование.



Пример ГПУ

Рассмотрим ГПУ, показанный во врезке. Предположим, что код присваивает разные значения a в блоках B_1 и B_5 и что больше ни в каком блоке значение a не присваивается. Значение из B_5 – единственное значение a, которое может достичь B_6, B_7 и B_8. Поскольку B_5 доминирует над этими тремя блоками, он лежит на любом пути из B_0 в B_6, B_7 или B_8. Определение в B_1 не может достичь их.

С блоком B_3 ситуация иная. Ни один из его предшественников в ГПУ, B_2 и B_7, не доминирует над B_3. Использование a в B_3 может получать значение от B_1 или B_5, в зависимости от того, по какому пути был достигнут B_3. Присваивания a в B_1 и B_5 вынуждают поставить φ-функцию для a в начало B_3.

B_5 доминирует над регионом (B_6, B_7, B_8). Это непосредственный доминатор всех трех вершин. Определение a в B_5 достигнет использования в этом регионе, если только a не будет переопределена перед использованием. Определение в B_5 не является обоснованием необходимости φ-функции в этом регионе.

Строгое доминирование
Говорят, что вершина ГПУ p *строго доминирует* над вершиной q, если $p \in \mathrm{Dom}(q)$ и $p \neq q$.

Мы обозначаем это $p \in (\mathrm{dom}(q) - q)$.

B_3 лежит вне региона, над которым доминирует B_5. У него есть два предшественника в ГПУ, и B_5 доминирует только над одним из них. Таким образом, B_3 лежит на ребре ГПУ, находящемся вне региона, над которым доминирует B_5. В общем случае определение a в таком блоке B_i обусловливает необходимость φ-функции в любой вершине, которая, подобно B_3, на одно ребро выступает за пределы региона, над ко-

торым доминирует B_i. *Границей доминирования* B_i, обозначаемой DF(B_i), называется множество всех таких вершин.

Граница доминирования
Говорят, что вершина ГПУ q принадлежит *границе доминирования* вершины p, если (1) p доминирует над каким-то предшественником q в ГПУ и (2) p не строго доминирует над q.

Мы обозначаем границу доминирования p DF(p).

Повторим: $q \in \mathrm{DF}(p)$, если на некотором пути q на одно ребро выступает за пределы региона, над которым доминирует p. Таким образом:

- q имеет в ГПУ предшественника, над которым доминирует p. Существует такая x, что (x, q) является ребром ГПУ и $p \in \mathrm{Dom}(x)$;
- p нестрого доминирует над q. То есть $p \notin (\mathrm{Dom}(q) - q)$.

DF(p) – это просто множество всех вершин, удовлетворяющих обоим этим условиям.

Определение a в блоке n вынуждает вставить φ-функцию для a в начало каждого блока $m \in \mathrm{DF}(n)$. На рис. 9.9 показаны множества Dom, IDom и DF для нашего примера ГПУ.

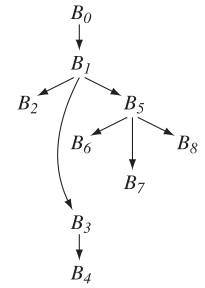
	B_0	B_1	B_2	B_3	B_4	B_5	B_6	B_7	B_8
DOM	{0}	{0,1}	{0,1,2}	{0,1,3}	{0,1,3,4}	{0,1,5}	{0,1,5,6}	{0,1,5,7}	{0,1,5,8}
IDOM	–	0	1	1	3	1	5	5	5
DF	∅	1	3	1	∅	3	7	3	7

Рис. 9.9 Множества Dom, IDom и DF для примера ГПУ

Обратите внимание на роль строгого доминирования. В нашем примере ГПУ строгое доминирование гарантирует, что $B_1 \in \mathrm{DF}(B_1)$. Таким образом, присваивание какому-то имени a в B_1 вынуждает вставить φ-функцию в B_1. Если бы в определении границ доминирования использовалось вместо этого множество Dom, то DF(B_1) было бы пустым.

Деревья доминирования

Дерево доминирования
дерево, в котором закодирована информация о доминировании в графе потока.



Дерево доминирования для примера ГПУ

В алгоритме вычисления границ доминирования используется структура данных, *дерево доминирования*, для представления отношений доминирования. В дереве доминирования имеется вершина для каждого блока ГПУ. Ребра кодируют непосредственное доминирование; если m = IDom(n), то n – непосредственный потомок m в дереве доминирования.

Дерево доминирования также кодирует множества Dom. Для вершины n Dom(n) содержит в точности вершины на пути из n к корню дерева доминирования. Вершины

на этом пути упорядочены по отношению IDom. Дерево доминирования для нашего примера показано во врезке.

Вычисление границ доминирования

Чтобы обеспечить эффективность вставки φ-функций, компилятор должен для каждой вершины ГПУ заранее вычислить множество DF(n), содержащее границу доминирования n. В алгоритме, показанном на рис. 9.10, для построения множеств DF(n) используется и дерево доминирования, и ГПУ.

```
for каждой вершины ГПУ n do
  DF(n) ← ∅
for каждой вершины ГПУ n do
  if n имеет несколько предшественников then
    for каждого предшественника p вершины n do
      runner ← p
      while runner ≠ IDOM(n) do
        DF(runner) ← DF(runner) ∪ { n }
        runner ← IDOM(runner)
```

Рис. 9.10 Алгоритм вычисления границ доминирования

Обратите внимание, что множества DF могут содержать только вершины, являющиеся точками соединения в ГПУ, т. е. вершины, имеющие несколько предшественников. Следовательно, алгоритм начинает работу с точек соединения. В точке соединения n обходит ее предшественников p в ГПУ и вставляет n в DF(p) по мере необходимости.

Если p = IDom(n), то p также доминирует над всеми другими предшественниками n. В нашем примере B_0 = IDom(B_1).

- Если p = IDom(n), то n не принадлежит DF(p). Она также не принадлежит DF(m) для любого предшественника m вершины p.
- Если $p \neq$ IDom(n), то n принадлежит DF(p). Она также принадлежит DF(q) для любой q такой, что $q \in$ Dom(p) и $q \notin$ (Dom(n) − n). Алгоритм находит такие вершины q, поднимаясь по дереву доминирования.

Правильность алгоритма вытекает из следующих наблюдений. Он инициализирует DF(n) пустым множеством ∅ для всех вершин ГПУ n. Затем он находит все точки соединения в ГПУ и обходит предшественников p каждой точки соединения n. Если p доминирует над n, то обработка p закончена. В противном случае он добавляет n в DF(p) и поднимается вверх по дереву доминирования, добавляя n в множество DF каждого предка в дереве доминирования, пока не найдет непосредственный доминатор n. Алгоритму необходимы дополнительные меры, чтобы избежать многократного добавления n в множество DF.

Снова рассмотрим наш пример ГПУ и его дерево доминирования. Анализатор исследует вершины в некотором порядке, ища те, у которых несколько предшественников. В предположении, что вершины перебираются в порядке

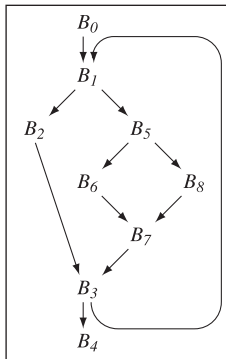
следования имен, он находит точки соединения в порядке B_1, затем B_3 и потом B_7.

B_1 Для предшественника в ГПУ B_0 алгоритм находит, что B_0 является IDom(B_1), поэтому в цикл while не входит. Для предшественника B_3 он добавляет B_1 в DF(B_3) и полагает *runner* равным IDom(B_3) = B_1. Алгоритм добавляет B_1 в DF(B_1) и полагает *runner* равным IDom(B_1) = B_0, после чего останавливается.

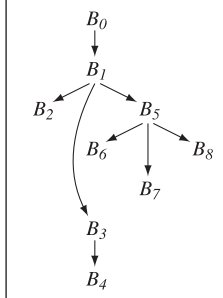
B_3 Для предшественника B_2 алгоритм добавляет B_3 в DF(B_2) и полагает *runner* равным IDom(B_2) = B_1. Так как B_1 = IDom(B_3), он останавливается. Для предшественника B_7 алгоритм добавляет B_3 в DF(B_7) и полагает *runner* равным IDom(B_7) = B_5. Он добавляет B_3 в DF(B_5) и полагает *runner* равным IDom(B_5) = B_1, после чего останавливается.

B_7 Для предшественника B_6 алгоритм добавляет B_7 в DF(B_6) и полагает *runner* равным IDom(B_6) = B_5, после чего останавливается. Для предшественника B_8 он добавляет B_7 в DF(B_8) и полагает *runner* равным IDom(B_8) = B_5, после чего останавливается.

В результате получаются множества DF, показанные на рис. 9.9.



Пример ГПУ



Пример дерева доминирования

9.3.3. Размещение φ-функций

Наивный алгоритм помещал φ-функцию для каждой переменной в начало каждой точки соединения. Зная границы доминирования, компилятор может более точно определить, где могут понадобиться φ-функции. Основная идея проста.

- С точки зрения потока управления, присваивание переменной x в вершине ГПУ n индуцирует вставку φ-функции в каждой вершине ГПУ $m \in$ DF(n). Каждая вставленная φ-функция создает новое присваивание; это присваивание может, в свою очередь, индуцировать дополнительные φ-функции.
- С точки зрения потока данных, φ-функция необходима, только если ее результат жив в точке вставки. Компилятор мог бы вычислить информацию о живучести и проверить каждую φ-функцию при вставке; такой подход ведет к *усеченной SSA-форме*.

На практике компилятор может избежать большинства мертвых φ-функций с помощью дешевой аппроксимации живучести. Имя x заведомо не нуждается в φ-функции, если только оно не является живым в нескольких блоках. Компилятор может вычислить множество *глобальных* имен – тех, что живы

Здесь слово *глобальная* означает, что имя представляет интерес во всей процедуре.

в нескольких блоках. При построении SSA-формы можно игнорировать все неглобальные имена, что уменьшает пространство имен и количество φ-функций. Получающаяся при этом SSA-форма называется *полуусеченной*.

Компилятор может найти глобальные имена без больших затрат. В каждом блоке он ищет имена с экспонированными сверху использованиями – множество UEVar из вычисления живых переменных. Любое имя, встречающееся в множестве LiveOut, должно находиться в множестве UEVar какого-то блока. Взяв объединение всех множеств UEVar, компилятор получит множество имен, которые живы при входе в один или более блоков, а значит, живы в нескольких блоках.

Алгоритм нахождения глобальных имен, показанный на рис. 9.11(a), выведен из очевидного алгоритма нахождения UEVar. Он строит множество глобальных имен, *Globals*, и для каждого имени – множество блоков, содержащих определение этого имени. Алгоритм использует эти списки блоков, чтобы построить начальные рабочие списки для вставки φ-функций.

```
Globals ← ∅
Инициализировать все множества Blocks значением ∅
for каждого блока b
  VarKill ← ∅
  for каждой операции i в b по порядку
    предполагаем, что op_i вида "x ← y op z"
    if y ∉ VarKill then
      Globals ← Globals ∪ { y }
    if z ∉ VarKill then
      Globals ← Globals ∪ { z }
    VarKill ← VarKill ∪ { x }
  Blocks(x) ← Blocks(x) ∪ { b }
```

(a) Нахождение глобальных имен

```
for каждого имени x ∈ Globals
  WorkList ← Blocks(x)
  for каждого блока b ∈ WorkList
    удалить b из WorkList
    for каждого блока d в DF(b)
      if d не имеет φ-функции для x then
        вставить φ-функцию для x в d
        WorkList ← WorkList ∪ { d }
```

(b) Вставка φ-функций

Рис. 9.11 Вставка φ-функций

Алгоритм вставки φ-функций на рис. 9.11(b) обходит глобальные имена. Для каждого имени x он инициализирует *WorkList* значением *Blocks(x)*. Для каждого блока b в *WorkList* он вставляет φ-функцию в начало каждого блока d, принадлежащего границе доминирования b. Семантика параллельного выполнения φ-функций позволяет алгоритму вставлять их в начало d в любом порядке. Добавляя φ-функцию для x в d, алгоритм добавляет d в *WorkList*, чтобы отразить новое присваивание x в d.

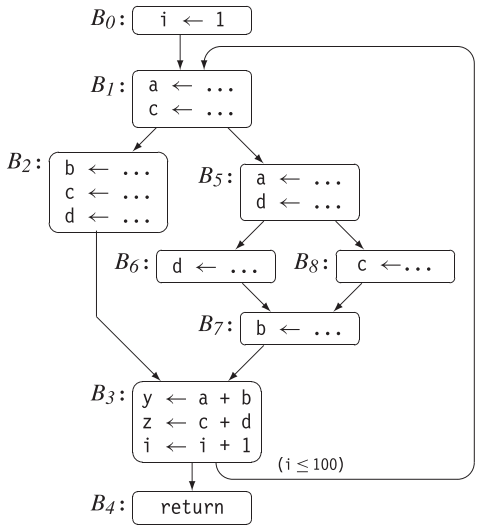
Пример

На рис. 9.12 снова показан наш сквозной пример: на рисунке (a) – код, а на рисунке (b) – границы доминирования в ГПУ.

Компилятор мог бы избежать вычисления множеств *Blocks* для неглобальных имен ценой дополнительного прохода по коду.

На первом шаге алгоритм вставки φ-функций находит глобальные имена и вычисляет множества *Blocks* для каждого имени. Глобальными

являются имена {a, b, c, d, i}. Множества *Blocks* для глобальных имен показаны на рисунке (c). Хотя алгоритм вычисляет множества *Blocks* также для y и z, в таблице эти имена опущены, потому что не являются глобальными.



(a) Код в простых блоках

	B_0	B_1	B_2	B_3	B_4
DF	$\emptyset$	$\{B_1\}$	$\{B_3\}$	$\{B_1\}$	$\emptyset$

	B_5	B_6	B_7	B_8
DF	$\{B_3\}$	$\{B_7\}$	$\{B_3\}$	$\{B_7\}$

(b) Границы доминирования в ГПУ

Name	Blocks
a	$\{B_1, B_5\}$
b	$\{B_2, B_7\}$
c	$\{B_1, B_2, B_8\}$
d	$\{B_2, B_5, B_6\}$
i	$\{B_0, B_3\}$

(c) Множества блоков для каждого глобального имени

Рис. 9.12 Пример кода для вставки φ-функций

Алгоритм вставки φ-функций, показанный на рис. 9.11(b), обрабатывает каждое имя. Рассмотрим его действия для переменной a. Сначала он инициализирует рабочий список значением *Blocks*(a) = {B_1, B_5}, чтобы отразить тот факт, что a определена в B_1 и B_5.

Поскольку a определена в B_1, необходимо вставить φ-функцию для a в начало каждого блока, принадлежащего DF(B_1) = {B_1}. φ-функция в B_1 – это новое присваивание, поэтому алгоритм добавляет B_1 в *WorkList*. Затем алгоритм удаляет B_5 из рабочего списка и вставляет φ-функцию в каждый блок, принадлежащий DF(B_5) = {B_3}. Новая φ-функция в B_5 понуждает алгоритм добавить B_3 в рабочий список. После извлечения B_1 из рабочего списка алгоритм обнаруживает, что φ-функция, индуцированная B_1, уже существует в B_1. Он не добавляет ни дубликат φ-функции, ни блоки в *Worklist*. Выбрав B_3 из рабочего списка, алгоритм также находит φ-функцию для a в B_1. В этот момент *WorkList* оказывается пустым и обработка a завершается.

Следуя той же логике для каждого имени в *Globals*, алгоритм порождает следующие вставки:

	a	b	c	d	i
φ-функции	$\{B_1,B_3\}$	$\{B_1,B_3\}$	$\{B_1,B_3,B_7\}$	$\{B_1,B_3,B_7\}$	$\{B_1\}$

Результирующий код показан на рис. 9.13.

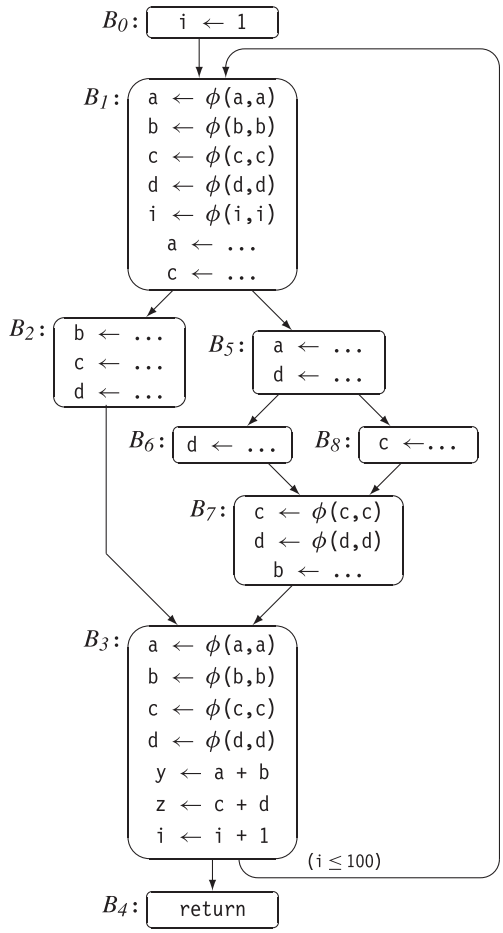


Рис. 9.13 Пример кода после вставки φ-функций

Ограничение алгоритма глобальными именами не дает ему вставлять мертвые φ-функции для переменных y и z в блок B_1. ($B_1 \in \mathrm{DF}(B_3)$, и в B_3 определены y и z.) Однако различия между локальными и глобальными именами недостаточно, чтобы избежать всех мертвых φ-функций. Например, φ-функция для b в B_1 не живая, потому что b переопределена перед использованием ее значения. Чтобы избежать вставки таких φ-функций, компилятор может построить множества LiveOut и добавить проверку живучести во внутренний цикл алгоритма вставки φ-функций. При такой модификации алгоритм порождает *усеченную SSA-форму*.

Повышение эффективности

Чтобы повысить эффективность, компилятор должен избегать двух видов дублирования. Во-первых, алгоритм не должен помещать никакой блок в рабочий список более одного раза на каждое глобальное имя. Он может хранить

список уже обработанных блоков для текущего имени и заново инициализировать его в начале обработки нового имени.

Во-вторых, данный блок может принадлежать границе доминирования нескольких вершин, встречающихся в *WorkList*. При каждой вставке алгоритм должен проверять, существует ли уже φ-функция для текущего имени. Вместо того чтобы искать φ-функции в блоке, компилятор должен хранить список блоков, уже содержащих φ-функции для текущей переменной. В начале обработки нового имени этот список тоже нужно опустошать.

Оба списка можно реализовать как разреженные множества (см. приложение B.2.3).

9.3.4. Переименование

Раньше мы отметили, что алгоритм переименования переменных концептуально прост. Однако детали нуждаются в пояснении.

В окончательной SSA-форме каждое глобальное имя становится базовым именем, а отдельные определения этого базового имени различаются добавленным индексом. Для имени, которое соответствует переменной исходного языка, скажем `a`, алгоритм использует `a` в качестве базового имени. Таким образом, первое определение `a`, встретившееся алгоритму переименования, получит имя a_0, а второе – a_1. Для временной переменной, сгенерированной компилятором, алгоритм может использовать ее имя в представлении, предшествующем SSA, в качестве базового.

Алгоритм, показанный на рис. 9.14, переименовывает определения и использования, обходя дерево доминирования процедуры в прямом порядке. В каждом блоке он сначала переименовывает значения, определенные φ-функциями в начале блока. Затем он посещает все операции в блоке по порядку. Он заменяет операнды текущими SSA-именами, после чего создает новое SSA-имя для результата операции. Это последнее действие делает новое имя текущим. После того как все операции в блоки переписаны, алгоритм переписывает соответствующие параметры φ-функции в каждом преемнике блока в ГПУ с использованием текущих SSA-имен. Наконец, он рекурсивно переходит к потомкам блока в дереве доминирования. После возврата из этих рекурсивных вызовов он восстанавливает множество текущих имен в состояние, которое существовало перед посещением текущего блока.

Для управления именами алгоритм использует счетчик и стек для каждого глобального имени. В стеке имени хранится индекс из его текущего SSA-имени. Для каждого определения алгоритм генерирует новый индекс для определяемого базового имени, для чего помещает значение его текущего счетчика в стек и увеличивает счетчик на единицу. Следовательно, значение на вершине стека для n всегда равно индексу текущего SSA-имени n.

На последнем шаге, после рекурсивной обработки потомков блока в дереве доминирования, алгоритм выталкивает все имена, сгенерированные в этом блоке, из их стеков. Это действие делает видимыми имена, которые

имели место в конце непосредственного доминатора этого блока. Они могут понадобиться для обработки оставшихся узлов дерева доминирования, находящихся на том же уровне, что текущий блок.

```
// Алгоритм переименования
for каждого глобального имени i do
  counter[i] ← 0
  stack[i] ← ∅

Rename(корень ГПУ)


NewName(n)
  i ← counter[n]
  counter[n] ← counter[n] + 1
  поместить i в stack[n]
  return "ni"
```

```
Rename(b)
  for каждой φ-функции в b, "x ← φ(...)" do
    переписать x как NewName(x)
  for каждой операции "x ← y op z" в b do
    if y ∈ Globals then
      переписать y с индексом top(stack[y])
    if z ∈ Globals then
      переписать z индексом top(stack[z])
    if x ∈ Globals then
      переписать x как NewName(x)
  for каждого преемника b в ГПУ do
    заполнить параметры φ-функции
  for каждого преемника s блока b в дереве
      доминирования do
    Rename(s)
  for каждой операции "x ← y op z" в b и
      каждой φ-функции "x ← φ(...)" do
    pop(stack[x])
```

Рис. 9.14 Алгоритм переименования после вставки φ-функций

Стек и счетчик служат разным целям. По мере того как алгоритм перемещается вверх и вниз по дереву доминирования, стек используется для эмуляции времени жизни самого последнего определения в текущем блоке. С другой стороны, счетчик монотонно возрастает, чтобы каждое последующее определение получало уникальное SSA-имя.

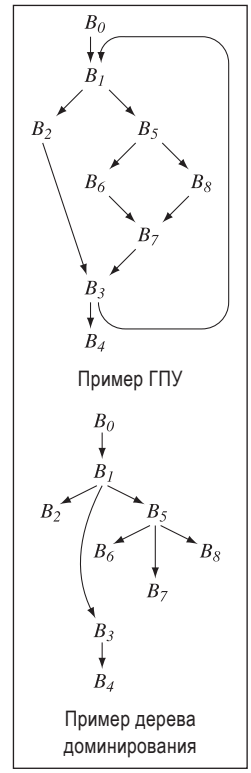
На рис. 9.14 приведен псевдокод алгоритма. Он инициализирует стеки и счетчики и вызывает *Rename* для корня дерева доминирования – входной вершины ГПУ. *Rename* обрабатывает блок, обновляет аргументы φ-функций в его блоках-преемниках в ГПУ и рекурсивно вызывает себя для потомков блока в дереве доминирования. Для завершения обработки блока *Rename* выталкивает из стеков все имена, добавленные во время его обработки. Функция *NewName* манипулирует счетчиками и стеками для создания новых SSA-имен по мере необходимости.

Остается последняя деталь. Когда *Rename* переписывает параметры φ-функций в каждом преемнике b в ГПУ, ей необходимо отображение b на слот его параметров в этих φ-функциях для b. То есть она должна знать, какой слот параметров в φ-функциях соответствует b.

Рисуя SSA-форму, мы предполагаем порядок слева направо, что соответствует порядку, в котором изображаются ребра. Но на внутреннем уровне компилятор может нумеровать ребра и слоты параметра любым согласованным способом – лишь бы получить желаемый результат. Это требует кооперации между кодом, который строит SSA-форму, и кодом, который строит ГПУ.

(Например, если в реализации ГПУ используется список ребер, исходящих из каждого блока, то порядок в этом списке может определять отображение.)

Пример



Чтобы закончить наш сквозной пример, давайте применим алгоритм переименования к коду на рис. 9.13. Предположим, что a_0, b_0, c_0 и d_0 определены при входе в B_0. На рис. 9.15 показаны состояния счетчиков и стеков для глобальных имен в различных точках процесса.

Алгоритм производит обход дерева доминирования в прямом порядке, который в данном примере соответствует посещению узлов в порядке возрастания имен. На рис. 9.15(a) показано начальное состояние стеков и счетчиков. В процессе работы алгоритм выполняет следующие действия.

Блок B_0. Этот блок содержит только одну операцию. Функция *Rename* заменяет `i` на i_0, увеличивает счетчик и помещает i_0 в стек для `i`. Затем она посещает преемника B_0 в ГПУ, B_1, и переписывает параметры φ-функции, соответствующие B_0, их текущими именами: a_0, b_0, c_0, d_0 и i_0. Далее она рекурсивно обрабатывает сыновний узел B_0 в дереве доминирования, B_1. После этого она выталкивает элемент из стека для `i` и возвращает управление.

Блок B_1. Состояние *Rename* при входе в B_1 показано на рис. 9.15(b). Она переписывает цели φ-функции новыми именами, a_1, b_1, c_1, d_1 и i_1. Затем она создает новые имена для определений `a` и `c` и переписывает их. Ни в одном из преемников B_1 в ГПУ нет φ-функций, поэтому *Rename* рекурсивно переходит к обработке сыновних узлов B_1 в дереве доминирования, B_2, B_3 и B. Наконец, она очищает стеки и возвращает управление.

Блок B_2. Состояние *Rename* при входе в B_2 показано на рис. 9.15(c). В этом блоке нет φ-функций, которые надо было бы переписывать. *Rename* переписывает определения `b`, `c` и `d`, создавая для каждого новое SSA-имя. Затем она переписывает параметры φ-функции в преемнике B_2 в ГПУ, B_3. На рис. 9.15(d) показаны стеки и счетчики непосредственно перед их очисткой. Наконец, *Rename* очищает стеки и возвращает управление.

Block B_3. Состояние *Rename* при входе в B_3 показано на рис. 9.15(e). Заметим, что стеки приведены в состояние, которое имели при входе *Rename* в B_2, но счетчики отражают имена, созданные внутри B_2. В B_3 *Rename* переписывает цели φ-функции, создавая для каждой новые SSA-имена. Затем она переписывает каждое присваивание в блоке, употребляя текущие SSA-имена

Так как `y` и `z` – глобальные имена, алгоритм переименования их не изменяет.

	a	b	c	d	i
Счетчики	1	1	1	1	0
Стеки	a_0	b_0	c_0	d_0	

(a) Начальное состояние, перед B_0

	a	b	c	d	i
Счетчики	1	1	1	1	1
Стеки	a_0	b_0	c_0	d_0	i_0

(b) При входе в B_1

	a	b	c	d	i
Счетчики	3	2	3	2	2
Стеки	a_0	b_0	c_0	d_0	i_0
	a_1	b_1	c_1	d_1	i_1
	a_2		c_2		

(c) При входе в B_2

	a	b	c	d	i
Счетчики	3	3	4	3	2
Стеки	a_0	b_0	c_0	d_0	i_0
	a_1	b_1	c_1	d_1	i_1
	a_2	b_2	c_2	d_2	
			c_3		

(d) В конце B_2

	a	b	c	d	i
Счетчики	3	3	4	3	2
Стеки	a_0	b_0	c_0	d_0	i_0
	a_1	b_1	c_1	d_1	i_1
	a_2		c_2		

(e) При входе в B_3

	a	b	c	d	i
Счетчики	4	4	5	4	3
Стеки	a_0	b_0	c_0	d_0	i_0
	a_1	b_1	c_1	d_1	i_1
	a_2	b_3	c_2	d_3	i_2
	a_3		c_4		

(f) В конце B_3

	a	b	c	d	i
Счетчики	4	4	5	4	3
Стеки	a_0	b_0	c_0	d_0	i_0
	a_1	b_1	c_1	d_1	i_1
	a_2		c_2		

(g) При входе в B_5

	a	b	c	d	i
Счетчики	5	4	5	5	3
Стеки	a_0	b_0	c_0	d_0	i_0
	a_1	b_1	c_1	d_1	i_1
	a_2		c_2	d_4	
	a_4				

(h) В конце B_6

	a	b	c	d	i
Счетчики	5	4	5	6	3
Стеки	a_0	b_0	c_0	d_0	i_0
	a_1	b_1	c_1	d_1	i_1
	a_2		c_2	d_4	
	a_4				

(i) При входе в B_7

	a	b	c	d	i
Счетчики	5	5	6	7	3
Стеки	a_0	b_0	c_0	d_0	i_0
	a_1	b_1	c_1	d_1	i_1
	a_2		c_2	d_4	
	a_4				

(f) В конце B_8

Рис. 9.15 Состояния в примере переименования

для использований глобальных имен и создавая новые SSA-имена для определений глобальных имен.

У B_3 два преемника в ГПУ, B_1 и B_4. В B_1 *Rename* переписывает параметры φ-функции, которые соответствуют ребру из B_3, пользуясь стеками и счетчиками, показанными на рис. 9.15(f). В B_4 φ-функций нет. Далее *Rename* рекурсивно обрабатывает сыновний узел B_3 в дереве доминирования, B_4. Получив обратно управление, *Rename* очищает стеки и возвращается.

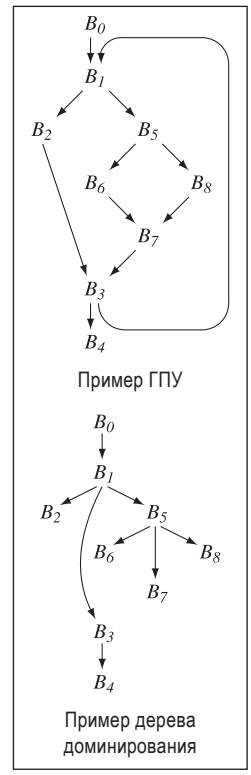
Блок B_4. Этот блок содержит только предложение return. В нем нет φ-функций, определений, использований и преемников – ни в ГПУ, ни в дереве доминирования. Поэтому *Rename* ничего не делает и оставляет стеки и счетчики без изменения.

Блок B_5. После B_4 *Rename* возвращается через B_3 назад в B_1. Со стеками, показанными на рис. 9.15(g), она рекурсивно приступает к обработке последнего потомка B_1 в дереве доминирования, B_5. В B_5 нет φ-функций. *Rename* переписывает оба предложения присваивания, создавая новые SSA-имена по мере необходимости. Ни в одном из преемников B_5 в ГПУ нет φ-функций. Далее *Rename* рекурсивно обрабатывает сыновние узлы B_5 в дереве доминирования, B_6, B_7 и B_8. Наконец, она очищает стеки и возвращается.

Блок B_6. Состояние *Rename* при входе в B_6 показано на рис. 9.15(h). В B_6 нет φ-функций. *Rename* переписывает присваивание d, генерируя новое SSA-имя d_5. Затем она посещает φ-функции в преемнике B_6 в ГПУ, B_7. Она переписывает аргументы φ-функции вдоль ребра из B_6 их текущими именами, c_2 и d_5. Поскольку у B_6 нет потомков в дереве доминирования, *Rename* очищает стек для d и возвращает управление.

Блок B_7. Состояние *Rename* при входе в B_7 показано на рис. 9.15(i). Сначала она переписывает цели φ-функции новыми SSA-именами, c_5 и d_6. Затем переписывает присваивание b новым SSA-именем b_4. После этого она переписывает аргументы φ-функции в преемнике B_7 в ГПУ, B_3, текущими именами. Так как у B_7 нет потомков в дереве доминирования, *Rename* очищает стек для d и возвращает управление.

Блок B_8. Состояние *Rename* при входе в B_9 показано на рис. 9.15(j). В B_8 нет φ-функций. *Rename* переписывает присваивание c новым SSA-именем c_6. Она переписывает соответствующие аргументы φ-функции в B_7 текущими именами, c_6 и d_4. Поскольку у *B8* нет потомков в дереве доминирования, *Rename* очищает стек для d и возвращает управление.



На рис. 9.16 показан код после остановки *Rename*.

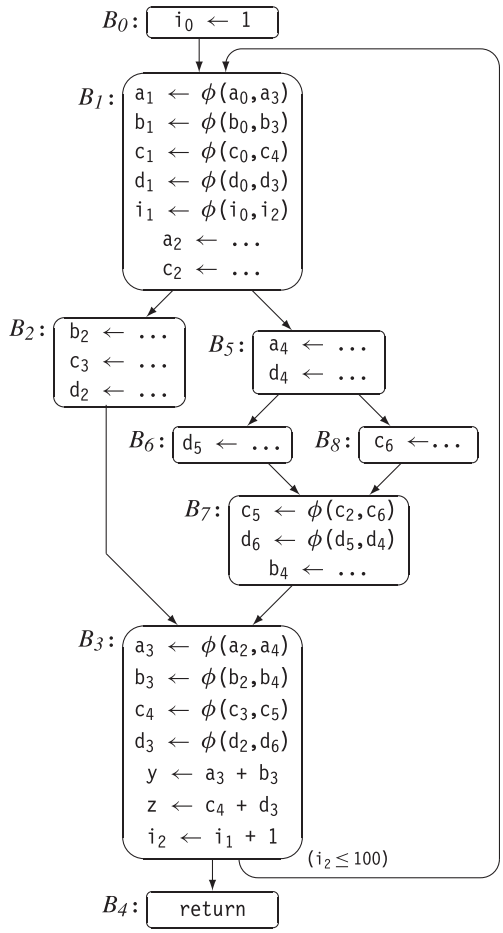


Рис. 9.16 Пример кода после переименования

Последнее улучшение

Благодаря изобретательной реализации *NewName* мы можем уменьшить время и память, необходимые для манипуляций со стеками. Основное назначение стеков – восстанавливать состояние пространства имен по выходе из блока. Если в блоке одно и то же базовое имя переопределено несколько раз, то в стеке должно храниться только самое последнее имя. Например, в блоке B_1 имена a и c определяются дважды. В *NewName* можно было бы повторно использовать слоты для a_1 и c_1 при создании a_2 и c_2.

При таком изменении *Rename* выполняет одно заталкивание и одно выталкивание на каждое базовое имя, определенное в блоке. *NewName* может хранить список созданных ей записей в стеке; по выходе из блока *Rename* может обойти этот список и вытолкнуть записи из нужных стеков. Стеки занимают меньше места; их размер ограничен глубиной дерева доминирования. Манипуляции со стеками упрощаются; алгоритм выполняет меньше операций

заталкивания и выталкивания, и операция заталкивания может не проверять стек на переполнение.

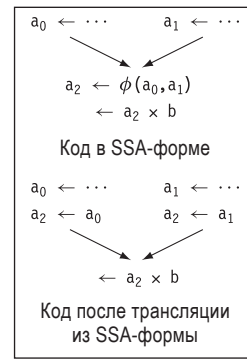
9.3.5. Трансляция программы в форме SSA

Физические процессоры не реализуют φ-функций, поэтому компилятор должен переписывать код без использования φ-функций.

Компилятор, в котором используется форма SSA, должен транслировать эту форму кода в более традиционную модель – без φ-функций, – прежде чем код можно будет выполнить на обычном компьютерном оборудовании. Компилятор должен заменить φ-функции операциями копирования и поместить их в код, воспроизведя семантику φ-функций: выбор значений, диктуемый логикой управления, и параллельное выполнение в начале блока.

В этом разделе рассматривается трансляция из формы SSA. Мы начнем с сильно упрощенной, или наивной, трансляции, лежащей в основе реальных схем. Затем представим два примера задач, на которых продемонстрируем проблемы, возникающие при трансляции из формы SSA в традиционный код. Наконец, будет показан унифицированный каркас, решающий известные проблемы трансляции.

Наивная трансляция



φ-функция – это просто операция копирования, которая выбирает свой вход на основе предшествующего потока управления. Чтобы воспроизвести эффект φ-функции в начале блока b, компилятор может вставить в конец каждого предшественника b в ГПУ операцию копирования, которая перемещает соответствующий аргумент φ-функции в имя, определенное φ-функцией (показано во врезке). Вставив операцию копирования, компилятор может удалить φ-функцию.

Несмотря на концептуальную простоту, в этом процессе есть некоторые сложности. Рассмотрим пример на рис. 9.16. Три блока ГПУ содержат φ-функции: B_1, B_3 и B_7. На рис. 9.17 показан код после вставки операций копирования.

Для B_3 и B_7 вставка в предшествующие блоки работает. У каждого из предшественников B_3 и B_7 имеется по одному преемнику, поэтому операции копирования, вставленные в конец предшественников, не оказывают влияния ни на какой путь, кроме ведущего к φ-функции.

Но для блока B_1 ситуация сложнее. Вставка копирования в конец блока B_0, предшествующего B_1, дает желаемый результат: копирование может иметь место только на пути (B_0, B_1). Что же касается другого предшественника B_1, блока B_3, то простая вставка не годится. Копирование, вставленное в конец B_3, будет выполняться на путях (B_3, B_1) и (B_3, B_4). На пути (B_3, B_4) операция копирования может изменить значение, которое живо в B_4.

Критическое ребро
Ребро графа потока (i, j) называется критическим, если i имеет несколько преемников, а j – несколько предшественников.

Оптимизации, которые перемещают или вставляют код, часто должны разрывать критические ребра.

Ребро (B_3, B_1) демонстрирует более общую проблему размещения кода на *критическом ребре*. У B_3 несколько преемников, поэтому компилятор не может вставить копирование в конец B_3. У B_1 несколько предшественников, поэтому компилятор не может вставить копирование в начало B_1. Поскольку ни то, ни другое решение не работает, компилятор должен разорвать ребро и создать новый блок для вставленных операций копирования. После разрывания ребра и создания блока B_9 оттранслированный код точно воспроизводит эффекты SSA-формы кода.

Проблемы наивной трансляции

Если компилятор применяет наивную трансляцию к коду, полученному сразу после трансляции в форму SSA, то результаты будут правильны, коль скоро критические ребра можно разорвать. Но если компилятор сначала преобразует код, уже имеющий форму SSA, особенно если преобразования перемещают точки определения или использования SSA-имен или если разорвать критические ребра нельзя, то наивная трансляция может порождать некорректный код. На двух примерах мы продемонстрируем, какие ошибки возможны при наивной трансляции.

Проблема потерянного копирования

На рис. 9.17 компилятор был вынужден разорвать ребро (B_3, B_1), создав место для размещения ассоциированных с этим ребром операций копирования. В некоторых ситуациях компилятор не может или не должен разрывать критические ребра. Например, основанный на SSA распределитель регистров не должен добавлять ни блоков, ни ребер в процессе вставки копирований (см. раздел 13.5.3). Комбинация неразорванного критического ребра с оптимизацией, расширяющей интервал жизни некоторого SSA-имени, может создать ситуацию, в которой наивная вставка копирования ведет к ошибке.

На рис. 9.18(a) показан пример, иллюстрирующий проблему. В цикле увеличивается `i`. В вычислении `z` после цикла используется предпоследнее значение `i`. На рис. 9.18(b) показана усеченная SSA-форма для этого кода.

Свертка операций копирования
оптимизация, которая устраняет ненужные операции копирования, назначая источнику и приемнику одно и то же имя, если такое переименование не изменяет поток значений.

Свертка операций копирования называется также *слиянием* (см. раздел 13.4.3).

На рис. 9.18(c) показан код после свертки операций копирования. Использование y_0 при вычислении z_0 было заменено использованием i_1. Последним использованием i_1 на рисунке (b) было присваивание его y_0; свертка копирований расширяет интервал жизни i_1 за пределы цикла на рисунке (c).

Вставка операций копирования в код на рис. 9.18(c) добавляет операцию $i_1 \leftarrow i_0$ в конец блока, предшествующего циклу, и опе-

рацию $i_1 \leftarrow i_2$ в конец цикла. К сожалению, это последнее присваивание уничтожает значение в i_1; теперь вычисление z_0 получает последнее, а не предпоследнее значение i. Вставка копирования порождает некорректный код, потому что расширяет интервал жизни i_1.

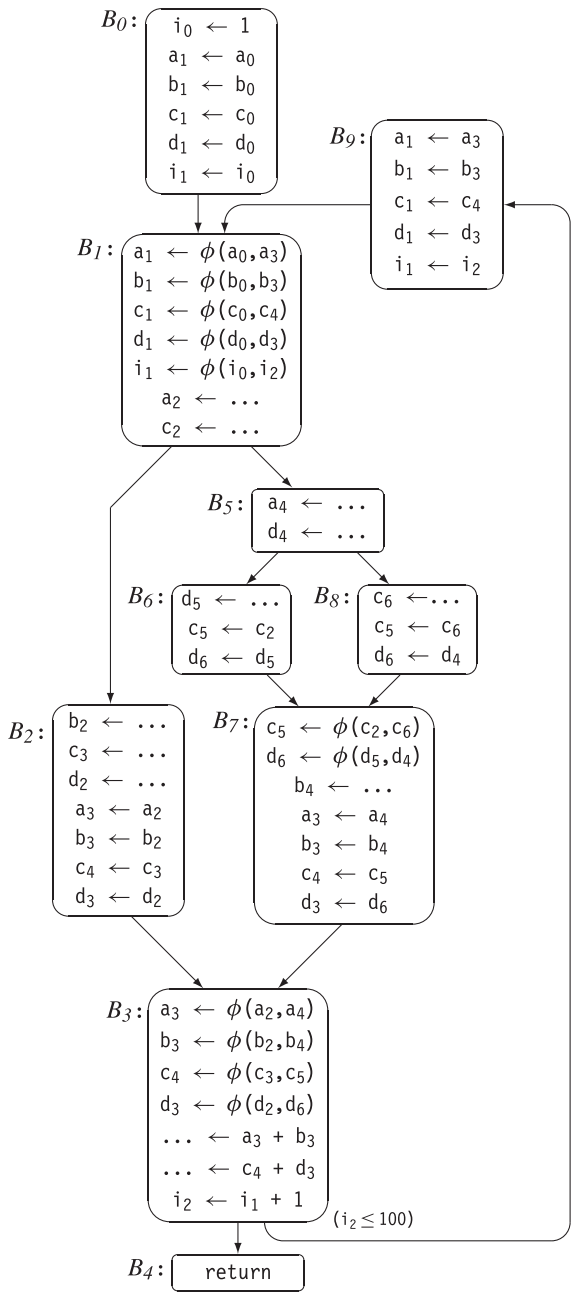


Рис. 9.17 Пример после вставки операций копирования

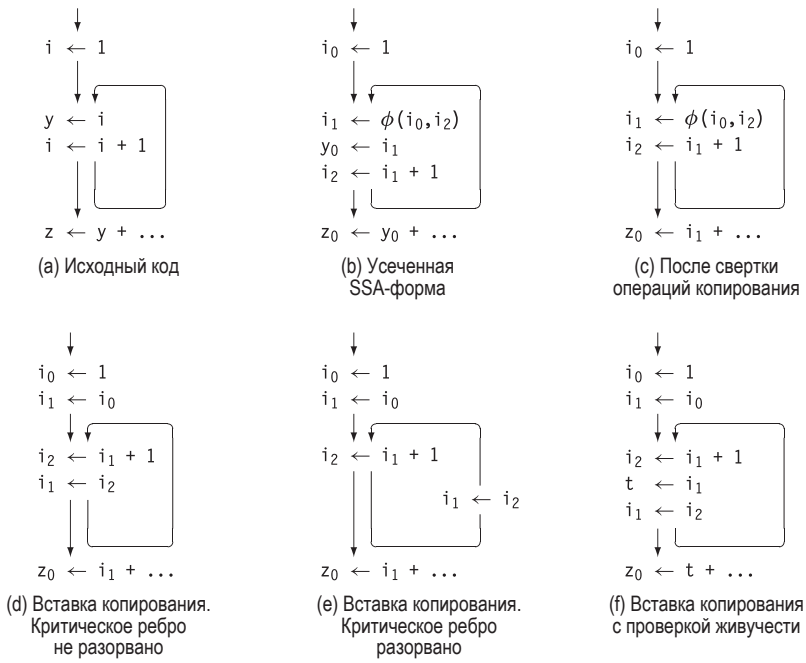


Рис. 9.18 Пример проблемы потерянного копирования

Разрывание критического ребра решает проблему, как показано на рис. 9.18(e); копирование не выполняется на последней итерации цикла. Когда компилятор не может разорвать это ребро, он должен добавить новое имя для сохранения значения i_1, как показано на рис. 9.18(f). Простое ситуативное добавление в процесс вставки операций копирования помогает избежать проблемы потерянного копирования. При вставке копирований компилятор должен проверять, жива ли цель нового копирования в точке вставки. Если цель жива, то компилятор должен ввести новое имя, скопировать в него живое значение и распространить это имя во все места использования после точки вставки.

Проблема обмена

Конкурентная семантика φ-функций создает еще одну проблему для трансляции из формы SSA, которую мы называем проблемой обмена (swap problem). Поясняющий пример показан на рис. 9.19(a): простой цикл, в котором обмениваются значения x и y. Если компилятор строит усеченную SSA-форму, как на рис. 9.19(b), и выполняет свертку операций копирования, как на рис. 9.19(c), то получается корректная программа в форме SSA, которая опирается непосредственно на конкурентную семантику φ-функций в одном блоке.

Поскольку обе φ-функции читают свои значения конкурентно, а затем конкурентно же записывают результаты, код на рисунке (c) имеет такую же семантику, как исходный код на рисунке (a). Однако наивная вставка копирований заменяет каждую φ-функцию операцией последовательного копиро-

вания, как показано на рисунке (d). Эти два последовательных копирования дают результат, отличный от двух φ-функций; подстановка фундаментально изменяет семантику кода.

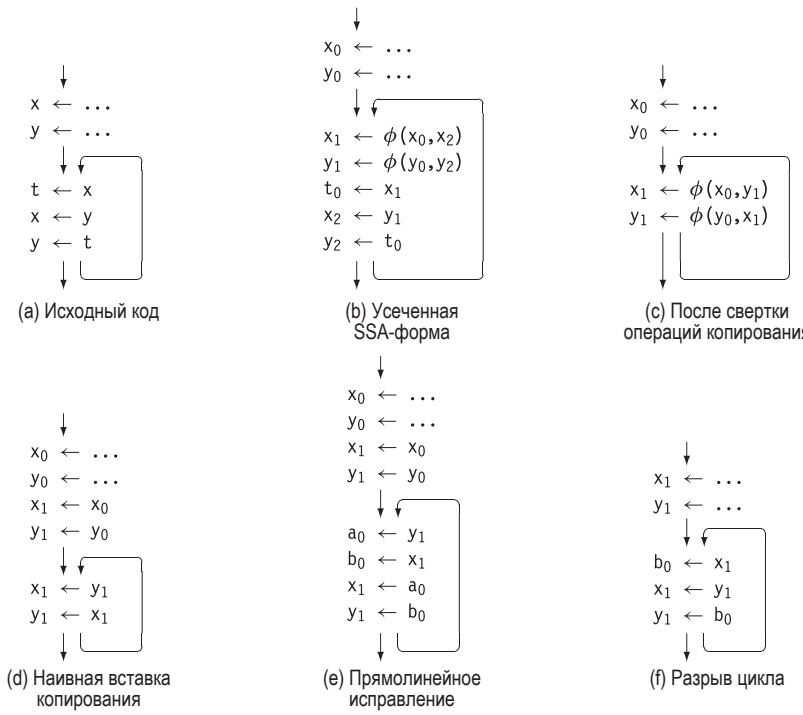


Рис. 9.19 Пример проблемы обмена

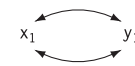
Чтобы сохранить семантику исходного кода, компилятор должен гарантировать, что вставленные операции копирования точно воспроизводят поток значений, определяемый φ-функциями. Следовательно, он должен обращать внимание на все значения, которые определены одной φ-функцией, а используются другой φ-функцией в том же блоке.

В некоторых случаях компилятор должен вводить одно или несколько новых имен. Прямолинейное решение этой проблемы – принять двухэтапный протокол копирования, показанный на рис. 9.19(e). На первом этапе аргументы каждой φ-функции копируются в собственные временные имена, моделируя тем самым поведение φ-функции в плане выбора и параллельного чтения аргументов, диктуемых потоком управления. На втором этапе эти значения копируются в цели φ-функции.

К сожалению, это решение удваивает число операций копирования, необходимых для трансляции из формы SSA. Компилятор может уменьшить количество временных имен и дополнительных операций копирования, если построит небольшой граф зависимостей для множества параллельных копирований, индуцированных φ-функциями, и воспользуется им для управления вставкой операций последовательного копирования. Если граф за-

висимостей ациклический, то компилятор может его использовать, чтобы спланировать такое выполнение операций копирования, при котором не требуются дополнительные имена или операции (см. главу 12).

Если граф зависимостей содержит циклы, то компилятор должен разорвать каждый цикл копированием в имя, не участвующее в цикле. Для этого может понадобиться новое имя. Граф зависимостей для нашего примера, показанный справа, содержит цикл из двух вершин. Для разрыва этого цикла нужно одно новое имя, и в результате порождается код, показанный на рисунке (f).



Унифицированный подход к трансляции программы в форме SSA

Причинами проблем обмена и потерянного копирования являются два разных явления: преобразования, которые изменяет интервал, в котором живо SSA-имя, и невозможность сохранить параллельную семантику φ-функции при трансляции из SSA-формы. Стандартные преобразования кода, такие как свертка операций копирования, перемещение кода и межблочное планирование команд, могут создавать обстоятельства, ведущие к этим проблемам. Хотя решения, предложенные в предыдущем разделе, позволяют сгенерировать корректный код, ни одно из них не дает хорошего каркаса для понимания корневых причин проблемы.

В унифицированном подходе используется трехэтапный план решения обеих проблем, вызванных применением преобразований кода в форме SSA: изменения интервала жизни SSA-имен и неявного использования параллельной семантики выполнения φ-функции. На первом этапе вводится новое множество имен, чтобы изолировать φ-функции от остального кода; затем вставляются операции параллельного копирования, чтобы связать эти имена с окружающим контекстом. На втором этапе φ-функции заменяются операциями параллельного копирования в предшествующих блоках. На третьем этапе каждый блок параллельных копирований переписывается эквивалентным рядом последовательных копирований. Этот процесс позволяет избежать как проблемы обмена, так и проблемы потерянного копирования. И кроме того, устраняет необходимость разрывать критические ребра.

Этап 1

Мы будем обозначать группу операций параллельного копирования, добавляя общий индекс к оператору присваивания, $\leftarrow_i$.

Чтобы изолировать пространство имен φ-функции, например $a_0 \leftarrow \varphi(a_1, a_2, \ldots, a_n)$, на первом этапе оно переписывается в виде $a'_0 \leftarrow \varphi(a'_1, a'_2, \ldots, a'_n)$. Чтобы связать новые имена со штрихом с окружающим кодом, компилятор добавляет операцию копирования $a'_i \leftarrow a_i$ в конец предшествующего блока, ассоциированного с a_i, для каждого параметра a_i. Чтобы сохранить параллельную семантику выполнения

φ-функций, компилятор будет использовать группы операций параллельного копирования для тех копирований, которые вставляет.

После группы φ-функций в начале блока компилятор должен вставить еще одну группу параллельных операций копирования. Для каждой φ-функции в блоке $a_i \leftarrow \varphi(\ldots)$ группа должна включать операцию копирования вида $a_i \leftarrow a'_i$. Общий эффект этих трех действий заключается в том, чтобы изолировать имена, используемые в φ-функциях, от окружающего кода и сделать последствия параллельного выполнения явными, вынесенными за пределы φ-функций.

На рис. 9.20 изображены результаты этого преобразования для примера, демонстрирующего проблему обмена. На рисунке (a) показан исходный код, а на рисунке (b) – его усеченная SSA-форма со свернутыми операциями копирования. На рисунке (c) показан код, получившийся после того, как компилятор изолировал φ-функции. Параметры φ-функций переименованы, и вставлены группы параллельных операций копирования.

- Группа 1 параллельных операций копирования в начале первого блока присваивает начальные значения x'_0 и y'_0.
- Группа 2 параллельных операций копирования в конце тела цикла присваивает значения x'_1 и y'_1, получившиеся в результате вычисления внутри цикла. (Тело цикла является своим собственным предшественником.)
- Группа 2 параллельных операций копирования после φ-функций копирует значения, определенные φ-функциями, в имена, которые у них были до преобразования переименования.

В этой точке компилятор может переименовать все переменные со штрихом и опустить индексы в SSA-именах, как показано на рисунке (d). Код после переименования сохраняет семантику исходного кода.

Этап 2

На этом этапе φ-функции заменяются путем вставки операций копирования в предшествующие блоки и удаления самих φ-функций. Чтобы сохранить семантику φ-функций, компилятор использует операции параллельного копирования в каждом блоке.

В конце первого этапа, как показано на рисунке (d), происходит фактический обмен значений в процессе вычисления аргументов φ-функций. После замены φ-функции, показанной на рисунке (e), этот обмен значений происходит в группе 5 параллельных операций копирования в конце тела цикла.

В конце второго этапа компилятор устранил все φ-функции. Код все еще содержит группы параллельных операций копирования, реализующие семантику φ-функций. Чтобы завершить процесс, компилятор должен переписать каждую группу параллельных операций копирования набором последовательных копирований. Код, вероятно, будет содержать несколько (возможно, много) ненужных операций копирования. Слияние может устранить некоторые из них или даже все (см. раздел 13.4.3).

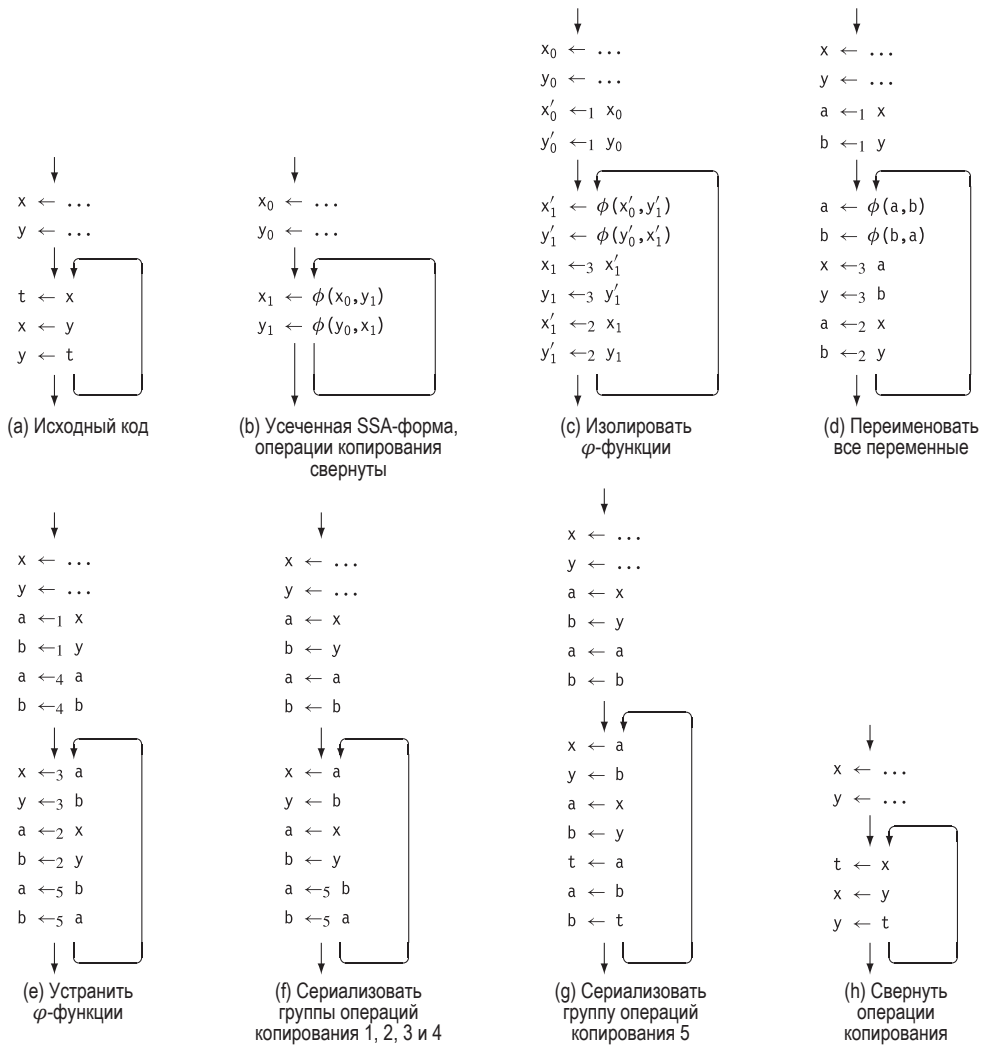
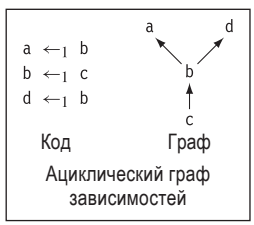
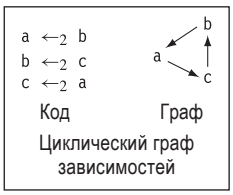


Рис. 9.20 Унифицированный подход к трансляции из SSA-формы

Этап 3



На последнем этапе каждая группа параллельных операций копирования исследуется и переписывается эквивалентной группой последовательных операций копирования. Для группы операций строится граф зависимостей по данным (см. раздел 4.3.2). Если этот граф ациклический, как граф во врезке, то компилятор может просто вставить операции копирования в порядке, индуцированном графом, – от листьев к корням. В примере во врезке граф требует, чтобы операции a ← b и d ← b предшествовали b ← c.



Циклический граф зависимостей

Если граф зависимостей содержит цикл, как показано в примере во врезке, то компилятор должен вставлять операции копирования способом, разрывающим цикл. В данном случае он должен скопировать одно из значений, скажем a, в новое временное имя, скажем t. Затем он может выполнить операции a ← b и b ← c. И завершить группу операцией c ← t. Это разрывает цикл и корректно реализует семантику группы параллельных операций копирования.

В некоторых случаях компилятор может избежать создания нового имени с помощью продуманного упорядочения. Так, если бы второй пример включал также операцию d $\leftarrow_2$ a, то компилятор мог бы спланировать первой d ← a и разорвать цикл, переписав c ← a в виде c ← d.

В нашем примере группы 1, 2, 3 и 4 можно сериализовать без дополнительных имен, как показано на рис. 9.20(f). Группа 5 операций копирования содержит цикл, поэтому требует одного нового имени, t. На рисунке (g) показан результат перезаписи группы 5. На рисунке (h) представлен окончательный код после свертки операций копирования.

9.3.6. Использование формы SSA

Автор компилятора использует форму SSA, потому что она улучшает качество анализа, качество оптимизации или то и другое вместе. Чтобы разобраться, чем анализ формы SSA отличается от классических методов анализа потока данных, представленных в разделе 9.2, рассмотрим задачу о глобальном распространении констант в форме SSA и воспользуемся алгоритмом *разреженного простого распространения констант* (sparse simple constant propagation – SSCP).

Полурешетка
множество L и оператор *встречи* $\wedge$ такие, что $\forall\ a, b, c \in L$,

1) $a \wedge a = a$
2) $a \wedge b = b \wedge a$ и
3) $a \wedge (b \wedge c) = (a \wedge b) \wedge c$.

Компиляторы используют полурешетки для моделирования областей определения данных в задачах анализа.

В SSCP компилятор аннотирует каждое SSA-имя значением. Множество возможных значений образует *полурешетку*. Полурешетка состоит из множества значений L и оператора встречи $\wedge$. Оператор встречи должен быть идемпотентным, коммутативным и ассоциативным, он индуцирует порядок на элементах L:

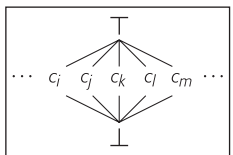
$a \geqslant b$ тогда и только тогда, когда $a \wedge b = b$, и
$a > b$ тогда и только тогда, когда $a \geqslant b$ и $a \neq b$.

В каждой полурешетке существует нижний элемент, $\bot$, такой, что

$\forall\ a \in L\ a \wedge \bot = \bot$ и $\forall\ a \in L\ a \geqslant \bot$.

В некоторых полурешетках существует также верхний элемент, $\top$, такой, что

$\forall\ a \in L\ a \wedge \top = a$ и $\forall\ a \in L\ \top \geqslant a$.



Полурешетка для распространения констант

При распространении констант структура полурешетки, используемая для моделирования значений в программе, играет критически важную роль для сложности алгоритма. Полурешетка для одного SSA-имени показана во врезке. Она состоит из элементов ⊤, ⊥ и бесконечного множества различных постоянных значений. Для любого значения x имеют место равенства $x \wedge \top = x$ и $x \wedge \bot = \bot$. Для двух констант, c_i и c_j: $c_i \wedge c_j = \bot$, если $c_i \neq c_j$. Если $c_i = c_j$, то $c_i \wedge c_j = c_i$.

Алгоритм SSCP, показанный на рис. 9.21, состоит из фазы инициализации и фазы распространения. В фазе инициализации перебираются SSA-имена. Для каждого SSA-имени n алгоритм исследует операцию, в которой определяется n, и устанавливает *Value*(n), применяя набор простых правил:

1) если n определено φ-функцией, то SSCP полагает *Value*(n) равным ⊤;
2) если значение n неизвестно, то SSCP полагает *Value*(n) равным ⊤;
3) если значением n является известная константа c_i, то SSCP полагает *Value*(n) равным c_i;
4) если значение n *узнать невозможно* – например, если оно определяется в результате чтения из внешнего источника, – то SSCP полагает *Value*(n) равным ⊥.

Если *Value*(n) не равно ⊤, то алгоритм добавляет n в рабочий список.

```
// Фаза инициализации
WorkList ← ∅
for каждого SSA-имени n do
  инициализировать Value(n) по правилам, описанным в тексте
  if Value(n) ≠ ⊤ then
    WorkList ← WorkList ∪ {n}

// Фаза распространения – продолжать итерации до нахождения неподвижной точки
while (WorkList ≠ ∅) do
  удалить какое-то n из WorkList   // выбрать произвольное имя
  for каждой операции op, в которой используется n, do
    положить m равным SSA-имени, которое определяет op
    if Value(m) ≠ ⊥ then              // пересчитать и проверить, изменилось ли
      t ← Value(m)
      Value(m) ← результат интерпретации op на элементах решетки
      if Value(m) ≠ t then
        WorkList ← WorkList ∪ {m}
```

Рис. 9.21 Алгоритм разреженного простого распространения констант

Эти инициализации проливают свет на использование ⊤ и ⊥ в полурешетке распространения констант. ⊤ означает, что компилятор еще ничего не знает о значении, но может получить информацию о значении в будущем. Напротив, ⊥ означает, что компилятор доказал, что значение не является константой. Для любого SSA-имени m *Value*(m) может измениться не более двух

раз – это высота полурешетки. Если *Value*(m) вначале равно $\top$, то оно может впоследствии стать равным какой-то константе, c_i или $\bot$. Если *Value*(m) равно некоторой константе c_i, то оно может стать равным $\bot$. После того как *Value*(m) стало равно $\bot$, оно уже не может измениться.

Этап распространения не вызывает трудностей. SSA-имя n удаляется из рабочего списка. Алгоритм исследует каждую операцию *op*, в которой n используется, где *op* определяет какое-то SSA-имя m. Если *Value*(m) уже достигла $\bot$, то больше никаких вычислений не требуется. В противном случае алгоритм моделирует вычисление *op*, интерпретируя операцию на решеточных значениях ее операндов. Если результат в решетке меньше *Value*(m), то он соответственно уменьшает *Value*(m) и добавляет m в рабочий список. Алгоритм останавливается, когда рабочий список становится пуст.

Интерпретация операции над решеточными значениями требует аккуратности. Для φ-функции результат просто равен встрече решеточных значений всех аргументов φ-функции; правила встречи показаны на полях в порядке предшествования. Для других видов операций компилятору необходим набор правил. Рассмотрим, к примеру, $a \times b$. Если $a = 4$ и $b = 17$, то $a \times b = 68$. Однако если $a = \bot$, то $a \times b = \bot$, если только b не равно 0. ($a \times 0 = 0$ для любого a.)

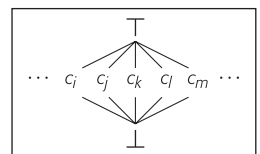
$\top \wedge x = x$	$\forall\, x$
$\bot \wedge x = \bot$	$\forall\, x$
$c_i \wedge c_j = c_i$,	если $c_i = c_j$
$c_i \wedge c_j = \bot$,	если $c_i \neq c_j$

Правила встречи

В общем случае правила вычисления операторов должны сохранять $\top$, если только другой операнд не вынуждает выбрать другое значение, как в случае умножения на ноль. Если $a = \top$, то вопрос о том, равна ли сумма $a + b$ значению $\top$, откладывается до момента, когда станет известно значение b – константа или $\bot$.

Сложность

Фаза распространения SSCP – классический пример схемы с неподвижной точкой. Доказательство завершения и оценка сложности следуют из длины убывающих цепочек на полурешетке, еще раз показанной во врезке. Решеточное значение SSA-имени может измениться не более двух раз: $\top$ в c_i и c_i в $\bot$.



Полурешетка для распространения констант

SSCP добавляет SSA-имя в рабочий список, только когда его значение изменяется, поэтому каждое имя встречается в рабочем списке не более двух раз. SSCP вычисляет операцию, когда один из ее операндов удаляется из рабочего списка, что ограничивает сверху количество операций удвоенным числом мест использования в коде.

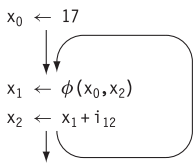
Оптимизм: роль $\top$

Как отмечалось выше, SSCP использует решеточное значение $\top$, чтобы представить отсутствие знаний. Эта практика отличается от задач о потоке данных в разделе 9.2, где используется значение $\bot$, а не $\top$. Использование $\top$

в качестве начального значения играет критическую роль в распространении констант; оно позволяет константам распространяться в циклы графа.

Поскольку неизвестные значения инициализируются $\top$, а не $\bot$, алгоритм может распространять значения в циклы графа – петли в случае ГПУ. Алгоритмы, которые начинают работу со значения $\top$, а не $\bot$, часто называют оптимистическими. Слово *оптимизм* здесь интуитивно означает, что инициализация значением $\top$ позволяет алгоритму распространять информацию в циклический регион, оптимистично предполагая, что значение вдоль обратного ребра подтвердит это начальное распространение. Инициализация значением $\bot$, называемая *пессимистической*, запрещает такую возможность.

Рассмотрим фрагмент формы SSA на рис. 9.22. Если алгоритм инициализирует x_1 и x_2 значением $\bot$ (пессимистично), то он не распространит значение 17 в петлю. При вычислении φ-функции он положит x_1 равным $17 \wedge \bot = \bot$. Когда $x_1 = \bot$, алгоритм распространения полагает $x_2 = \bot$ независимо от значения i_{12}.



(a) Фрагмент кода

Временной шаг	Решеточные значения: Пессимистичный x_0	x_1	x_2	Оптимистичный x_0	x_1	x_2
0	17	$\bot$	$\bot$	17	$\top$	$\top$
1	17	$\bot$	$\bot$	17	17	$17 + i_{12}$

(b) Результаты пессимистичного и оптимистичного анализов

Рис. 9.22 Пример оптимистичного распространения констант

С другой стороны, если алгоритм инициализирует $x_1 = \top$ и $x_2 = \top$ (оптимистично), то может распространить значение x_0 в цикл. Он вычисляет значение x_1 как $17 \wedge \top = 17$. Поскольку значение x_1 изменилось, алгоритм помещает x_1 в *WorkList*. Затем алгоритм пересчитывает определение x_2. Если, например, $i_{12} = 0$, то x_2 получает значение 17, и алгоритм добавляет x_2 в рабочий список. Пересчитывая φ-функцию, он полагает $x_1 = 17 \wedge 17 = 17$.

Рассмотрим, что произошло бы в случае $i_{12} = 2$. Тогда при вычислении $x_1 + i_{12}$ SSCP положил бы $x_2 = 19$. Затем он пересчитывает $x_1 = 17 \wedge 19 = \bot$. Это значение $\bot$, в свою очередь, распространяется в x_2, доказывая, что x не является константой в цикле.

Ценность формы SSA

Применение формы SSA в SSCP приводит к простому и эффективному алгоритму. Чтобы убедиться в этом, рассмотрим классический подход к задаче о потоке данных. Мы создали бы в начале каждого блока множество ConstantsIn, а в конце каждого блока – множество ConstantsOut. Эти множества содержали бы пары ⟨*переменная*, *значение*⟩.

Для блока b компилятор мог бы вычислить ConstantsIn(b) как попарное пересечение ConstantsOut(p) по всем $p \in preds(b)$. Все значения для одного имени нужно было бы пересечь, используя тот же оператор пересечения, что в SSCP. Чтобы вывести ConstantsOut(b) из ConstantsIn(b), компилятор мог

бы применить версию ЛНЗ, дополненную обработкой $\perp$ и $\top$. Итеративный алгоритм поиска неподвижной точки остановился бы, когда множества прекратили изменяться.

Это сильно упрощенное изложение алгоритма. В такой постановке отсутствует единственная неподвижная точка, поэтому результаты зависят от порядка обработки блоков. Кроме того, отсутствуют свойства, обеспечивающие быструю сходимость итеративного алгоритма. Время работы решателя может составлять $O(n^2)$, а то и больше.

С другой стороны, SSCP – простой итеративный алгоритм поиска неподвижной точки, работающий на разреженном графе и совсем неглубокой решетке. У него возникают те же сложности при интерпретации каждой операции для известных постоянных значений, но он интерпретирует одиночные операции, а не целые блоки. Его временная сложность понятна. В данном случае использование формы SSA приводит к простому, эффективному, разреженному методу распространения глобальных констант.

ИТОГИ РАЗДЕЛА

Форма SSA кодирует информацию о потоках данных и управления в концептуально простом промежуточном представлении. В этом разделе мы рассмотрели алгоритмы трансляции кода в полуусеченную SSA-форму и из нее. Процесс начального построения формы состоит из двух шагов. На первом шаге вставляются φ-функции в точки соединения, где могут сходиться различные определения. Для обеспечения эффективности этот алгоритм пользуется границами доминирования. На втором шаге создается пространство SSA-имен путем добавления индексов к исходным базовым именам в процессе систематического обхода всей процедуры.

Поскольку физические процессоры не реализуют φ-функции непосредственно, компилятор должен транслировать код из формы SSA, прежде чем его можно будет выполнить. Преобразование кода, представленного в форме SSA, может затруднить эту трансляцию. В разделе 9.3.5 рассмотрены проблемы «потерянного копирования» и «обмена» и описаны подходы к их решению. Наконец, в разделе 9.3.6 показан алгоритм распространения глобальных констант по форме SSA.

ВОПРОСЫ

1. Максимальная SSA-форма включает бесполезные φ-функции, которые определяют неживые значения, и избыточные φ-функции, которые объединяют одинаковые значения (например, $x_8 \leftarrow \varphi(x_7, x_7)$). Возможна ли вставка неживых или избыточных φ-функций в полуусеченной SSA-форме? Если да, то как компилятор может устранить их?
2. Предположим, что ваш компилятор ориентирован на ISA, в которой реализована команда `swap` r_1, r_2, одновременно выполняющая присваивания $r_1 \leftarrow r_2$ и $r_2 \leftarrow r_1$. Какое влияние команда `swap` могла бы оказать на трансляцию из SSA-формы?

9.4. Межпроцедурный анализ

Вызовы процедур приводят к неэффективностям двух видов: (1) потере знаний при анализе и оптимизации одной процедуры из-за наличия места вызова; (2) накладным расходам на реализацию абстракций, неотъемлемым от вызовов процедур. Межпроцедурный анализ был предложен для реше-

ния первой проблемы. В разделе 9.2.4 мы видели, что компилятор может вычислить множества, обобщающие побочные эффекты каждого места вызова. В этом разделе мы изучим более сложные вопросы межпроцедурного анализа.

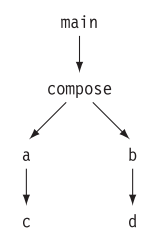
9.4.1. Построение графа вызовов

Первая проблема, которую компилятор должен решить в процессе межпроцедурного анализа, – построение графа вызовов. В простейшем случае, когда при каждом вызове вызывается процедура с фиксированным литеральным именем, например `call fee(x, y, z)`, проблема решается в лоб. Компилятор создает вершину графа вызовов для каждой процедуры в программе и добавляет в этот граф ребро для каждого места вызова. Процесс занимает время, пропорциональное числу процедур и числу мест вызова; на практике лимитирующим фактором является стоимость нахождения мест вызова.

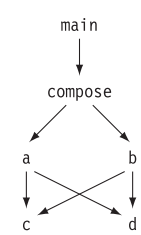
Особенности исходного языка могут затруднить построение графа вызовов. Например, рассмотрим коротенькую программу на C, показанную на рис. 9.23(a). Ее точный граф вызовов представлен на рис. 9.23(b). В следующих подразделах описаны особенности языка, осложняющие построение графа вызовов.

```
int compose( int f(), int g() ) {
  return f(g);
}
int a( int z() ) {
  return z();
}
int b( int z() ) {
  return z();
}
int c( ) {
  return 0;
}
int d( ) {
  return 1;
}
int main( int argc, char *argv[] ) {
  printf("(a,c) returns %d.\n",compose(a,c));
  printf("(b,d) returns %d,\n",compose(b,d));
}
```

(a) Пример программы на C



(b) Точный граф вызовов



(c) Приближенный граф вызовов

Рис. 9.23 Построение графа вызовов в случае, когда параметрами являются функции

Переменные, принимающие процедуры в качестве значений

Если в программе используются переменные, принимающие процедуры в качестве значений, то компилятор должен либо предполагать, что с помощью такой переменной можно вызвать любую процедуру, либо проанализировать программу и оценить множество процедур, которые могут быть вызваны из каждого места. Для выполнения такого анализа компилятор может построить граф по вызовам, в которых используются явные фиксированные имена процедур. Затем он может проследить распространение функций как значений по этому подмножеству графа вызовов, добавляя ребра по мере необходимости.

Компилятор может использовать простой аналог распространения глобальных констант для переноса значений-функций от входа в процедуру до мест вызова, где они используются, применяя в качестве операции встречи теоретико-множественное объединение.

> В SSCP следует инициализировать все формальные параметры, принимающие значения-функции, известными постоянными значениями. Фактические параметры с известными значениями покажут, как передаются функции.

Построив множество процедур, которые могут быть переданы параметру-функции, компилятор должен смоделировать передачу этого параметра в каждом месте вызова внутри процедуры. Большинство языков программирования не допускают операций со значением-процедурой, поэтому такое моделирование можно организовать просто и эффективно (см. обсуждение скачковых функций в разделе 9.4.2).

На рис. 9.23 показан бесхитростный анализ, который может построить завышенную оценку множества ребер графа вызовов. Код вызывает функцию `compose` для вычисления `a(c)` и `b(d)`. Однако простой анализ приведет к выводу, что формальный параметр `g` функции `compose` может принимать значение `c` или `d`, и, значит, в результате в программе может появиться любая из композиций `a(c)`, `a(d)`, `b(c)` и `b(d)`, как показано на рисунке (c). Чтобы построить точный граф вызовов, показанный на рисунке (b), компилятор должен проследить, какие наборы параметров передаются вместе, по одному и тому же пути. Затем компилятор мог бы рассмотреть каждый набор независимо и вывести точный граф. Альтернативно он мог бы пометить каждое значение путем, по которому это значение проходит, и использовать информацию о путях, чтобы избежать добавления посторонних ребер, например `(a,d)` или `(b,c)`.

Контекстно разрешаемые имена

Некоторые языки позволяют использовать имена, которые разрешаются в зависимости от контекста. В объектно-ориентированных языках с иерархией наследования привязка имени метода к конкретной реализации зависит от класса получателя и от состояния иерархии наследования.

Если иерархия наследования и все процедуры фиксированы в момент анализа, то компилятор может использовать межпроцедурный анализ структуры классов, чтобы сузить множество методов, которые могут быть вызваны в любом заданном месте. Построитель графа вызовов должен включать ребро из этого места вызова к каждой процедуре или методу, которые могут быть вызваны.

Динамическая компоновка, применяемая в некоторых операционных системах для уменьшения требований к виртуальной памяти, привносит аналогичные сложности. Если компилятор не может определить, какой код будет выполняться, то он не сможет построить полный граф вызовов.

Для языка, который позволяет импортировать в программу исполняемый код или новые определения классов во время выполнения, компилятор должен построить консервативный граф вызовов, который отражает полное множество потенциальных вызываемых сущностей в каждом месте вызова. Один из вариантов – поручить компилятору построить по одной вершине графа вызовов для представления этих неизвестных процедур и наделить эту вершину поведением в худшем случае, таком как максимальные множества MayMod и MayRef. Эта стратегия гарантирует, что у других видов анализа будут консервативные аппроксимации множества возможных фактов.

Анализ с целью разрешения неоднозначных вызовов может повысить точность графа вызовов за счет уменьшения числа посторонних ребер – для вызовов, которые вообще не могут иметь места во время выполнения. Такую же или даже большую важность имеет тот факт, что любое место вызова, которое разрешается единственным образом, может быть реализовано с помощью прямого вызова; если же разрешение неоднозначно, то может понадобиться поиск во время выполнения для диспетчеризации вызова (см. раздел 6.3.2). Поиск во время выполнения для поддержки динамической диспетчеризации может оказаться гораздо дороже прямого вызова.

Другие языково-зависимые проблемы

При внутрипроцедурном анализе мы предполагаем, что граф потока управления имеет одну входную и одну выходную вершины; если у процедуры несколько точек выхода, то мы добавляем в граф искусственную выходную вершину. Аналогичные проблемы возникают в межпроцедурном анализе.

Например, в JAVA есть инициализаторы и финализаторы. Виртуальная машина JAVA вызывает инициализатор класса после его загрузки и проверки, а инициализатор объекта – после выделения для него памяти, но перед возвратом хеш-кода объекта. Методы запуска потока, финализаторы и деструкторы также обладают тем свойством, что выполняются без явного вызова со стороны программы.

Построитель графа вызовов должен распознавать и понимать такие процедуры. Он должен правильно включать их в граф вызовов. Конкретные детали зависят от языка и вида анализа. Например, анализ MayMod мог бы

игнорировать их как нерелевантные, тогда как анализ межпроцедурного распространения констант может нуждаться в информации от методов инициализации и запуска.

9.4.2. Межпроцедурное распространение констант

Анализ межпроцедурного распространения констант прослеживает распространение постоянных значений глобальных переменных и параметров по графу вызовов как внутри тел процедур, так и по ребрам графа. Цель этого анализа – найти места, где процедура всегда получает или возвращает известное постоянное значение. Найдя такую константу, компилятор может специализировать для нее код.

Поначалу ограничимся нахождением формальных параметров с постоянными значениями. А в конце раздела обобщим анализ на глобальные переменные.

Концептуально межпроцедурное распространение констант состоит из трех подпроблем: найти начальное множество констант, распространить известные значения констант по графу вызовов и смоделировать прохождение значений через процедуры.

Нахождение начального множества констант

Анализатор должен в каждом месте вызова определить, какие фактические параметры имеют известные постоянные значения. Для этого имеется широкий круг методов. Самый простой – распознать литеральные константы, используемые в качестве параметров. Более эффективный и дорогостоящий подход мог бы использовать распространение глобальных констант (например, алгоритм SSCP из раздела 9.3.6) для идентификации параметров с постоянными значениями.

Распространение известных постоянных значений по графу вызовов

Зная начальное множество констант, анализатор распространяет постоянные значения по ребрам графа вызовов и внутри процедур от точки входа к каждому месту вызова. Эта часть анализа напоминает итеративные алгоритмы потока данных, описанные в разделе 9.2. Итеративный алгоритм решит эту проблему, но может потребовать значительно больше итераций, чем понадобилось для более простых задач, таких как живучесть переменных или доступные выражения.

Моделирование передачи значений через процедуры

Всякий раз, как анализатор обрабатывает вершину графа вызовов, он должен определить, как постоянные значения, известные при входе в процедуру,

влияют на множество постоянных значений, известных в каждом месте вызова внутри процедуры. Для этого он строит небольшую модель для каждого фактического параметра, называемую *скачковой функцией*. Для места вызова s будем обозначать $\mathcal{J}_s^a$ скачковую функцию для параметра a.

Каждое место вызова s представлено вектором скачковых функций. Если s имеет n параметров, то алгоритм строит вектор $\mathcal{J}_s = \langle \mathcal{J}_s^a, \mathcal{J}_s^b, \mathcal{J}_s^c, \ldots, \mathcal{J}_s^n \rangle$, где a – первый формальный параметр вызываемой стороны, b – второй и т. д. Каждая скачковая функция $\mathcal{J}_s^x$ зависит от значений некоторого подмножества глобальных переменных и формальных параметров процедуры p, содержащей s; мы будем обозначать это множество $Support(\mathcal{J}_s^x)$.

Временно предположим, что $\mathcal{J}_s^x$ состоит из дерева выражения, листьями которого являются все глобальные переменные, формальные параметры вызывающей стороны или литеральные константы. Мы требуем, чтобы $\mathcal{J}_s^x$ возвращала $\top$, если $Value(y)$ равно $\top$ для любого $y \in Support(\mathcal{J}_s^x)$.

Алгоритм

На рис. 9.24 показан алгоритм межпроцедурного распространения констант. Он похож на алгоритм SSCP из раздела 9.3.6.

Алгоритм ассоциирует поле $Value(x)$ с каждым формальным параметром x каждой процедуры p. (Предполагается, что имена формальных параметров уникальны или полностью квалифицированы.) На первом этапе все поля *Value* оптимистически полагаются равными $\top$. Затем алгоритм перебирает все фактические параметры a в каждом месте вызова s в программе, присваивает полю *Value* формального параметра f, соответствующего a, новое значение $Value(f) \wedge \mathcal{J}_s^f$ и добавляет f в рабочий список. На этом шаге начальное множество констант, представленное скачковыми функциями, разбирается по полям *Value*, а в рабочий список помещаются все формальные параметры.

На втором этапе из рабочего списка в цикле выбирается и распространяется формальный параметр. Чтобы распространить формальный параметр f процедуры p, анализатор находит каждое место вызова s в p и каждый формальный параметр x (который соответствует фактическому параметру в месте вызова s) такой, что $f \in Support(\mathcal{J}_s^x)$. Он вычисляет $\mathcal{J}_s^x$ и объединяет его с $Value(x)$. Если $Value(x)$ изменилось, то x добавляется в рабочий список. Рабочий список должен быть реализован структурой данных, например разреженным множеством, которая не допускает дубликатов (см. раздел B.2.3).

Этот алгоритм опирается на то же доказательство остановки, основанное на полурешетке, которое использовалось для SSCP в разделе 9.3.6.

Второй этап завершается, потому что каждое множество *Value* может принимать не более трех значений на полурешетке: $\top$, некоторое c_i и $\bot$. Переменная x может попасть в рабочий список, только когда ее начальное *Value* вычислено или изменилось. Каждая переменная x может встретиться в рабочем списке не более трех раз. Таким образом, полное число изменений ограничено, и итерации останав-

ливаются. После завершения второго этапа на этапе постобработки строится множество констант, известных при входе в каждую процедуру.

```
// Этап 1: инициализация
Построить все скачковые функции и отображения Support
Worklist ← ∅
for каждой процедуры p в программе do
  for каждого формального параметра f процедуры p do
    Value(f) ← ⊤              // оптимистическое начальное значение
    Worklist ← Worklist ∪ { f }

for каждого места вызова s в программе do
  for каждого формального параметра f, который получает значение в s do
    Value(f) ← Value(f) ∧ J_s^f      // начальные константы разбираются по J_s^f

// Этап 2: продолжать итерации до нахождения неподвижной точки
while (Worklist ≠ ∅) do
  выбрать параметр f из Worklist  // выбрать произвольный параметр
  пусть p – процедура, в которой объявлен f

  // обновить Value каждого параметра, зависящего от f
  for каждого места вызова s в p и параметра x таких, что f ∈ Support(J_s^x) do
    t ← Value(x)
    Value(x) ← Value(x) ∧ J_s^x   // Вычислить новое значение
    if (Value(x) < t) then
      Worklist ← Worklist ∪ { x }

// Постобработка множеств Value для порождения Constants
for каждой процедуры p do
  Constants(p) ← ∅
  for каждого формального параметра f процедуры p do
    if (Value(f) = ⊤) then
      Value(f) ← ⊥
    if (Value(f) ≠ ⊥) then
      Constants(p) ← Constants(p) ∪ { ⟨f, Value(f)⟩ }
```

Рис. 9.24 Итеративный алгоритм межпроцедурного распространения констант

Реализация скачковой функции

Реализации скачковых функций варьируются от простых статических аппроксимаций, которые не изменяются на протяжении анализа, до небольших параметризованных моделей и более сложных схем, в которых выполняется детальный анализ в каждой точке вычисления скачковой функции. В любой из этих схем действует несколько принципов. Если анализатор определяет, что параметр x в месте вызова s является известной константой c, то $J_s^x = c$ и $Support(J_s^x) = \emptyset$. Если $y \in Support(J_s^x)$ и $Value(y) = \top$, то $J_s^x = \top$. Если анализатор понимает, что значение J_s^x определить невозможно, то $J_s^x = \bot$.

Например, $Support(J_s^x)$ мог бы содержать значение, прочитанное из файла, так что $J_s^x = \bot$.

Анализатор может реализовать $\mathcal{J}_s^x$ многими способами. Простая реализация могла бы распространять константу, только если значение попадает в процедуру в виде формального параметра и без изменения доходит до параметра в месте вызова, т. е. фактический параметр *x* является SSA-именем формального параметра в процедуре, содержащей место вызова *s*. (Похожую функциональность можно получить, используя информацию из множества Reaches из раздела 9.2.4.)

Возможны и более сложные схемы, которые находят больше констант. Компилятор мог бы построить выражения, составленные из SSA-имен формальных параметров и литеральных констант. Тогда скачковая функция интерпретировала бы выражение на полурешеточных значениях содержащихся в нем SSA-имен и констант. Чтобы получить еще более точные результаты, компилятор мог бы по требованию выполнять алгоритм SSCP для обновления значений скачковых функций.

Обобщение алгоритма

Алгоритм, показанный на рис. 9.24, только распространяет постоянные фактические параметры по ребрам графа вызовов. Мы можем естественным образом обобщить его, чтобы он обрабатывал возвращаемые значения и переменные, глобальные относительно процедуры.

Точно так же, как алгоритм строит скачковые функции для моделирования потока значений от вызывающей стороны к вызываемой, он может построить *возвратные скачковые функции* для моделирования значений, возвращаемых от вызываемой стороны вызывающей. Возвратные скачковые функции особенно важны для процедур, которые инициализируют значения – не важно, путем заполнения общего блока в FORTRAN или задания начальных значений для объекта или класса в JAVA. Алгоритм может обрабатывать возвратные скачковые функции точно так же, как обыкновенные; единственное существенное осложнение заключается в том, что реализация должна избегать создания расходящихся циклов возвратных скачковых функций (например, для процедуры с хвостовой рекурсией).

Чтобы обобщить алгоритм на более широкий класс переменных, компилятор может подходящим образом расширить вектор скачковых функций. Расширение множества переменных увеличит стоимость анализа, но два фактора могут ее снизить. Во-первых, при построении скачковой функции анализатор может заметить, что у многих из этих переменных нет значений, которые можно легко смоделировать; он может отобразить эти переменные на универсальную скачковую функцию, которая возвращает $\perp$, и не помещать их в рабочий список. Во-вторых, для переменных, которые могут иметь постоянные значения, структура решетки гарантирует, что они встретятся в рабочем списке не более двух раз. Поэтому алгоритм по-прежнему должен работать быстро.

ИТОГИ РАЗДЕЛА

Компиляторы выполняют межпроцедурный анализ, чтобы понять поведение всех процедур в программе и использовать эти знания для оптимизации отдельных процедур. Чтобы выполнить межпроцедурный анализ, компилятор должен смоделировать весь анализируемый код. Для решения типичной межпроцедурной задачи компилятор должен построить граф вызовов (или какой-то его аналог), аннотировать его информацией, выведенной из отдельных процедур, и распространить эту информацию по графу.

Собранная в процессе межпроцедурного анализа информация применяется непосредственно к внутрипроцедурному анализу и оптимизации. Например, множества MayMod и MayRef можно использовать, чтобы уменьшить влияние места вызова на анализ глобального потока данных или избежать необходимости φ-функций после места вызова. Результаты межпроцедурного распространения констант можно использовать, чтобы инициализировать глобальный алгоритм, такой, например, как разреженное условное распространение констант (см. раздел 10.7.1).

ВОПРОСЫ

1. Построение графа вызовов во многом похоже на межпроцедурное распространение констант. Алгоритм может достичь хороших результатов, используя относительно простые скачковые функции. Какие языковые средства могли бы привести к необходимости более сложных скачковых функций в построителе графа вызовов?
2. Как анализатор мог бы включить информацию из множеств MayMod в алгоритм межпроцедурного распространения констант? Какого эффекта вы ожидаете от этого?

9.5. Дополнительные темы

В разделе 9.2 речь шла об итеративном анализе потока данных. Мы уделили столько внимания итеративному подходу, потому что он простой, надежный и эффективный. Другие подходы к анализу потока данных опираются на структурные свойства анализируемого графа. В разделе 9.5.1 обсуждается приводимость графа потока – критически важное свойство большинства структурных алгоритмов.

В разделе 9.5.2 мы вернемся к итеративному каркасу доминирования из раздела 9.2.1. Простота делает его привлекательным, однако у более специализированных и сложных алгоритмов асимптотическая сложность значительно меньше. Мы введем набор структур данных, благодаря которым простой итеративный метод становится конкурентоспособен с алгоритмами доминирования для графов с числом вершин порядка нескольких тысяч.

9.5.1. Структурный анализ потока данных и приводимость

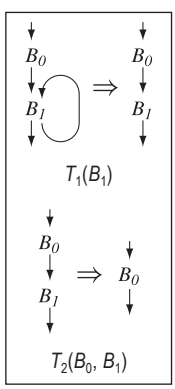
В главах 8 и 9 представлена итеративная формулировка анализа потока данных. Вообще говоря, итеративный алгоритм работает для любой регулярной системы уравнений на любом графе. Существуют и другие алгоритмы об-

работки потока данных; в основе многих из них лежит вывод простой модели структуры потока управления для анализируемого кода и использование этой модели для решения уравнений. Часто модель строится путем отыскания последовательности преобразований ГПУ, уменьшающей его сложность, – путем тщательно подобранного комбинирования вершин или ребер. Этот процесс приведения графа составляет основу почти любого алгоритма работы с потоком данных, *кроме* итеративного.

Приводимый граф
Граф потока называется *приводимым*, если два преобразования, T_1 и T_2, сводят его к одной вершине. Если этот процесс завершается неудачно, то граф называется *неприводимым*.

В этих *структурных* алгоритмах анализа потока данных используется небольшое подмножество преобразований, каждое из которых выбирает некоторый подграф и заменяет его одной представляющей его вершиной. В результате создается последовательность производных графов, в которой каждый граф получается из своего предшественника применением одного преобразования. Преобразуя граф, анализатор вычисляет связанные с потоком данных множества для новых репрезентативных вершин каждого производного графа. Эти множества суммируют свойства замененных подграфов. Преобразования сводят графы с регулярным поведением к одной вершине. Затем алгоритм обращает этот процесс, т. е. переходит от конечного производного графа с одной вершиной обратно к исходному графу потока. Расширяя граф до исходной формы, анализатор вычисляет окончательные множества потока данных для каждой вершины.

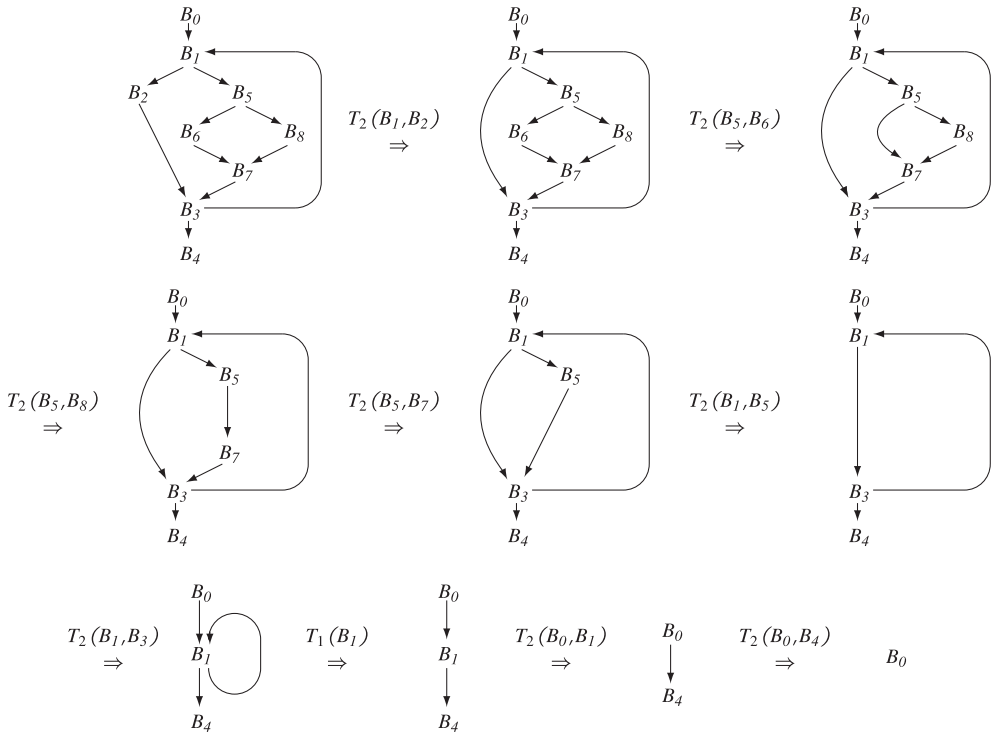
По существу, на этапе приведения собирается и консолидируется информация обо всем графе, а на этапе расширения свойства, обнаруженные в консолидированном множестве, распространяются на вершины исходного графа. Любой граф, для которого такая последовательность приведения завершается успешно, считается *приводимым*. Если граф нельзя свести к одной вершине, то он называется *неприводимым*.



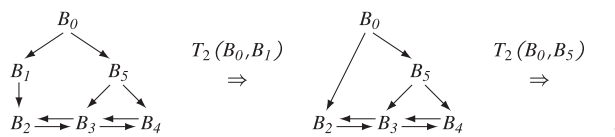
Для демонстрации приводимости мы можем использовать два преобразования графа, называемых T_1 и T_2 и показанных во врезке. Те же самые преобразования лежат в основе классического алгоритма потока данных. T_1 удаляет петлю, т. е. ребро, начинающееся и заканчивающееся в одной и той же вершине. Во врезке показано применение T_1 к B_1, обозначаемое $T_1(B_1)$. T_2 ворачивает вершину B_1, имеющую в точности одного предшественника B_0, в B_0; оно удаляет ребро $\langle B_0, B_1\rangle$ и делает B_0 начальной вершиной для всех ребер, ранее исходивших из B_1. Если при этом получается несколько ребер, ведущих из B_0 в какую-то другую вершину n, то все они склеиваются в одно. Во врезке показано преобразование $T_2(B_0, B_1)$.

Любой граф, который можно преобразовать, или *свести*, к одной вершине повторным применением T_1 и T_2, называется приводимым. Чтобы понять, как это работает, рассмотрим ГПУ из нашего сквозного примера. На

рис. 9.25(а) показана одна последовательность преобразований T_1 и T_2, которая приводит ГПУ к графу с одной вершиной. В ней T_2 применяется до тех пор, пока это возможно: $T_2(B_1, B_2)$, $T_2(B_5, B6)$, $T_2(B_5, B_8)$, $T_2(B_5, B7)$, $T_2(B_1, B_5)$ и $T_2(B_1, B_3)$. Затем $T_1(B_1)$ применяется, чтобы удалить петлю. Наконец, применяются преобразования $T_2(B_0, B_1)$ и $T_2(B_0, B_4)$ для сведения графа к одной вершине. Эта последовательность доказывает, что граф приводим.



(a) Пример ГПУ на рис. 9.4



(b) Неприводимый граф

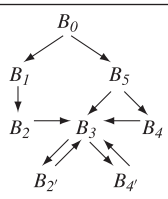
Рис. 9.25 Последовательности приведения

Для приведения графа можно применить преобразования и в другом порядке. Например, начав с $T_2(B_1, B_5)$, мы получим другую последовательность преобразований. Преобразования T_1 и T_2 обладают свойством конечности Чёрча–Россера, которое гарантирует, что конечный результат не зависит от порядка их применения и последовательность завершается. Таким образом, анализатор может найти места в графе, где применимо T_1 или T_2, и использовать их оппортунистически.

На рис. 9.25(b) показано, что может случиться, когда T_1 и T_2 применяются к графу с несколькими петлями. Анализатор использует $T_2(B_0, B_1)$, а затем $T_2(B_0, B_5)$. Однако в этот момент не остается ни вершин, ни пар вершин, к которым можно было бы применить T_1 или T_2. Следовательно, анализатор не может продолжить приведение графа. (Никакой другой порядок не изменит это положение.) Граф нельзя свести к одной вершине, он неприводим.

Существует много других критериев приводимости графа. Один простой и быстрый критерий предлагает применить к графу итеративный каркас Dom в порядке обхода РОП. Если для вычисления нужно больше двух итераций, то граф неприводим.

Невозможность привести граф с помощью преобразований T_1 и T_2 проистекает из фундаментального свойства графа. Граф неприводим, потому что содержит цикл, в который можно войти из нескольких вершин. В терминах исходного языка это означает, что программа, для которой построен граф, имеет цикл с несколькими точками входа. Мы можем наблюдать это свойство в графе; взгляните на цикл, образованный ребрами B_2 и B_3. В него можно войти из вершин B_1, B_4 и B_5. Аналогично в цикл, образованный B_3 и B_4, ведут ребра из B_2 и B_5.



Неприводимый граф после расщепления вершины

Неприводимость ставит серьезную проблему перед алгоритмами, построенными на базе преобразований типа T_1 и T_2. Если алгоритм не может свести граф к одной вершине, то метод должен либо сообщить о неудаче и необходимости модифицировать граф путем расщепления одной или нескольких вершин, либо воспользоваться итеративным подходом для решения системы уравнений на частично приведенном графе. В общем случае структурные алгоритмы анализа потока данных работают только на приводимых графах. Напротив, итеративный алгоритм правильно работает на неприводимом графе, хотя и медленнее.

Чтобы преобразовать неприводимый граф в приводимый, анализатор может расщепить одну или несколько вершин. Простейшее расщепление для графа на рис. 9.25(b) показано во врезке. Преобразование клонировало B_2 и B_4, создав соответственно $B_{2'}$ и $B_{4'}$. Затем анализатор перенаправил ребра (B_3, B_2) и (B_3, B_4), образовав более сложный цикл $\{B_3, B_{2'}, B_{4'}\}$. У нового цикла всего одна точка входа, B_3.

Это преобразование создает приводимый граф, выполняющий ту же последовательность операций, что и исходный. Пути, которые в исходном графе входили в B_3 из B_2 или B_4, теперь выполняются как прологи к циклу $\{B_3, B'_2, B'_4\}$. У вершин B_2 и B_4 в новом графе по одному предшественнику. У B_3 предшественников несколько, но она является единственной точкой входа в цикл, и цикл приводим. Таким образом, расщепление породило приводимый граф ценой клонирования двух вершин.

В фольклоре и в научных статьях предполагается, что неприводимые графы редко возникают в анализе глобального потока данных. Благодаря пропаганде структурного программирования в 1970-х годах программисты стали

В обращенном ГПУ `break` становится второй точкой входа в циклический регион.

гораздо реже использовать произвольные передачи управления, в частности предложение `goto`. Структурные циклические конструкции типа `do`, `for`, `while` и `until` не порождают неприводимые графы. Однако передача управления вовне цикла (например, предложение `break` в языке C) создает неприводимый ГПУ. Неприводимые графы также часто возникают при межпроцедурном анализе вследствие взаимно рекурсивных подпрограмм. Например, граф вызовов для парсера рекурсивного спуска, скорее всего, будет иметь неприводимые подграфы. По счастью, итеративный анализатор умеет обрабатывать неприводимые графы корректно и эффективно.

Простой способ избежать поведения в худшем случае, возникающего из-за неприводимого графа в итеративном анализаторе, заключается в том, чтобы вычислить два порядка обхода: при одном вершины на одном уровне обходятся слева направо, а при другом справа налево. Чередование порядка на соседних проходах улучшит поведение для неприводимых графов в худшем случае.

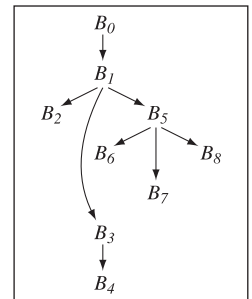
9.5.2. Ускорение итеративного каркаса доминирования

Итеративный каркас для вычисления доминирования особенно прост. Если для большинства задач о потоке данных уравнения включают несколько множеств, то для Dom нужно лишь вычислить попарные пересечения множеств Dom и добавить в эти множества один элемент. Простая природа этих уравнений открывает удобную возможность: мы можем использовать разреженную структуру данных, чтобы повысить скорость вычисления Dom.

В итеративном каркасе Dom, описанном в разделе 9.2.1, в каждой вершине хранится полное множество Dom. Компилятор может достичь того же результата, если будет хранить в каждой вершине только непосредственный доминатор, IDom, и вычислять IDom. Компилятор легко может воссоздать Dom(n), когда возникнет необходимость. Поскольку IDom – множество с одним элементом, реализация может оказаться очень эффективной.

Вспомним наш пример ГПУ из раздела 9.2.1, он повторен во врезке вместе с деревом доминирования. Вот его множества IDom:

	B_0	B_1	B_2	B_3	B_4	B_5	B_6	B_7	B_8
IDom(n)	–	0	1	1	3	1	5	5	5



Пример дерева доминирования

Заметим, что дерево доминирования и множества IDom изоморфны. IDom(b) – это просто предшественник b в дереве доминирования. У корня дерева доминирования нет предшественника, поэтому для него множество IDom не определено.

Компилятор может прочитать множества Dom графа из его дерева доминирования. Множество Dom для вер-

шины n – это просто множество вершин, лежащих на пути из n в корень дерева доминирования, включая конечные точки. В нашем примере путь в дереве доминирования из B_7 в B_0 состоит из вершин (B_7, B_5, B_1, B_0), что совпадает с Dom(B_7) из раздела 9.2.1.

Таким образом, компилятор может использовать множества IDom как заместители множеств Dom, при условии что сумеет эффективно инициализировать их и вычислять пересечения. Небольшая модификация итеративного алгоритма может упростить инициализацию. Для пересечения нужен более тонкий подход, показанный на рис. 9.26. Критическая процедура *Intersect* опирается на два наблюдения.

1. Когда алгоритм проходит по пути из вершины в корень, чтобы воссоздать множество Dom, он посещает узлы в единообразном порядке. Пересечение двух множеств Dom – это просто общий суффикс меток на путях из узлов в корень.
2. Алгоритм должен распознать общий суффикс. Он начинает с двух узлов, чьи множества пересекаются, *i* и *j*, и поднимается по дереву доминирования из каждой из них к корню. Если мы присвоим узлам имена, совпадающие с их номерами в порядке обхода РОП, то простое сравнение позволит алгоритму найти ближайшего общего предка – IDom *i* и *j*.

```
for всех вершин b do // инициализировать массив доминаторов
  IDoms[b] ← Undefined
IDoms[b0] ← b0
Changed ← true
while (Changed) do
  Changed ← false
  for всех вершин b в реверсивном обратном порядке (кроме корня) do
    NewIDom ← первый (обработанный) предшественник b // выбрать один
    for всех остальных предшественников p вершины b do
      if IDoms[p] ≠ Undefined then // т. е. Doms[p] уже вычислен
        NewIdom ← Intersect(p, NewIdom)
    if IDoms[b] ≠ NewIdom then
      IDoms[b] ← NewIdom
      Changed ← true

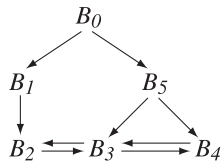
Intersect(i, j)
  finger1 ← i
  finger2 ← j
  while (finger1 ≠ finger2)
    while (РОП(finger1) > РОП(finger2)) do
      finger1 = IDoms[finger1]
    while (РОП(finger2) > РОП(finger1)) do
      finger2 = IDoms[finger2]
  return finger1
```

Рис. 9.26 Модифицированный итеративный алгоритм поиска доминаторов

Функция *Intersect* – вариант классического «двухпальцевого» алгоритма. В нем используется два указателя для прослеживания путей вверх по дереву. Когда они совпадут, оба будут указывать на узел, представляющий результат пересечения.

На рис. 9.26 показан переформулированный итеративный алгоритм для IDom. Информация о IDom хранится в массиве *IDoms*. Алгоритм инициализирует элемент IDom для корня, b_0, собой же, чтобы упростить последующую логику. Он обрабатывает узлы в реверсивном обратном порядке. При вычислении пересечений предшественники, чьи IDom еще не были вычислены, игнорируются.

Чтобы понять, как работает алгоритм, рассмотрим неприводимый граф на рис. 9.27(a). На рис. 9.27(b) показан порядок РОП для этого графа, иллюстрирующий проблемы, вызванные неприводимостью. При использовании этого порядка алгоритм вычисляет множества IDomB_3 и B_4 на первой итерации. Ему требуется две итерации, чтобы исправить эти IDom, и еще одна итерация, чтобы обнаружить, что множества IDom перестали изменяться.



(a) Неприводимый граф

	B_0	B_1	B_2	B_3	B_4	B_5
RPO(n)	0	1	5	4	3	2

(b) РОП в худшем случае

	IDom(n)					
	B_0	B_1	B_2	B_3	B_4	B_5
0	0	–	–	–	–	–
1	0	0	0	5	5	0
2	0	0	0	0	5	0
3	0	0	0	0	0	0
4	0	0	0	0	0	0

(c) Ход вычисления IDom

Рис. 9.27 Вычисление доминаторов на неприводимом графе

Этот улучшенный алгоритм работает быстро. Он потребляет немного памяти. На любом приводимом графе он останавливается после двух проходов: на первом вычисляются правильные множества IDom, а на втором подтверждается, что не произошло никаких изменений. Для неприводимого графа нужно больше двух проходов. На самом деле алгоритм дает простой критерий приводимости: если на втором проходе хотя бы оно множество IDom изменяется, то граф неприводим.

9.6. Итоги и перспективы

Большинство оптимизаций адаптируют общий случай к конкретному контексту, встречающемуся в откомпилированном коде. Способность компилятора адаптироваться к коду часто ограничена незнанием диапазона возможных поведений программы на этапе выполнения.

Анализ потока данных позволяет компилятору на этапе компиляции моделировать поведение программы во время выполнения и выводить важные конкретные факты из этих моделей. Было предложено немало задач о потоке данных; в этой главе описаны некоторые из них. У многих из них имеются свойства, повышающие эффективность анализа.

Форма SSA является одновременно промежуточным представлением и инструментом анализа. Она кодирует как информацию о потоке данных, так и информацию о зависимостях по управлению в пространстве имен программы. У использования формы SSA в качестве основы алгоритма есть три потенциальных достоинства. Она может повышать точность анализа, потому что SSA включает информацию о потоке управления. Она может приводить к более эффективным алгоритмам, потому что SSA является разреженным представлением информации о потоке данных. Она позволяет упростить применяемую оптимизацию (см. 10.7.2). Эти достоинства побудили исследователей и программистов-практиков сделать форму SSA определяющим представлением в современных компиляторах.

Замечания к этой главе

Честь первого анализа потока данных обычно признают за Высотским из компании Bell Labs, работавшим в начале 1960-х годов [351]. Но работа Лоис Хайбт по оригинальному компилятору FORTRAN предшествует трудам Высотского. Написанная ей фаза компилятора строила граф потока управления и выполняла анализ ГПУ по типу марковского для оценки частот выполнения [27].

У итеративного анализа потока данных имеется долгая история в литературе. Среди основополагающих трудов на эту тему упомянем статью Килдалла 1973 года [234], работу Хехта и Ульмана [197], и две статьи Кама и Ульмана [221, 222]. Изложение в этой главе следует работам Кама и Ульмана.

В этой главе мы занимались преимущественно итеративным анализом потока данных. Но было предложено много других алгоритмов решения задач о потоке данных [229]. Интересующемуся читателю рекомендуется изучить структурные методы, включая интервальный анализ [18, 19, 68], T_1-T_2 анализ [196, 348], алгоритм Грэхема–Вегмана [178, 179], алгоритм сбалансированного дерева со сжатием путей [342, 343], графовые грамматики [230] и метод разделенных переменных [371]. Итеративный метод с чередованием направлений обхода, упомянутый в конце раздела 9.5.1, принадлежит Харви [109].

Доминирование имеет долгую историю в литературе. Проссер ввел понятие доминирования в 1959 году, но не привел никакого алгоритма вычисления доминаторов [300]. Лоури и Медлок описали алгоритм, примененный в их компиляторе [260]; он требует как минимум времени $O(N^2)$, где N – число предложений в процедуре. Несколько авторов разработали более быстрые алгоритмы, основанные на удалении вершин из ГПУ [4, 9, 301]. Тарьян пред-

ложил алгоритм со сложностью $O(N \log N + E)$, основанный на поиске в глубину и нахождении объединения [341]. Ленгауэр и Тарьян улучшили эту оценку сложности [252], то же сделали и другие [24, 67, 190]. Формулировка задачи о потоке данных для доминаторов взята из работ Аллена [13, 18]. Первые структуры данных для итеративного доминирования предложил Харви [110]. Алгоритм на рис. 9.10 заимствован из работы Ферранте, Оттенштейна и Уоррена [155].

Построение формы SSA базируется на основополагающей работе Цитрона и др. [120]. Эта работа основана на статьях Шапиро и Сэйнта [323], Реифа [305, 344] и Ферранте, Оттенштейна и Уоррена [155]. Алгоритм в разделе 9.3 строит полуусеченную SSA-форму [55]. В работе Бриггса и др. описаны детали алгоритма переименования и ситуативный подход к трансляции из SSA-формы [56]. Унифицированный подход к трансляции из SSA-формы предложен в работе Буассино и др. [51]. Сложности, связанные с критическими ребрами, осмыслены в литературе по оптимизации уже давно [139, 141, 144, 236, 312]; не должно вызывать удивления, что они возникают также при трансляции SSA-формы в исполняемый код. Алгоритм *разреженного простого распространения констант* предложен Реифом и Льюисом [306]. Вегман и Цадлек переформулировали SSCP с использованием формы SSA [358, 359].

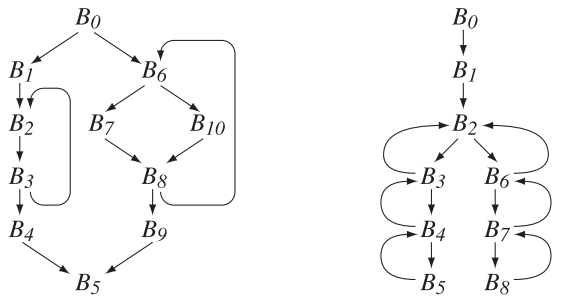
Оптимизирующий компилятор IBM PL/I был одной из ранних систем, в которой выполнялся межпроцедурный анализ потока данных [334]. Построение графа вызовов сильно зависит от языка: Райдер рассматривал проблемы, возникающие в Fortran [314], C [272] и JAVA [372]. Шиверс написал классическую статью об анализе потока управления в языках типа Scheme [325]. В ранних работах по анализу побочных эффектов больше внимания уделялось постановке задач, чем их быстрому решению [35, 37]. Купер и Кеннеди разработали простые каркасы для множеств MayMod и MayRef, которые вели к быстрым алгоритмам решения этих задач [112, 113]. Алгоритм межпроцедурного распространения констант взят из диссертации и последующих работ Торкзон [74, 182, 271]; Цитрон и Вегман предложили другие подходы к этой проблеме [121, 359]. Бёрк и Торкзон [70] описали анализ, который определяет, какие модули большой программы должны быть перекомпилированы после изменения межпроцедурной информации. Анализ указателей принципиально является межпроцедурным; эта задача выступает темой многих работ, число которых продолжает расти [84, 87, 123, 134, 149, 202, 203, 209, 247, 322, 360, 363]. Айерс, Готтлиб и Шулер описали практическую систему, которая анализировала и оптимизировала подмножество всей программы [26].

Упражнения

К разделу 9.2

1. В анализе живучести множества LiveOut каждого блока инициализируются значением $\emptyset$. Возможны ли другие инициализации? Изменится ли при этом результат анализа? Обоснуйте свой ответ.

2. Как в анализе живучести компилятор должен обрабатывать блок, содержащий вызов процедуры? Что должно содержать множество UEVar такого блока? А множество VarKill?
3. Для каждого из следующих графов потока управления:



(a) Несколько циклов (b) Тело с двойным циклом

   a. Вычислите номера вершин в реверсивном обратном порядке обхода ГПУ и обращенного ГПУ.
   b. Вычислите реверсивный прямой порядок обхода ГПУ.
   c. Эквивалентен ли реверсивный прямой порядок обхода ГПУ обратному порядку обхода обращенного ГПУ?

К разделу 9.3

4. Рассмотрите три графа потока управления, показанных на рис. 9.28.

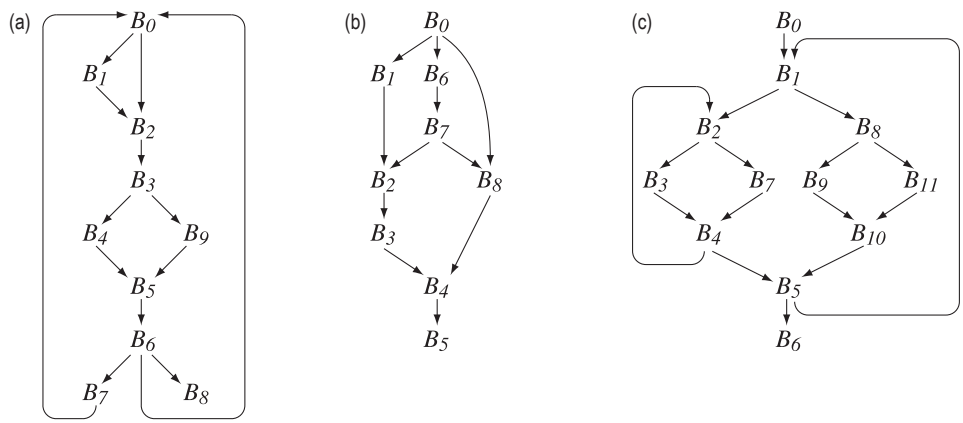
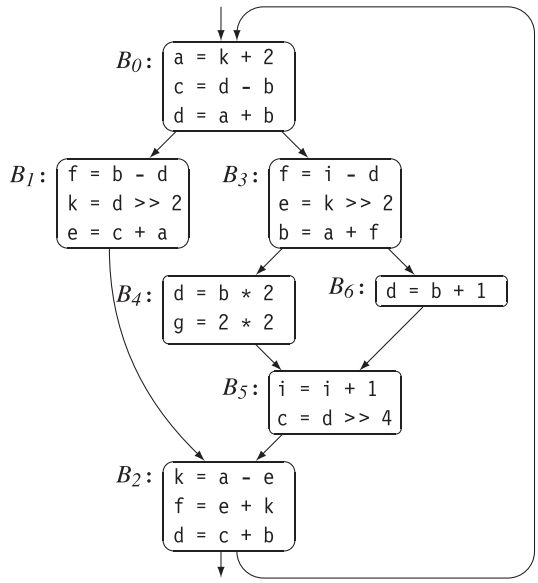


Рис. 9.28 Графы управления потоком для упражнения 4

   a. Вычислите деревья доминирования для ГПУ (a), (b) и (c).
   b. Вычислите границы доминирования для вершин 3 и 5 графа (a), вершин 4 и 5 графа (b) и вершин 2 и 10 графа (c).
5. Транслируйте показанный ниже ГПУ в форму SSA. Покажите только конечные результаты после вставки φ-функций и переименования.



6. Пусть имеется присваивание некоторой переменной *v* в блоке *b*. Рассмотрим множество блоков, которые вследствие этого нуждаются в φ-функции. Алгоритм на рис. 9.11 вставляет φ-функцию в каждый блок, принадлежащий DF(*b*). Затем он добавляет каждый из этих блоков в рабочий список; они, в свою очередь, могут добавлять дополнительные блоки в рабочий список. Назовем множество всех этих блоков $DF^+(b)$. Мы можем определить DF^+(b) как предел последовательности:

Алгоритм должен добавлять блок в рабочий список только один раз.

$DF_1(b) = DF(b)$
$DF_2(b) = DF_1(b) \cup_{x \in DF_1(b)} DF_1(x)$
$DF_3(b) = DF_2(b) \cup_{x \in DF_2(b)} DF_2(x)$
…
$DF_i(b) = DF_{i-1}(b) \cup_{x \in DF_{i-1}(b)} DF_{i-1}(x)$

Использование этих расширенных множеств $DF^+(b)$ приводит к более простому алгоритму вставки φ-функций.

a. Разработайте алгоритм вычисления $DF^+(b)$.

b. Разработайте алгоритм вставки φ-функций с использованием множеств DF^+, вычисленных в части «а».

c. Сравните общую стоимость вашего алгоритма, включая вычисление множеств DF^+, со стоимостью алгоритма вставки φ-функций, приведенного в разделе 9.3.3.

7. Алгоритм построения максимальной SSA-формы прост и интуитивно понятен. Однако он может вставить гораздо больше φ-функций, чем в алгоритме построения полуусеченной SSA-формы. В частности, он может вставлять избыточные φ-функции ($x_i \leftarrow \varphi(x_j, x_j)$) и мертвые φ-функции, результаты которых никогда не используются.

a. Предложите метод обнаружения и удаления лишних φ-функций, вставленных алгоритмом построения максимальной SSA-формы.

b. Может ли ваш метод уменьшить множество φ-функций, оставив только те, которые вставляет алгоритм построения полуусеченной SSA-формы?

c. Сравните асимптотическую сложность вашего метода со сложностью алгоритма построения полуусеченной SSA-формы.

8. Примените унифицированную схему трансляции из SSA-формы к примеру кода для проблемы потерянного копирования, показанному на рис. 9.18(a).

9. Примените унифицированную схему трансляции из SSA-формы к примеру кода для проблемы обмена, показанному на рис. 9.19(a).

К разделу 9.5

10. Для каждого графа управления потоком на рис. 9.29 определите, является ли он приводимым. (**Указание**: приведите пример последовательности преобразований T_1 и T_2, доказывающий, что граф приводим. Если такой последовательности не существует, то граф неприводим.)

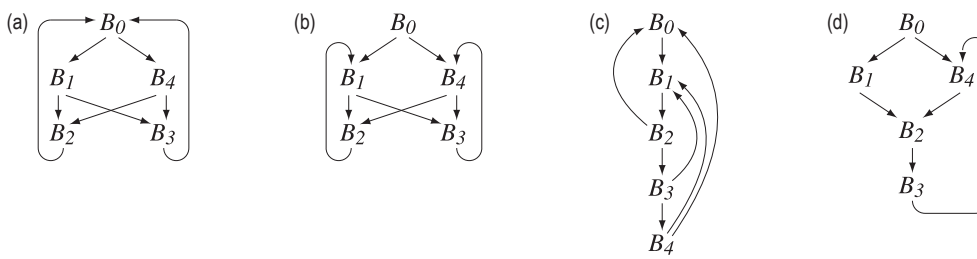


Рис. 9.29 Графы управления потоком для упражнения 10

11. Докажите, что следующее определение приводимого графа эквивалентно определению, в котором используются преобразования T_1 и T_2: граф G называется приводимым, если для каждого цикла в G существует вершина n, принадлежащая этому циклу, которая доминирует над любой другой вершиной этого цикла.

Глава 10

Скалярная оптимизация

АННОТАЦИЯ

Оптимизирующий компилятор улучшает качество сгенерированного кода путем применения к нему преобразований. Качество можно измерять скоростью работы, размером кода или более сложными метриками, например энергопотреблением процессора. В этой главе используются общие сведения об оптимизации из главы 8 и материалы по статическому анализу из главы 9, но в основном она посвящена оптимизации кода однопоточных программ – так называемой скалярной оптимизации. Будет описан ряд машинно-независимых скалярных оптимизаций, устраняющих разнообразные неэффективности откомпилированного кода.

КЛЮЧЕВЫЕ СЛОВА

Оптимизация, преобразование, машинно-независимый, машинно-зависимый, избыточность, мертвый код, распространение констант

10.1. Введение

Входящий в состав компилятора оптимизатор анализирует и преобразует ПП кода в попытке улучшить его производительность. Он применяет разные виды статического анализа (см. главу 9), чтобы найти возможности для преобразований и доказать их безопасность. Затем он преобразует, или переписывает, ПП способами, которые предположительно должны привести к улучшению кода, генерируемого фазой синтеза.

История оптимизации кода такая же долгая, как история самих компиляторов. Первый компилятор FORTRAN включал тщательную оптимизацию с целью обеспечить производительность, которая могла бы соперничать с написанным вручную ассемблерным кодом. Со времен того первого оптимизирующего компилятора, датируемого концом 1950-х годов, литература по оптимизации разрослась и теперь насчитывает тысячи статей с описанием различных видов анализа и преобразований. Решение о том, какие преоб-

разования использовать и в каком порядке, остается одним из самых сложных решений, с которыми сталкивается автор компилятора.

Скалярная оптимизация
методы улучшения кода, ориентированные на один поток управления.

Эта глава посвящена *скалярной оптимизации*, т. е. оптимизации кода в одном потоке управления (thread of control)[1]. Описывается пять основных источников неэффективности откомпилированного кода и оптимизации, позволяющих их устранить. В основу организации материала положены эти пять вопросов; мы ожидаем, что автор компилятора при выборе оптимизаций сможет воспользоваться такой организационной схемой.

Концептуальная дорожная карта

Оптимизация в рамках компилятора – процесс анализа кода с целью определения его свойств и использования результатов этого анализа для приведения кода к более эффективному виду. Улучшение можно измерять многими способами, в том числе: сокращение времени выполнения, уменьшение размера кода или снижение энергопотребления. Для каждого компилятора имеется некий набор программ, для которых он порождает высокоэффективный код. Хороший оптимизатор должен сделать такую же производительность доступной для гораздо более широкого круга программ. Оптимизатор должен быть устойчивым, т. е. небольшие изменения входной программы не должны приводить к резким изменениям производительности.

Машинная независимость
Преобразование, которое улучшает код на большинстве целевых машин, называется *машинно-независимым*.

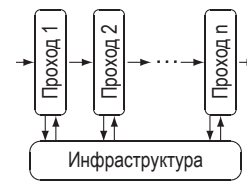
Машинная зависимость
Преобразование, которое полагается на знания о конкретном целевом процессоре, называется *машинно-зависимым*.

Оптимизатор достигает этих целей с помощью двух основных механизмов. Он исключает излишние накладные расходы на поддержку абстракций языка программирования и адаптирует потребности результирующей программы к имеющимся аппаратным и программным ресурсам целевой машины. В самом широком смысле преобразования можно разделить на *машинно-независимые* и *машинно-зависимые*. Например, заменить избыточное вычисление повторным использованием ранее вычисленного значения обычно быстрее, чем вычислить значение заново, поэтому устранение избыточности считается машинно-независимым преобразованием. С другой стороны, реализация операции копирования строки литер с помощью оборудования «распределения и сборки» на векторном процессоре, очевидно, машинно-зависима.

[1] Перегруженность слова «поток» – проблема, терзающая любого переводчика англоязычной литературы по программированию. Так переводятся термины flow, thread, stream и flux. При этом thread правильнее переводить «поток выполнения», а не управления, но авторы употребляют словосочетание «flow of control». Надеюсь, читателю из контекста будет понятно, о чем идет речь. – *Прим. перев.*

Замена этой операции копирования обращением к оптимизированной вручную системной функции `memmove` имеет шансы на более широкое применение.

Краткий обзор



Большинство оптимизаторов построены как серия проходов, пользующихся общей инфраструктурой для анализа и преобразования ПП, как показано справа. Каждый проход принимает на входе код в форме ПП и порождает на выходе переписанную версию этого ПП. Такая структура позволяет разделить реализацию на более мелкие части и избежать сложностей, присущих большим монолитным программам. Проходы можно создавать и отлаживать независимо, что упрощает разработку, тестирование и сопровождение. У компилятора появляется естественный способ предложить разные уровни оптимизации: для каждого уровня определяется набор проходов. Структура на основе проходов дает автору компилятора возможность прогонять некоторые проходы несколько раз, если это желательно. На практике одни проходы выполняются единожды, тогда как другие могут выполняться неоднократно в разных точках последовательности.

При проектировании оптимизатора выбор преобразования и порядка их применения играет важнейшую роль с точки зрения общей эффективности. Выбор преобразований определяет, какие конкретно неэффективности программы в форме ПП оптимизатор обнаруживает и как он переписывает код, чтобы их устранить. Порядок применения преобразований определяет, как проходы взаимодействуют.

Последовательности оптимизации

Выбор конкретных преобразований и порядок их применения сильно влияют на эффективность оптимизатора. Проблема осложняется еще и тем, что эффекты отдельных преобразований перекрываются (например, локальной и суперлокальной нумерациями значений), а у разных программ имеются разные комбинации неэффективностей.

Есть и другая трудность – преобразования, предназначенные для разных целей, взаимодействуют друг с другом. Одно преобразование может создать новые возможности для других. И наоборот, одно преобразование может скрыть или вообще исключить возможности для других.

Классические оптимизирующие компиляторы предлагают несколько уровней оптимизации (например, `-O`, `-O1`, `-O2`, ...) как один из способов предоставить пользователю различные последовательности преобразований. Исследователи изучали методы задания пользовательских последовательностей, состоящих из специальных кодов, которые определяют множество преобразований и порядок их применения (см. также раздел 10.7.3).

Например, в подходящем контексте (скажем, $r_2 > 0$ и $r_5 = 4$) оптимизатор мог бы заменить команду `r2,r5⇒r17` командой `lshiftI r2, 2⇒r17`. При

Напомним, что умножение коммутативно, а сдвиг – нет.

этом операция целочисленного умножения, занимающая несколько тактов, заменяется однотактовой операцией сдвига и уменьшается спрос на регистры. В большинстве случаев такое переписывание целесообразно. Однако если в следующем преобразовании компилятор собирался изменить порядок выражений, воспользовавшись коммутативностью, то замена умножения сдвигом может лишить его такой возможности. Если некоторое преобразование делает последующие проходы менее эффективными, то может пострадать качество кода в целом. Откладывание замены умножений сдвигами поможет обойти эту проблему; контекст, необходимый для доказательства безопасности и целесообразности переписывания, скорее всего, сохранится после промежуточных проходов.

Первое препятствие на пути проектирования и конструирования оптимизатора концептуальное. В литературе по оптимизации описаны сотни алгоритмов улучшения ПП программы. Разработчик компилятора должен решить, какое подмножество этих преобразований он собирается реализовать и применить. Чтение оригинальных статей, конечно, может помочь в реализации, но не в самом процессе принятия решения; авторы большинства статей агитируют за использование своих преобразований.

Автор компилятора должен понимать, какие неэффективности возможны в приложениях, оттранслированных его компилятором, и как они отразятся на производительности приложения. Зная множество конкретных дефектов, он может затем выбрать преобразования для их устранения. На самом деле многие преобразования ориентированы на устранение нескольких неэффективностей, так что тщательный отбор может сократить количество необходимых проходов. Поскольку ресурсы, доступные большинству оптимизаторов, ограничены, автор компилятора может расставить приоритеты преобразований по их ожидаемому влиянию на конечный код.

Различия между категориями не всегда очевидны. Мы называем преобразование машинно-независимым, если оно сознательно игнорирует такие технические детали, как влияние оборудования на распределение регистров.

Как отмечалось в разделе «Концептуальная дорожная карта», преобразования можно отнести к двум большим категориям: машинно-независимые и машинно-зависимые. Примерами машинно-независимых преобразований могут служить рассмотренные в предыдущих главах локальная нумерация значений (ЛНЗ), встраивание и распространение констант. Многие машинно-зависимые преобразования относятся к этапу генерирования кода. Примерами могут служить оконная (peephole) оптимизация (см. раздел 11.3), планирование команд и распределение регистров. Но есть и машинно-зависимые преобразования, применяемые оптимизатором, например балансировка деревьев по высоте, глобальное размещение кода и размещение процедур. Некоторые преобразования не поддаются классификации; у разворачивания цикла есть разные эффекты, включая уменьшение накладных

расходов на организацию цикла (машинно-независимое) и предъявление более независимых команд планировщику (машинно-зависимое).

Оптимизация как пример программной инженерии

Включение оптимизатора может упростить проектирование и реализацию компилятора. Упрощается фаза анализа, она может генерировать код общего назначения и игнорировать специальные случаи. Упрощается и фаза синтеза, она может сконцентрироваться на отображении ПП программы на целевую машину. Не будь оптимизатора, фазы анализа и синтеза должны были бы обращать внимание на то, какие есть возможности для улучшения кода.

В оптимизаторе, построенном на базе проходов, каждый проход содержит одно преобразование и анализ, необходимый для его поддержки. В принципе, каждую задачу, решаемую оптимизатором, можно реализовать однократно. В результате автор компилятора получает единую точку управления и может реализовать сложные функции один раз, а не многократно. Например, удалить операцию из ПП может быть трудно. Если после удаления операций простой блок оказывается пустым, т. е. содержит только завершающую блок операцию условного или безусловного перехода, то преобразование должно удалить блок и соединить его предшественников с его преемниками. Если эту функциональность достаточно написать только один раз, то код проще реализовать, понять и сопровождать.

С точки зрения программной инженерии, структура на основе проходов имеет смысл. Она позволяет сосредоточиться на одной задаче. Она обеспечивает четкое разделение обязанностей – нумерация значений игнорирует нехватку регистров, а распределение регистров игнорирует избыточность. Она позволяет автору компиляторов тестировать проходы независимо и тщательно, что упрощает изоляцию ошибок.

В главах 8 и 9 описан ряд методов оптимизации, которые мы отобрали для иллюстрации некоторых конкретных моментов. В главах 11–13 мы сосредоточимся на генерировании кода, которое принципиально машинно-зависимо. В этой главе будет представлен широкий круг преобразований, по преимуществу машинно-независимых. Преобразования сгруппированы по эффекту, который они оказывают на конечный код. Мы будем рассматривать пять таких эффектов:

- *устранение бесполезного и недостижимого кода.* Компилятор может определить, что некоторая операция бесполезна или недостижима. В большинстве случаев исключение такой операции дает более быстрый и меньший по размеру код;
- *перемещение кода.* Компилятор может переместить операцию в место, где она выполняется менее часто, но дает тот же результат. В большинстве случаев такое перемещение сокращает время выполнения. Иногда также уменьшается размер кода;
- *специализация вычисления.* Компилятор может специализировать последовательность кода с учетом окружающего контекста. Специализация уменьшает стоимость кода по сравнению с общим случаем;
- *исключение избыточных вычислений.* Компилятор может доказать, что некоторое значение уже было вычислено, и повторно использовать его. Во многих случаях повторное использование дешевле пересчета;

- *возможность других преобразований*. Компилятор может переписать код, так что откроются новые возможности для других оптимизаций. Примером может служить встраивание.

Этот набор категорий покрывает большинство машинно-независимых эффектов, с которыми может справиться компилятор. На практике многие преобразования ориентированы на эффекты, относящиеся к нескольким категориям. Например, ЛНЗ устраняет избыточные вычисления, специализирует вычисление с известными постоянными значениями и использует алгебраические тождества для идентификации и устранения некоторых видов бесполезных вычислений.

Далее в этой главе изучаются эти пять возможностей: устранение мертвого кода, перемещение кода, специализация, устранение избыточности и создание возможностей для других преобразований. Из каждой категории рассматривается несколько преобразований. В разделе «Дополнительные темы» представлен алгоритм, объединяющий распространение констант и устранение мертвого кода, а также алгоритм снижения стоимости операций.

10.2. Устранение мертвого кода

Иногда программы содержат вычисления, не имеющие видимого извне эффекта. Если компилятор может определить, что операция не влияет на результаты программы, то может ее устранить. Как правило, программисты не пишут такой код намеренно. Однако в большинстве программ он возникает как прямой результат других оптимизаций или вследствие наивной трансляции в фазе анализа.

Бесполезная
Операция *бесполезна*, если никакая другая операция не пользуется ее результатом или если все случаи его использования сами являются мертвым кодом.

Операция, которая не изменяет никакое значение, например `a ← a+0`, также *бесполезна*.

Недостижимая
Операция *недостижима*, если она не принадлежит никакому реализуемому пути выполнения программы.

Есть две причины, по которым операция может быть сочтена мертвым кодом, подлежащим удалению. Операция может быть *бесполезной*, т. е. ее результат не имеет никакого видимого извне эффекта. Или же операция может быть *недостижимой*, т. е. она не выполняется ни при каких обстоятельствах. Если операция относится к любой из этих категорий, то ее можно устранить. Для обозначения бесполезного или недостижимого кода часто употребляется термин *мертвый код*.

Удаление бесполезного или недостижимого кода уменьшает ПП кода, что ведет к уменьшению размера исполняемой программы, ускорению компиляции и зачастую к более быстрому выполнению. Также это может расширить возможности компилятора по улучшению кода. Например, недостижимый код может иметь эффекты, проявляющиеся в результатах статического анализа,

и мешать применению некоторых преобразований. В этом случае устранение недостижимого кода может изменить результаты анализа и открыть возможность для последующих преобразований (см., например, алгоритм разреженного условного распространения констант в разделе 10.7.1).

Некоторые виды устранения избыточности также удаляют бесполезный код. Как мы видели в разделе 8.4.1, ЛНЗ применяет алгебраические тождества для упрощения кода, например `a ← a + 0`, `b ← b x 1` и `c ← max(c,c)`. Каждое из этих упрощений устраняет бесполезную операцию, т. е. операцию, удаление которой никак не влияет на видимое извне поведение программы.

Поскольку алгоритмы, рассматриваемые в этом разделе, модифицируют граф потока управления (ГПУ) программы, мы различаем термины *условный переход*, как в команде ILOC `cbr`, и *безусловный переход*, как в команде ILOC `jump`. Внимательное отношение к этому различию поможет читателю понять алгоритмы.

10.2.1. Устранение бесполезного кода

Операция может установить значение, возвращаемое процедурой, несколькими способами: присвоить значение параметру, переданному по ссылке, или глобальной переменной, выполнить присваивание по неоднозначному указателю или передать значение в предложении return.

Классические алгоритмы устранения бесполезного кода работают по принципу сборщиков мусора с пометкой и очисткой, только применяются к коду в промежуточной форме как к данным (см. раздел 6.6.2). Как и сборщики мусора, они выполняют два прохода по коду. На первом проходе все поля пометки очищаются, а «критические» операции помечаются как «полезные». Операция считается *критической*, если она устанавливает значение, возвращаемое процедурой, является предложением ввода-вывода или влияет на значение по адресу памяти, который может быть доступен извне текущей процедуры. Примерами критических операций служат пролог и эпилог процедуры, а также предварительная и заключительная последовательности в местах вызова. Затем алгоритм прослеживает операнды полезных операций до их определений и помечает эти операции как полезные. Этот процесс продолжается, следуя простой итеративной схеме с рабочим списком, пока не кончатся потенциально полезные операции. На втором проходе алгоритм пробегает по коду и удаляет все операции, не помеченные как полезные.

На рис. 10.1 эти идеи конкретизированы. Алгоритм, который мы назовем *Dead*, предполагает, что код представлен в форме SSA. Это упрощает процесс, потому что каждое использование ссылается на единственное определение. Алгоритм *Dead* состоит из двух проходов. На первом, названном *Mark*, строится множество полезных операций. На втором, названном *Sweep*, бесполезные операции удаляются. *Mark* опирается на зависимость по управлению, которая тесно связана с границами доминирования (см. раздел 9.3.2).

```
Mark( )
  WorkList ← ∅
  for каждой операции i do
    очистить пометку i
    if i критическая then
      пометить i
      WorkList ← WorkList ∪ {i}
    while (WorkList ≠ ∅) do
      удалить операцию i из WorkList
        (предполагается, что i имеет вид
        x ← y op z)
      if def(y) не помечена then
        пометить def(y)
        WorkList ← WorkList ∪ {def(y)}
      if def(z) не помечена then
        пометить def(z)
        WorkList ← WorkList ∪ {def(z)}
      for каждого блока b ∈ RDF(block(i)) do
        пусть x – ветвь, которой заканчивается b
        if x не помечена then
          пометить x
          WorkList ← WorkList ∪ {x}
```

(a) Фаза пометки

```
Sweep( )
  for каждой операции i do
    if i не помечена then
      if i - условный переход then
        переписать i, используя
          безусловный переход на
          ближайший к i помеченный
          постдоминатор
      if i - небезусловный переход
      then
        удалить i
```

(b) Фаза очистки

Рис. 10.1 Устранение бесполезного кода

Обработка операций, отличных от условных и безусловных переходов, не вызывает трудностей. На фазе пометки определяется, является ли операция полезной. На фазе очистки операции, не помеченные как полезные, удаляются.

Обработка операций, изменяющих поток управления, сложнее. Каждый безусловный переход считается полезным. Условные переходы считаются полезными, только если от их присутствия зависит выполнение какой-нибудь полезной операции. Поскольку фаза пометки находит все полезные операции, она заодно помечает подходящие условные переходы как полезные. Чтобы перейти от помеченной операции к условным переходам, которые делают ее полезной, алгоритм опирается на зависимость по управлению.

Постдоминирование
В ГПУ *j постдоминирует над i* тогда и только тогда, когда каждый путь из *i* в выходную вершину проходит через *j*.

См. также определение доминирования на стр. 471.

Определение зависимости по управлению опирается на понятие *постдоминирования*. В ГПУ вершина *j* постдоминирует над вершиной *i*, если любой путь из *i* в выходную вершину ГПУ проходит через *j*. Пользуясь понятием постдоминирования, мы можем определить зависимость по управлению следующим образом: вершина ГПУ *j* зависит по управлению от *i*, если:

1) существует непустой путь из *i* в *j* такой, что *j* постдоминирует над каждой вершиной на этом пути после *i*. Если выполнение начинается на

этом пути, то оно должно пройти через *j*, чтобы достичь выхода из ГПУ (по определению постдоминирования);

2) *j* нестрого постдоминирует на *i*. Другое ребро исходит из *i*, и управление может пройти по пути в вершину, не лежащую на пути к *j*. Должен существовать путь, начинающийся этим ребром, который ведет к выходу из ГПУ, не проходя через *j*.

Иными словами, блок *i* покидают два или более ребер. Одно или несколько ребер ведут в *j*, и одно или несколько не ведут туда. Таким образом, решение, принятое в точке ветвления, которая завершает блок *i*, может определить, выполняется *j* или нет. Если операция *j* полезна, то операция условного перехода, завершающая *i*, также полезна.

Это понятие зависимости по управлению точно улавливается *обратной границей доминирования j*, обозначаемой RDF(*j*). Обратная граница доминирования – это просто граница доминирования, вычисленная на обращенном ГПУ. Помечая операцию в блоке *b* полезной, алгоритм *Mark* посещает каждый блок, принадлежащий обратной границе доминирования *b*, и помечает заканчивающие их операции условного перехода как полезные. Попутно он добавляет их в рабочий список. Когда рабочий список оказывается пустым, алгоритм останавливается.

Алгоритм *Sweep* заменяет все непомеченные операции условного перехода безусловным переходом на первый постдоминирующий блок, который содержит помеченную операцию. Если условный переход не помечен, то его преемники вплоть до непосредственного постдоминатора не содержат полезных операций. (В противном случае в момент пометки этих операций операции условного перехода были бы помечены.) Похожее рассуждение применимо и тогда, когда непосредственный постдоминатор не содержит помеченных операций. Чтобы найти ближайший полезный постдоминатор, алгоритм может пройти вверх по дереву постдоминирования, пока не найдет блок, содержащий полезную операцию. Так как, по определению, выходной блок полезен, этот поиск должен завершиться.

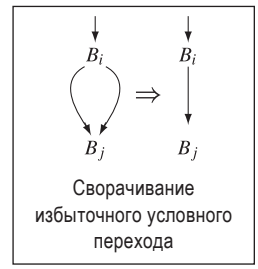
После выполнения *Dead* код не содержит бесполезных вычислений. Он может содержать пустые блоки, которые будут удалены следующим алгоритмом.

10.2.2. Устранение бесполезного потока управления

Оптимизация может переписать ПП программы, так что в ней появляется бесполезный поток управления. Если компилятор включает оптимизации, которые влияют на поток управления в качестве побочного эффекта, то он должен включать и проход, который упрощает ГПУ, устраняя бесполезный поток управления. В данном разделе представлен алгоритм *Clean*, решающий эту задачу.

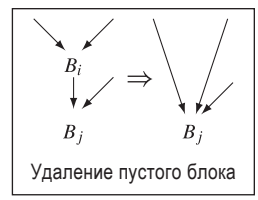
Clean работает непосредственно с ГПУ процедуры. Он состоит из четырех преобразований, показанных во врезке ниже, которые применяются в следующем порядке.

1. *Свернуть избыточный условный переход.* Если *Clean* находит блок, завершающийся условным переходом, и обе ветви этого перехода ведут на один и тот же блок, то условный переход заменяется безусловным. Такая ситуация возникает в результате других преобразований. Например, B_i мог бы иметь двух преемников, каждый из которых безусловно переходит на B_j. Если какое-то другое преобразование уже сделало эти блоки пустыми, то удаление пустых блоков, обсуждаемое ниже, могло бы дать начальный граф, показанный во врезке.



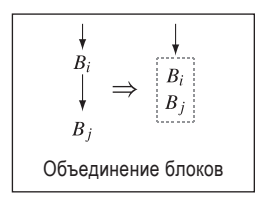
Сворачивание избыточного условного перехода

2. *Удалить пустой блок.* Если *Clean* находит блок, содержащий только безусловный переход, то он может объединить его с преемником. Эта ситуация возникает, когда другие проходы удалили все операции из блока B_i. Рассмотрим начальный граф, показанный во врезке. Поскольку у B_i имеется только один преемник, B_j, то преобразование перенаправляет ребра, входящие в B_i, на B_j и удаляет B_i из множества предшественников B_j. Это преобразование упрощает граф. Оно также должно ускорить выполнение. В исходном графе пути, проходящие через B_i, нуждались в двух переходах, чтобы достичь B_j. В преобразованном графе им достаточно одного перехода.



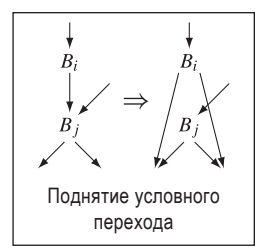
Удаление пустого блока

3. *Объединение блоков.* Если *Clean* находит блок B_i, заканчивающийся безусловным переходом на B_j, и B_j имеет только одного предшественника, то можно объединить B_i и B_j, как показано во врезке. Такая ситуация может возникнуть несколькими способами. Какое-то другое преобразование могло бы устранить другие ребра, входящие в B_j, или B_i и B_j могли бы быть результатом сворачивания избыточного условного перехода. В любом случае оба блока можно объединить в один. Это устраняет переход в конец B_i.



Объединение блоков

4. *Поднятие условного перехода.* Если *Clean* находит блок B_i, заканчивающийся безусловным переходом на пустой блок B_j, и B_j заканчивается условным переходом, то *Clean* может заменить безусловный переход в B_i копией условного перехода из B_j . Это означает, что мы поднимаем условный переход на уровень B_i, как показано во врезке. Эта ситуация возникает, когда другие проходы устранили операции в B_j, оставив только безусловный переход на условный. Переписанный код достигает того же результата, затратив на один безусловный переход меньше. При этом в ГПУ добавляется одно ребро.



Поднятие условного перехода

Обратите внимание, что B_i не может быть пустым, иначе *Clean* удалил бы его. И он не может быть единственным предшественником B_j, иначе *Clean* объединил бы B_i и B_j. (После поднятия у B_j останется еще хотя бы один предшественник.)

Многие компиляторы и ассемблеры включали ситуативный проход для устранения безусловных переходов на безусловный или условный переход. *Clean* достигает того же результата систематически.

Для реализации этих преобразований необходим внутренний учет. Некоторые модификации тривиальны. Чтобы свернуть избыточный условный переход в программе, представленной ILOC и ГПУ, *Clean* просто перезаписывает условный переход, завершающий блок, безусловным и подправляет список преемников и предшественников блоков. Другие преобразования сложнее. Объединение двух блоков может включать выделение памяти для нового блока, копирование в него операций, корректировку списков предшественников и преемников нового блока и его соседей в ГПУ, а также уничтожение обоих исходных блоков.

Clean применяет эти четыре преобразования систематическим способом. Он обходит граф в обратном порядке, так чтобы упрощать преемников B_i раньше самого B_i, если только преемник не принадлежит обратному ребру относительно нумерации в обратном порядке. В таком случае *Clean* посещает предшественника раньше преемника. Такой порядок обхода неизбежен в любом циклическом графе. Упрощение преемников раньше предшественников уменьшает количество перемещений некоторых ребер.

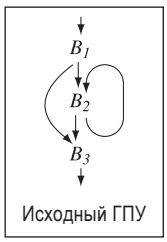
В некоторых ситуациях может быть применимо несколько преобразований. Внимательный анализ различных случаев приводит к порядку, показанному на рис. 10.2, который соответствует порядку, в котором они были представлены в тексте. Алгоритм может применить к блоку несколько преобразований за одно посещение.

```
MakeAPass( )
  for каждого блока i в обратном порядке do
    if i заканчивается условным переходом then
      if оба места назначения одинаковы then
        заменить условный переход безусловным                    /* случай 1 */
    if i заканчивается безусловным переходом на j then
      if i пуст then
        заменить переходы на i переходами на j                   /* случай 2 */
      if j имеет только одного предшественника then
        объединить i и j                                         /* случай 3 */
      if j пуст и заканчивается условным переходом then
        перезаписать безусловный переход i копией условного перехода j  /* случай 4 */

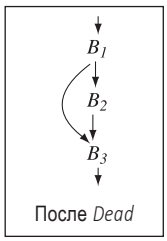
Clean( )
  while ГПУ продолжает изменяться do
    вычислить обратный порядок
    MakeAPass( )
```

Рис. 10.2 Алгоритм *Clean*

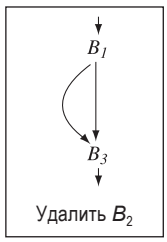
Если ГПУ содержит обратные ребра, то проход *Clean* может создать дополнительные возможности – необработанные преемники на обратных ребрах. Они, в свою очередь, также могут создать новые возможности. Поэтому *Clean* повторяет последовательность преобразований, пока ГПУ не перестанет изменяться. После каждого вызова *MakeAPass* он должен заново вычислить нумерацию в обратном порядке, потому что любой проход изменяет исходный граф. На рис. 10.2 приведен псевдокод *Clean*.



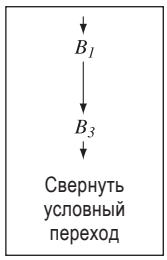
Clean не может сам по себе устранить пустой цикл. Рассмотрим ГПУ, показанный во врезке. Предположим, что блок B_2 пуст. Ни одно из преобразований, выполняемых *Clean*, не может устранить B_2, потому что условный переход, заканчивающий B_2, не является избыточным. B_2 не заканчивается безусловным переходом, поэтому *Clean* не может объединить его с B_3. Его предшественник заканчивается условным переходом, а не безусловным, поэтому *Clean* не может ни объединить блок B_2 с B_1, ни свернуть его условный переход в B_1.



Однако кооперация между *Clean* и *Dead* может исключить пустой цикл. *Dead* пользовался зависимостью по управлению для пометки полезных условных переходов. Если B_1 и B_3 содержат полезные операции, а B_2 нет, то проход *Mark* алгоритма *Dead* решит, что условный переход, завершающий B_2, не является полезным, потому что $B_2 \notin \mathrm{RDF}(B_3)$. Поскольку переход бесполезен, код, вычисляющий условие перехода, также бесполезен. Поэтому *Dead* устраняет все операции в B_2 и преобразует условный переход, завершающий его, в безусловный переход на ближайший полезный постдоминатор, B_3. Это устраняет исходный цикл и порождает ГПУ, помеченный «После Dead» во врезке.



В таком виде *Clean* сворачивает B_2 в B_1 и порождает ГПУ, помеченный «Удалить B_2» во врезке. Это действие делает условный переход в конце B_1 избыточным. *Clean* переписывает его безусловным переходом, что порождает ГПУ, помеченный «Свернуть условный переход» во врезке. В этой точке если B_1 является единственным оставшимся предшественником B_3, то *Clean* объединяет оба блока в один.



Эта кооперация проще и эффективнее добавления в *Clean* преобразования для обработки пустых циклов. Такое преобразование могло бы распознать условный переход из B_i в себя и для пустого B_i заменить его безусловным переходом на другую цель условного перехода. Проблема в определении того, является ли B_i действительно пустым. Если B_i не содержит никаких операций, кроме условного перехода, то код, который вычисляет условие перехода, должен находиться вне цикла. Таким образом, преобразование безопасно, только если петля никогда не выполняется. Для рассуждений о том, сколько раз выполняется петля, нужно что-то знать о значениях времени

выполнения, а эта задача в общем случае непосильна компилятору. Если блок содержит операции, но только такие, которые управляют условным переходом, то преобразование должно было бы распознать такую ситуацию с помощью сопоставления с образцом. В любом случае это новое преобразование было бы сложнее тех четырех, которые входят в состав *Clean*. Использование комбинации *Dead* и *Clean* достигает желаемого результата более простым и модульным способом.

10.2.3. Устранение недостижимого кода

Иногда ГПУ содержит недостижимый код. Компилятор должен найти такие блоки и удалить их. Блок может оказаться недостижимым по двум причинам: в ГПУ отсутствует путь, ведущий к этому блоку, или все ведущие к нему пути не исполняются, например потому что стоящее перед ними условие всегда равно false.

С первым случаем все просто. Компилятор может выполнить простой анализ ГПУ на предмет достижимости в стиле алгоритма пометки и очистки. Сначала он помечает каждый блок как недостижимый. Затем, начиная с входной вершины, он помечает все вершины ГПУ, до которых может добраться, как достижимые. Если все условные и безусловные переходы однозначны, то все непомеченные блоки можно удалить. При наличии неоднозначных условных или безусловных переходов компилятор должен сохранить любой блок, которого можно достичь, выполнив переход. Это простой и недорогой анализ. Его можно выполнить попутно, в процессе обходов ГПУ для других целей или во время построения ГПУ.

> Если исходный язык допускает арифметические операции с указателями или метками, то компилятор должен сохранять все блоки. В противном случае он может ограничиться сохранением только тех блоков, на метки которых имеются ссылки.

Обработать второй случай труднее. От компилятора требуется умение рассуждать о значениях выражений, управляющих условными переходами. В разделе 10.7.1 представлен алгоритм, который обнаруживает, что некоторые блоки недостижимы, потому что ведущие к ним пути не выполняются.

ИТОГИ РАЗДЕЛА

Преобразования кода часто порождают бесполезный или недостижимый код. Многие преобразования просто оставляют за собой мертвые операции в ПП кода и полагаются на то, что специализированные преобразования типа *Dead* и *Clean* удалят их. Большинство оптимизирующих компиляторов содержат набор преобразований для устранения мертвого кода. Часто такие проходы включаются в последовательность преобразований несколько раз.

Алгоритмы *Dead* и *Clean* аккуратно устраняют бесполезный и недостижимый код. Однако ограничения на точность сопровождающих их видов анализа могут помешать преобразованиям доказать, что какой-то код мертвый. В частности, бесполезный и мертвый код может быть не замечен из-за ограничений на анализ значений указателей и на точность анализа потока управления.

ВОПРОСЫ

1. Опытные программисты часто уверены в том, что не пишут бесполезный или недостижимый код. Какие из рассмотренных в главе 8 преобразований могут порождать бесполезный код?

Указание: напишите код доступа к `A[i,j]`, где массив `A` объявлен как `A[1:N,1:M]`.

2. Как компилятор или компоновщик мог бы обнаружить и устранить недостижимые процедуры? Какую пользу можно извлечь из применения вашей процедуры?

10.3. Перемещение кода

Перемещение вычисления в точку кода, где оно выполняется реже, чем в исходном месте, должно уменьшить полное число выполненных операций в работающей программе. Первое преобразование, представленное в этом разделе, *отложенное перемещение кода*, служит для ускорения выполнения. Поскольку обычно циклы выполняются гораздо больше раз, чем окружающий их код, значительная часть работы в этом направлении призвана вынести из цикла инвариантные относительно него выражения. Отложенное перемещение кода обобщает идеи, первоначально сформулированные для решения задач о доступных выражениях, включив в рассмотрение операции, которые избыточны на некоторых, но не на всех путях. Алгоритм вставляет код, делающий такие выражения избыточными на всех путях, а затем удаляет ставшие избыточными выражения.

Однако некоторые компиляторы оптимизируют, ставя во главу угла другие критерии. Если важен размер исполняемой программы, то компилятор может выполнить перемещение кода, чтобы уменьшить число копий некоторой операции. Второе преобразование, представленное в этом разделе, *поднятие*, использует перемещение кода, чтобы исключить дублирующиеся команды. Оно находит места, в которые можно вставить одну операцию, так чтобы сделать несколько ее копий избыточными.

10.3.1. Отложенное перемещение кода

ОПК работает с ГПУ и ПП, а не с SSA-формой кода. SSA может усложнить алгоритмы перемещения кода. Перемещение операции, определяющей x_i, может потребовать переименования и, быть может, вставки одной или нескольких φ-функций.

Отложенное перемещение кода (ОПК) пользуется анализом потока данных, чтобы найти как сами операции, являющиеся кандидатами на перемещение, так и места, куда их можно поместить. Алгоритм работает с ПП программы и ее ГПУ, а не с формой SSA. Алгоритм решает три разных множества уравнений потока данных и выводит из результатов дополнительные множества. Для каждого ребра ГПУ он порождает множество выражений, которые следует вычислить на этом ребре, а для каждой вершины ГПУ – множество выражений, для которых экспонирован-

ные сверху вычисления следует удалить из соответствующего блока. Простая стратегия переписывания интерпретирует эти множества и модифицирует код.

ОПК объединяет перемещение кода с устранением избыточных и частично избыточных вычислений. Понятие избыточности было введено при обсуждении ЛНЗ и СНЗ в главе 6. Вычисление называется *частично избыточным* в точке p, если оно имеет место на некоторых, но не на всех путях, достигающих p, и ни один из входящих в него операндов не изменяется между этими вычислениями и p.

Избыточность
Выражение e называется *избыточным* в точке p, если оно уже было вычислено на всех путях, ведущих в p.

Частичная избыточность
Выражение e называется *частично избыточным* в точке p, если оно вычисляется на некоторых, но не на всех путях, достигающих p.

На рис. 10.3 показано две причины, по которым выражение может оказаться частично избыточным. На рисунке (a) выражение `b × c` вычисляется на одном пути, ведущем в точку соединения, но не на втором. Чтобы сделать второе вычисление избыточным, ОПК вставляет вычисление `b × c` на другой путь, как показано на рисунке (b). На рисунке (c) `b × c` избыточно на обратном ребре цикла, но не на ребре, которое входит в цикл. В самом цикле оно инвариантно. Вставка вычисления `b × c` перед циклом делает вычисление внутри цикла избыточным, как показано на рисунке (d). Сделав инвариантное относительно цикла вычисление избыточным и устранив его, ОПК выносит его из цикла; эта оптимизация называется также *перемещением кода, инвариантного относительно цикла*.

ОПК перемещает вычисления выражений, а не присваивания. Оно создает имя t_i для хранения `b × c` и переписывает `a ← b × c` в виде `a ← t`$_i$.

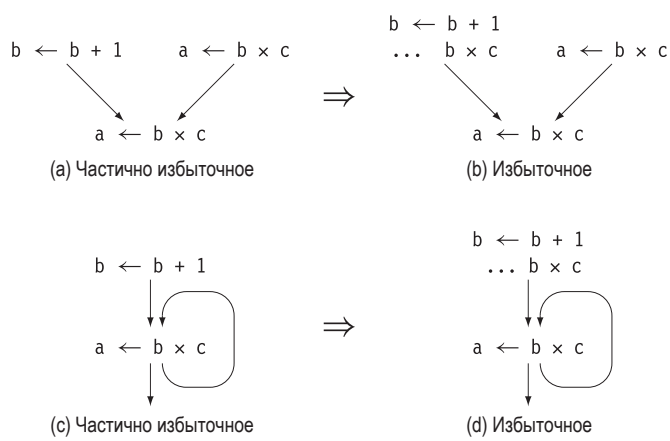


Рис. 10.3 Преобразование частичной избыточности в избыточность

Фундаментальные идеи, лежащие в основе ОПК, были введены в разделе 9.2.4. ОПК вычисляет как доступные, так и предсказуемые выражения. Далее ОПК использует результаты этих видов анализа, чтобы аннотировать

В этом контексте *самая ранняя* означает позицию в ГПУ, ближайшую к входной вершине.

каждое ребро ГПУ $\langle i, j \rangle$ множеством Earliest(i, j), содержащим выражения, для которых это ребро является *самым ранним допустимым размещением*. Затем ОПК решает третью задачу о потоке данных, чтобы найти *поздние размещения*, т. е. ситуации, когда вычисление выражения после самого раннего его размещения дает тот же эффект. Поздние размещения желательны, потому что могут сократить время жизни значений, определенных во вставленных вычислениях. Наконец, ОПК вычисляет свои конечные продукты, два множества Insert и Delete, которые направляют шаг переписывания кода.

Форма кода

Эти правила согласуются с правилами именования регистров в разделе 4.6.1.

ОПК опирается на несколько неявных предположений о форме кода. Текстуально одинаковые выражения всегда определяют одно и то же имя. Таким образом, каждый экземпляр $r_i + r_j$ всегда определяет один и тот же r_k, и алгоритм может использовать r_k в качестве заместителя $r_i + r_j$. Эта схема именования упрощает шаг переписывания; оптимизатор может просто заменить избыточное вычисление $r_i + r_j$ копией, взятой из r_k, вместо того чтобы создавать новое временное имя и вставлять копии в r_k после каждого предшествующего вычисления $r_i + r_j$. Мы добавим ограничение: $i < k$ и $j < k$.

ОПК перемещает вычисления выражения, а не присваивания. Дисциплина именования требует второго правила для переменных программы, потому что они получают значения разных выражений. Таким образом, переменные программы устанавливаются только операциями копирования типа регистр-регистр. Компилятору необходим четкий способ разделить пространство имен между переменными и выражениями.

Наконец, компилятор сначала назначает номера виртуальных регистров переменным, которые хранятся в регистрах, а затем использует бóльшие по величине номера регистров для временных и вычисленных значений. В совокупности эти правила гарантируют, что переменная всегда устанавливается копированием вида $r_i \leftarrow r_j$, где $i < j$. Поэтому присваивания легко отличить от вычисления выражений.

Сочетание этих правил создает четкое разделение между выражениями, которые можно перемещать, и присваиваниями, которые перемещать нельзя. Любая операция, кроме копирования типа регистр-регистр или передачи управления, является вычислением выражения. Любая операция копирования $r_i \Rightarrow r_j$, где $i > j$, является присваиванием: если $i \leq j$, то это вычисление выражения.

```
for i = 1 to n do
   x = a + b * c
```
Пример кода

На рис. 10.4(a) показан ILOC-код простого цикла во врезке. Цикл состоит из одного предложения, вся правая часть которого инвариантна относительно цикла. На рис. 10.4(b) показано отображение имен переменных

на номера регистров; регистры с индексами, большими 9, являются выражениями.

```
B1: loadI  1          ⇒ r10
    i2i    r10        ⇒ r4
    loadAI rarp, @n   ⇒ r11
    i2i    r11        ⇒ r5
    cmp_LE r4, r5     ⇒ r12
    cbr    r12        → B2, B3

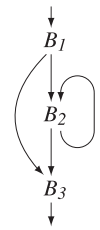
B2: mult   r2, r3     ⇒ r15
    add    r1, r15    ⇒ r16
    i2i    r16        ⇒ r6
    addI   r4, 1      ⇒ r13
    i2i    r13        ⇒ r4
    cmp_GT r4, r5     ⇒ r14
    cbr    r14        → B3, B2

B3: ...
```

(a) Простой цикл

Переменная	a	b	c	i	n	x
Имя ВР	r_1	r_2	r_3	r_4	r_5	r_6

(b) Отображение переменных на виртуальные регистры



(c) Граф управления потоком

Рис. 10.4 Пример отложенного перемещения кода

Задачи о потоке данных, используемые в ОПК, начинаются с вычисления трех разных множеств локальной информации. Для каждого блока b ОПК нуждается в следующих множествах:

DEExpr(b): множество экспонируемых вниз выражений в b;
UEExpr(b): множество экспонированных сверху выражений в b;
ExprKill(b): множество выражений, уничтоженных в b.

Эти множества встречались в разделе 9.2.4. Для программы на рис. 10.4 они выглядят следующим образом:

	B_1	B_2	B_3
DEExpr	$\{r_{10}, r_{11}, r_{12}\}$	$\{r_{14}, r_{15}, r_{16}\}$	$\emptyset$
UEExpr	$\{r_{10}, r_{11}\}$	$\{r_{13}, r_{15}, r_{16}\}$	$\emptyset$
ExprKill	$\{r_{12}, r_{13}, r_{14}\}$	$\{r_{12}, r_{13}, r_{14}\}$	$\emptyset$

Мы показываем множества для B_3 пустыми, потому что в примере весь код в B_3 абстрагируется. В более реалистичных сценариях B_3 содержал бы какой-то код, и множества для B_3 были бы непустыми.

Доступные выражения

На первом шаге ОПК вычисляются доступные выражения, определенные в разделе 9.2.4. ОПК нужна информация о доступности в конце каждого блока – множества AvailOut. Выражение e доступно при выходе из блока b, если

Чтобы упростить изложение, мы предположили, что ГПУ имеет единственную входную вершину n_0 и единственную выходную вершину n_f.

оно вычислялось на любом пути из n_0 в конец b и ни один из его аргументов впоследствии не был переопределен. ОПК может вычислить AvailOut, решив следующие уравнения:

$$\text{AvailIn}(n) = \bigcup_{m \in preds(n)} \text{AvailOut}(m),$$

$$\text{AvailOut}(n) = \left(\begin{array}{l} \text{DEExpr}(n) \cup \\ (\text{AvailIn}(n) \cap \overline{\text{ExprKill}(n)}) \end{array} \right),$$

AvailOut(n) содержит все экспонируемые вниз выражения в n плюс все выражения, доступные в конце всех предшественников n в ГПУ, которые не уничтожаются в n. Начальные значения задаются так:

$\text{AvailIn}(n_0) = \emptyset$,
$\text{AvailOut}(n) = \{$ *все выражения* $\}$ $\forall n \neq n_0$.

Компилятор может воспользоваться стандартным итеративным решателем. Для примера на рис. 10.4 этот процесс порождает следующие множества:

	B_1	B_2	B_3
AvailOut	$\{r_{10}, r_{11}, r_{12}\}$	$\{r_{10}, r_{11}, r_{14}, r_{15}, r_{16}\}$	...

ОПК использует множества AvailOut, чтобы определить возможные места размещения выражений в ГПУ. Если выражение $e \in \text{AvailOut}(b)$, то компилятор может разместить вычисление e в конце блока b и получить результат, порожденный самым недавним вычислением e на любом пути из n_0 в b в ГПУ. Если $e \notin \text{AvailOut}(b)$, то одно из входящих в e подвыражений было модифицировано на одном из этих путей, поэтому вычисление e в конце b могло бы дать другое значение. Таким образом, множества AvailOut показывают компилятору, как далеко вперед в ГПУ он может переместить вычисление e, игнорируя любые использования e.

Предсказуемые выражения

Чтобы собрать информацию для перемещения подвыражений назад, ОПК вычисляет предсказуемость. Напомним (см. раздел 9.2.4), что выражение называется предсказуемым в точке p, если оно вычисляется на каждом пути, исходящем из p, и результаты всех этих вычислений одинаковы. Поскольку ОПК нуждается в информации о предсказуемых выражениях в начале и в конце каждого блока, мы переработали уравнение, введя множество AntIn(n), которое содержит множество предсказуемых выражений для входа в блок, соответствующий вершине ГПУ n:

$$\text{AntOut}(n) = \bigcup_{m \in succs(n)} \text{AntIn}(m), \forall n \neq n_f,$$

$$\text{AntIn}(n) = \left(\begin{array}{l} \text{UEExpr}(n) \cup \\ (\text{AntOut}(n) \cap \overline{\text{ExprKill}(n)}) \end{array} \right).$$

AntIn(n) содержит все экспонированные сверху выражения в n плюс все выражения, предсказуемые в начале каждого преемника n в ГПУ и не уничтоженные в n. Начальные значения задаются так:

$\text{AntOut}(n_f) = \emptyset$,

$\text{AntOut}(n) = \{$ *все выражения* $\}$ $\forall n \neq n_f$.

Для нашего примера решение этих уравнений дает следующие множества:

	B_1	B_2	B_3
AntIn	$\{r_{10}, r_{11}\}$	$\{r_{13}, r_{15}, r_{16}\}$	$\emptyset$
AntOut	$\emptyset$	$\emptyset$	$\emptyset$

AntOut дает информацию о безопасности поднятия вычисления в начало или в конец текущего блока. Если $x \in \text{AntOut}(b)$, то компилятор может поместить вычисление x в конец b, с двумя гарантиями. Во-первых, вычисление в конце b даст то же значение, что следующее вычисление x на любом пути выполнения в процедуре. Во-вторых, на любом пути выполнения, исходящем из b, программа вычислит x до переопределения любого его аргумента.

Самое раннее размещение

Имея решения задач доступности и предсказуемости, компилятор может определить самую раннюю точку в программе, в которой можно вычислить каждое выражение. Чтобы упростить уравнения, ОПК предполагает, что вычисления размещаются на ребре ГПУ, а не в начале или в конце конкретного блока. Вычисление размещения на ребре позволяет компилятору отложить решение о том, вставлять ли вычисления в начале, в конце или в середине ребра (см. обсуждение критических ребер в разделе 9.3.5).

Для ребра ГПУ (i, j) выражение e принадлежит Earliest(i, j) тогда и только тогда, когда компилятор вправе переместить e на (i, j), но не может перенести его ни на какое более раннее ребро. Это условие определяется уравнением Earliest:

$$\begin{aligned} \text{Earliest}(i, j) = {} & \text{AntIn}(j) \cap \overline{\text{AvailOut}(i)} \\ & \cap (\text{ExprKill}(i) \cup \overline{\text{AntOut}(i)}). \end{aligned}$$

В этих терминах самое раннее размещение e определяется следующим образом:

1) $e \in \text{AntIn}(j)$ означает, что компилятор может безопасно переместить e в начало j. Предсказуемость гарантирует, что e породит то же значение, что и его следующее вычисление на любом пути, исходящем из j, и что на каждом из этих путей e вычисляется;

2) $e \notin \text{AvailOut}(i)$ показывает, что никакое предшествующее вычисление e недоступно при выходе из i. Если бы $e \in \text{AvailOut}(i)$, то вставка e на (i, j) была бы избыточной;
3) третье условие кодирует два случая. Если $e \in \text{ExprKill}(i)$, то компилятор не может переместить e через блок i из-за определения в i. Если $e \notin \text{AntOut}(i)$, то компилятор не может переместить e внутрь i, потому что $e \notin \text{AntIn}(k)$ для некоторого ребра (i,k). Если хотя бы одно из этих условий выполнено, то e не может быть перемещено дальше (i, j).

ОПК может игнорировать третий член в $\text{Earliest}(n_0, k)$ для любого k. ОПК не может переместить выражение раньше, чем в n_0. Множества Earliest для нашего сквозного примера выглядят так:

Earliest	$\{r_{13}, r_{15}, r_{16}\}$	$\emptyset$	$\{r_{13}\}$	$\emptyset$

Позднее размещение

Заметим, что $\text{Later}(i, j)$ определено для ребер (i, j), а не для произвольных пар вершин.

Последняя задача о потоке данных в ОПК определяет, когда самое раннее размещение можно отложить до более поздней точки в ГПУ с сохранением результата. Этот вид анализа формулируется как прямая задача о потоке данных на ГПУ, в которой с каждой вершиной ассоциировано множество $\text{LaterIn}(n)$, а с каждым ребром (i, j) – другое множество $\text{Later}(i, j)$. Эта система уравнений определяется следующим образом:

$$\text{LaterIn}(j) = \bigcup_{i \in preds(j)} \text{Later}(i, j), j \neq n_0,$$

$$\text{Later}(i, j) = \text{Earliest}(i, j) \cup (\text{LaterIn}(i) \cap \overline{\text{UEEapr}(i)}).$$

Начальные значения задаются так:

$$\text{LaterIn}(n_0) = \emptyset,$$
$$\text{LaterIn}(n) = \{ \textit{все выражения} \}\ \forall n \neq n_0.$$

Для решения компилятор может воспользоваться стандартным итеративным решателем. У этих уравнений имеется единственная неподвижная точка.

Выражение $e \in \text{LaterIn}(k)$ тогда и только тогда, когда всякий путь, достигающий k, включает ребро (p,q) такое, что $e \in \text{Earliest}(p,q)$ и на пути от q к k операнды e не переопределяются и не встречается такое вычисление e, которое более раннее размещение e могло бы предсказать. Член Earliest в уравнении Later гарантирует, что $\text{Later}(i, j)$ включает $\text{Earliest}(i, j)$. Оставшаяся часть этого уравнения отвечает за помещение e в $\text{Later}(i, j)$, если e можно переместить

вперед от i ($e \in \text{LaterIn}(i)$) и размещение при входе в i не предсказывает использование в i ($e \notin \text{UEExpr}(i)$).

Если известны множества Later и LaterIn, то из $e \in \text{LaterIn}(i)$ следует, что компилятор может переместить вычисление e вперед за i, не теряя никаких преимуществ, т. е. не существует вычисления e в i, которое могло бы предсказать более раннее вычисление, а из $e \in \text{LATER}(i, j)$ следует, что компилятор может переместить вычисление e из i в j.

Для нашего примера эти уравнения порождают следующие множества:

	B_1	B_2	B_3
LaterIn	$\emptyset$	$\{r_{13}\}$	$\emptyset$

	(B_1,B_2)	(B_1,B_3)	(B_2,B_2)	(B_2,B_3)
Later	$\{r_{13}, r_{15}, r_{16}\}$	$\emptyset$	$\{r_{13}\}$	$\emptyset$

Переписывание кода

Последний шаг ОПК – переписать код, воспользовавшись знаниями, полученными в результате вычислений потока данных. Для направления этого процесса в нужную сторону ОПК использует два дополнительных множества, Insert и Delete.

Множество Insert для каждого ребра определяет, какие вычисления ОПК должен вставить на это ребро.

$$\text{Insert}(i, j) = \text{Later}(i, j) \cap \overline{\text{LaterIn}(j)}.$$

Если у i только один преемник, то ОПК может вставить вычисления в конец i. Если у j только один предшественник, то вычисления можно вставить при входе в j. Если ни то, ни другое условие не выполнено, то ребро (i, j) является критическим, и компилятор должен разорвать его, вставив в середину ребра блок для вычисления выражений в $\text{Insert}(i, j)$.

Множество Delete определяет, какие вычисления ОПК должен удалить из блока.

$$\text{Delete}(i) = \text{UEExpr}(i) \cap \overline{\text{LaterIn}(i)},\ i \neq n_0.$$

$\text{Delete}(n_0)$ равно $\emptyset$, потому что у n_0 нет предшественника. Если $e \in \text{Delete}(i)$, то первое вычисление e в i оказывается избыточным после того, как все вставки выполнены. Любое последующее вычисление e в i, у которого имеются экспонированные сверху использования, т. е. операнды не переопределяются между началом i и вычислением, тоже может быть удалено. Поскольку все вычисления e определяют одно и то же имя, компилятору нет нужды переписывать последующие ссылки на удаленное вычисление. Эти ссылки будут естественным образом вести на более ранние вычисления e, порождающие тот же результат.

Для нашего примера множества Insert и Delete просты:

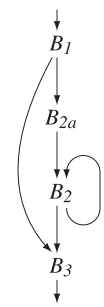
	(B_1,B_2)	(B_1,B_3)	(B_2,B_2)	(B_2,B_3)
Insert	$\{r_{15}, r_{16}\}$	$\emptyset$	$\{r_{13}\}$	$\emptyset$

	B_1	B_2	B_3
Delete	$\emptyset$	$\{r_{15}, r_{16}\}$	$\emptyset$

Компилятор интерпретирует множества Insert и Delete и переписывает код, как показано на рис. 10.5(a). ОПК удаляет из B_2 выражения, которые определяют r_{15} и r_{16}, и вставляет их на ребро (B_1, B_2).

```
B1:  loadI   1          ⇒ r10
     i2i     r10        ⇒ r4
     loadAI  rarp, @n   ⇒ r11
     i2i     r11        ⇒ r5
     cmp_LE  r4, r5     ⇒ r12
     cbr     r12        → B2a, B3
B2a: mult    r2, r3     ⇒ r15
     add     r1, r15    ⇒ r16
     jump               → B2
B2:  i2i     r16        ⇒ r6
     addI    r4, 1      ⇒ r13
     i2i     r13        ⇒ r4
     cmp_GT  r4, r5     ⇒ r14
     cbr     r14        → B3,B2
B3:  ...
```

(a) Преобразованный код



(b) Его ГПУ

Рис. 10.5 Пример после отложенного перемещения кода

Поскольку у B_1 два преемника, а у B_2 два предшественника, (B_1, B_2) является критическим ребром. Поэтому ОПК разбивает это ребро и создает новый блок B_{2a} для размещения вставленных вычислений r_{15} и r_{16}. Разбиение (B_1, B_2) добавляет в код команду `jump`. Алгоритмы генерирования кода и размещения блоков должны постараться разместить B_{2a} так, чтобы безусловный переход в конце блока стал неявной ветвью.

Отметим, что ОПК оставляет копирование, определяющее r_6, в B_2. (Напомним, что в r_6 хранится переменная `x`.) ОПК перемещает выражения, а не присваивания. Если копирование не нужно, то последующее объединение операций копирования – в распределителе регистров или на отдельном проходе – должно обнаружить этот факт и устранить лишнюю операцию.

10.3.2. Поднятие кода

Технику перемещения кода можно также использовать для уменьшения размера откомпилированного кода. *Поднятие кода* – прямой путь к достижению этой цели. В нем результаты анализа предсказуемости используются особенно простым способом.

Если выражение $e \in \text{AntOut}(b)$ для некоторого блока b, значит, e вычисляется на любом пути, исходящем из b, и, стало быть, вычисление e в конце b сделало бы первое вычисление на любом пути избыточным. Чтобы уменьшить размер кода, компилятор может вставить вычисление e в конец b и заменить первое появление e на каждом пути, исходящем из b, ссылкой на ранее вычисленное значение. Результатом такого преобразования является замена нескольких копий вычисления e одной, что уменьшает общее число операций в откомпилированном коде.

Уравнения AntOut гарантируют, что ни один операнд e не переопределяется на участке между концом b и следующим вычислением на любом пути, исходящем из b.

Некоторые авторы используют для AntOut обозначение VeryBusy, когда речь идет о поднятии.

Чтобы заменить эти выражения напрямую, компилятору необходимо их найти. Он мог бы вставить e, затем решить еще одну задачу о потоке данных, доказав, что на пути от b к некоторому вычислению e нет переопределений операндов e. Альтернативно он мог бы пройти по каждому пути, исходящему из b, чтобы найти первый блок, в котором e определяется, – заглядывая в множество UEExpr блока. Оба подхода выглядят слишком сложными.

Более простой для компилятора подход – посетить каждый блок b и вставить вычисление e в конец b для каждого выражения $e \in \text{AntOut}(b)$. Если компилятор применяет единообразную дисциплину именования, как было предложено в обсуждении ОПК, то каждое вычисление определит подходящее имя. Затем можно применить ОПК или СНЗ, чтобы удалить вновь появившиеся избыточные выражения.

ИТОГИ РАЗДЕЛА

Компиляторы выполняют перемещение кода по двум основным причинам. Перемещение операции в точку, где она выполняется меньшее число раз, чем в оригинальной позиции, должно уменьшить время выполнения. Перемещение операции в точку, где один экземпляр заменяет несколько, должно уменьшить размер кода. В этом разделе были представлены примеры обеих ситуаций.

Алгоритм ОПК – классический пример глобальной оптимизации, управляемой потоком данных. Он находит избыточные и частично избыточные выражения, вычисляет лучшее место для них и перемещает туда. По определению, любое инвариантное в цикле выражение либо избыточно, либо частично избыточно; ОПК выносит из цикла большой класс инвариантных выражений. Поднятие – гораздо более простой подход; оно находит операции, избыточные на каждом пути, исходящем из некоторой точки p, и заменяет избыточные вхождения одним вычислением в p. Таким образом, поднятие обычно применяется для уменьшения размера кода.

ВОПРОСЫ

1. Поднятие обнаруживает ситуацию, в которой некоторое выражение e встречается на каждом пути, исходящем из точки p, и все эти вхождения можно безопасно заменить вычислением e в p. Опишите симметричную и эквивалентную оптимизацию, *спуск кода*, которая обнаруживает ситуацию, когда несколько вычислений выражения можно безопасно переместить вперед – из точек, предшествующих p, в p.

Одна из форм спуска кода называется *перекрестным переходом* (cross jumping).

2. Рассмотрите, что произойдет, если применить ваше преобразование спуска кода на этапе компоновки, когда известен код всего приложения. Какие возможности оно сможет найти в коде связывания процедур?

10.4. Специализация

В большинстве компиляторов фаза анализа определяет форму ПП программы еще до начала детального анализа кода. По необходимости при этом порождается общий код, работающий в любом контексте, который может встретиться в исполняемой программе. Однако анализ зачастую позволяет компилятору собрать достаточно информации, чтобы сузить множество контекстов, в которых должна работать программа. Затем компилятор может специализировать последовательность операций в соответствии со своим знанием контекста, в котором будет исполняться код.

Важные методы специализации встречаются в других разделах этой книги. Распространение констант, описанное в разделах 9.3.6 и 10.7.1, анализирует процедуру на предмет поиска переменных и выражений, имеющих известные постоянные значения; затем эти значения подставляются напрямую в вычисление. Межпроцедурное распространение констант, описанное в разделе 9.4.2, применяет те же идеи ко всей программе. Снижение стоимости операций, представленное в разделе 10.7.2, заменяет индуктивные последовательности дорогостоящих вычислений эквивалентными последовательностями более быстрых операций. Оконная оптимизация, с которой мы познакомимся в разделе 11.3, применяет сопоставление с образцом к коротким последовательностям команд, чтобы найти возможность локального улучшения. Нумерация значений, описанная в разделах 8.4.1 и 8.5.1, систематически упрощает ПП кода путем применения алгебраических тождеств и локального свертывания констант. Каждый из этих методов является формой специализации.

Оптимизирующие компиляторы опираются на эти общие методы для улучшения кода. Дополнительно большинство оптимизирующих компиляторов включают методы специализации, ориентированные на конкретные свойства исходных языков, с которыми ожидает встретиться автор компилятора. Далее в этом разделе представлены три таких метода, нацеленных на неэффективности вызова процедур: оптимизация хвостового вызова, оптимизация листового вызова и перенос параметров.

10.4.1. Оптимизация хвостового вызова

Когда последним действием в процедуре является вызов, мы говорим, что имеет место хвостовой вызов. Компилятор может специализировать хвостовые вызовы с учетом контекста способами, которые устраняют значительную часть накладных расходов, подразумеваемых соглашением о вызове процедур. Чтобы понять, как возникает возможность улучшения, рассмотрим,

что происходит, когда *o* вызывает *p*, а *p* вызывает *q*. Возвращая управление, *q* выполняет команды эпилога и совершает переход на заключительную последовательность *p*. Выполнение продолжается в *p* до момента возврата, когда *p* выполняет свой эпилог и переходит на заключительную последовательность *o*.

Если вызов *q* из *p* является хвостовым, то между заключительной последовательностью и эпилогом *p* нет никаких полезных вычислений. Следовательно, любой код, который сохраняет и восстанавливает состояние *p*, сверх того, что необходимо для возврата из *p* в *o*, бесполезен. Стандартное соглашение о связях, описанное в разделе 6.5, тратит много усилий на сохранение состояния, которое бесполезно в контексте хвостового вызова.

Этот контекст понуждает код эпилога *q* вернуть управление прямо *o*.

В точке вызова *q* из *p* минимальная предварительная последовательность должна вычислять фактические параметры и при необходимости модифицировать связи доступа или дисплей. Ей не нужно сохранять никакие регистры, сохраняемые вызывающей стороной, потому что они не могут быть живыми. Не нужно также выделять память для новой ЗА, потому что *q* может использовать ЗА *p* при условии, что область локальных данных *p* достаточно велика для *q*. (Если нет, то компилятор может в прологе *p* выделить большую область данных.) Она должна оставить неизменным контекст, созданный для возврата в *o*, а именно адрес возврата и УЗА, которые *o* передала *p*, а также все сохраняемые вызываемой стороной регистры, которые *p* сохранила в своей собственной ЗА. Наконец, предварительная последовательность должна выполнить безусловный переход на подправленный пролог *q*.

В этой схеме *q* нуждается в специальном прологе, соответствующем специальной предварительной последовательности в *p*. Он должен сохранять только те части состояния *p*, которые позволяют вернуться в *o*. Пролог *q* не сохраняет регистров, обычно сохраняемых вызываемой стороной, по двум причинам. Во-первых, значения, помещенные *p* в эти регистры, мертвы. Во-вторых, сохраняемые вызываемой стороной значения, которые *p* сохранила, должны быть восстановлены при возврате из *q* в *o*. Таким образом, специальный пролог *q* должен просто инициализировать локальные переменные и значения, которые необходимы *q*, и совершить переход на код *q*.

С такими изменениями предварительной последовательности *p* и пролога *q* хвостовой вызов избегает сохранения и восстановления состояния *p*, что устраняет большинство накладных расходов на вызов. Конечно, после того как предварительная последовательность *p* была так модифицирована, последовательности *p* становятся недоступны. Стандартные алгоритмы типа *Dead* и *Clean* не обнаруживают этот факт, потому что предполагают, что межпроцедурные безусловные переходы на их метки выполняются. Когда оптимизатор модифицирует вызов, он может заодно устранить эти мертвые последовательности.

При некоторой осторожности оптимизатор может организовать операции в модифицированном прологе *q* так, чтобы они были последними операция-

ми в ее более общем прологе. В такой схеме хвостовой вызов q из p просто совершает безусловный переход в точку, расположенную дальше в прологе, чем сделал бы нормальный вызов из другой какой-то процедуры.

Если хвостовой вызов является авторекурсивным, т. е. p и q – одна и та же процедура, то оптимизация хвостового вызова может породить особенно эффективный код. В этом случае вся предварительная последовательность сводится к вычислению аргументов и обратному переходу на начало процедуры. Окончательный возврат из рекурсии требует одного условного перехода, а не по одному переходу на каждый рекурсивный вызов. Получающийся код по эффективности может соперничать с традиционным циклом.

10.4.2. Оптимизация листового вызова

Часть накладных расходов, связанных с вызовом процедуры, проистекает из необходимости подготовиться к вызовам, которые может выполнить вызываемая сторона. Процедура, в которой таких вызовов нет, называется листовой и открывает дополнительные возможности для оптимизации. Компилятор легко может распознать листовую процедуру по отсутствию в ней операций вызова.

Другая причина сохранять адрес возврата состоит в том, чтобы дать последующим инструментам возможность раскрутить стек. Если компилятор ожидает применения таких инструментов, то может оставить операцию сохранения.

В процессе трансляции листовой процедуры компилятор может избежать вставок операции, единственная цель которых – подготовить почву для последующих вызовов. Например, код пролога процедуры может копировать адрес возврата из регистра в специальную позицию 3A. Это действие излишне, если сама процедура не выполняет других вызовов. Если регистр, в котором хранится адрес возврата, нужен для какой-то другой цели, то распределитель регистров может выгрузить его значение в память. Аналогично, если реализация применяет дисплей для обеспечения адресуемости нелокальных переменных, как описано в разделе 6.4.3, то листовая процедура может избежать обновления дисплея в прологе.

Распределитель регистров должен в листовой процедуре сначала использовать сохраняемые вызывающей стороной регистры, а только потом сохраняемые вызываемой стороной. В той мере, в какой он может избежать сохраняемых вызываемой стороной регистров, он может устранить операции сохранения и восстановления в прологе и эпилоге. В небольших листовых процедурах компилятору, возможно, удастся избежать любого использования сохраняемых вызываемой стороной регистров. Если компилятор имеет доступ и к вызывающей, и к вызываемой стороне, то может поступить еще лучше; для листовых процедур, которым нужно меньше регистров, чем входит в множество сохраняемых вызывающей стороной, он может отказаться также от некоторых операций сохранения и восстановления на вызывающей стороне.

Кроме того, в листовых процедурах компилятор может избежать накладных расходов на выделение памяти для записи активации. В реализации, где ЗА размещаются в куче, эти расходы могут быть значительными. В приложении с одним потоком управления компилятор может статически выделять память для ЗА любой листовой процедуры. Более агрессивный компилятор мог бы выделить одну статическую ЗА, достаточно большую для работы с любой листовой процедурой, и сделать так, что все листовые процедуры будут использовать эту ЗА.

Если компилятор имеет доступ и к листовой процедуре, и к вызывающим ее сторонам, то может выделить место для ЗА листовой процедуры в ЗА каждой из вызывающих ее сторон. Эта схема распределяет затраты на выделение памяти для ЗА между как минимум двумя вызовами: вызывающей стороны и листовой процедуры. Если вызывающая сторона вызывает листовую процедуру несколько раз, то экономия возрастает.

10.4.3. Перенос параметров

Перенос
категория преобразований, которые перемещают неоднозначное значение в локальное скалярное имя, чтобы сделать его видимым распределителю регистров.

Неоднозначные ссылки на память мешают компилятору сохранять значения в регистрах. Иногда компилятор может воспользоваться результатами специального анализа ветвлений, анализа значений указателей или анализа индексных выражений и доказать, что потенциально неоднозначное значение на самом деле ссылается только на один адрес в памяти. В таких случаях компилятор может переписать код, переместив это значение в скалярную локальную переменную, которую распределитель регистров может хранить в регистре. Такой вид преобразований часто называют *переносом* (promotion). Анализ переноса ссылок на массивы или значений указателей выходит за рамки этой книги. Однако же преобразования можно неплохо проиллюстрировать и на более простом примере.

Параметры, передаваемые по ссылке, могут быть неоднозначными. В месте вызова одна и та же переменная может передаваться в двух или более параметрах. В качестве фактического параметра может быть передана глобальная переменная. Если компилятор не выполняет межпроцедурный анализ для исключения таких возможностей, то внутри вызываемой стороны должен рассматривать каждый формальный параметр, передаваемый по ссылке, как неоднозначный. Потенциальная неоднозначность заставляет компилятор хранить значение параметра в памяти, а не в регистре.

Если компилятор может доказать, что фактический параметр обязательно является однозначным на вызываемой стороне, то может перенести значение параметра. Он может скопировать значение во временное имя в начале процедуры и обратно перед возвратом из нее. Затем временное имя может конкурировать за регистры наравне с другими однозначными скалярными

Компилятор может создать новое имя для хранения перенесенного значения: локальное скалярное значение в случае ПП, близком к исходному коду, или виртуальный регистр в случае низкоуровневого ПП.

значениями. Если вызываемая сторона гарантированно не модифицирует параметр, то компилятор мог бы преобразовать параметр в передаваемый по значению. Чтобы применить это преобразование к процедуре *p*, оптимизатор должен найти все места, из которых может вызываться *p*. Он может либо доказать, что преобразование применимо ко всем этим местам, либо клонировать *p*, создав копию для обработки перенесенных значений (см. раздел 10.6.2).

ИТОГИ РАЗДЕЛА

В категорию специализации попадает много методов, которые адаптируют вычисления общего назначения к конкретным контекстам. В других главах и разделах представлены мощные методы глобальной и региональной специализаций, такие как распространение констант, локальная оптимизация и снижение стоимости операций.

В этом разделе мы сосредоточились на оптимизациях, которые компилятор может применить к коду, реализующему вызовы процедур. Оптимизация хвостовых вызовов – ценный инструмент, который преобразует хвостовую рекурсию в форму, соперничающую по эффективности с традиционной итерацией; она применяется также к нерекурсивным хвостовым вызовам. Листовые процедуры предлагают специальные возможности для улучшения, потому что вызываемая сторона может опускать значительные части, требуемые стандартным соглашением о связи. Перенос параметров – пример класса важных преобразований, устраняющих неэффективности, связанные с неоднозначными ссылками.

ВОПРОСЫ

1. Перенос параметров преобразует код, перемещая неоднозначный параметр, переданный по ссылке, в локальную скалярную переменную. Аналогичное преобразование могло бы проанализировать цикл и перенести указательное значение (например, *p в C) в локальную скалярную переменную, используемую на всем протяжении цикла. Какие условия безопасности необходимы компилятору, чтобы доказать возможность такого переноса указательных значений в скаляры?
2. Альтернативой оптимизациям вызовов процедур, рассмотренным в этом разделе, могло бы стать встраивание. Как бы вы применили встраивание в каждом случае? Как компилятору выбрать более целесообразный вариант?

10.5. Устранение избыточности

Вычисление $x + y$ избыточно в некоторой точке кода p, если на каждом пути, достигающем p, $x + y$ уже было вычислено и ни x, ни y не модифицировались после вычисления. Избыточные вычисления обычно являются артефактами трансляции или оптимизации.

Мы уже рассмотрели три эффективных метода устранения избыточности: локальная нумерация значений (ЛНЗ) в разделе 8.4.1, суперлокальная

нумерация значений (СНЗ) в разделе 8.5.1 и отложенное перемещение кода (ОПК) в разделе 10.3.1. Эти алгоритмы варьируются от простых и быстрых (ЛНЗ) до сложных и всесторонних (ОПК). И хотя все три метода применяются к разным областям, главное различие между ними – способ, которым они устанавливают идентичность двух значений. В разделе 10.5.1 этот вопрос рассматривается подробно. В разделе 10.5.2 представлена еще одна версия нумерации значений, основанная на доминировании.

10.5.1. Идентичность значений и идентичность имен

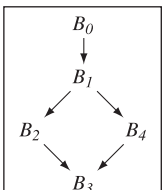
Алгоритм ЛНЗ ввел простой механизм доказательства того, что два выражения принимают одинаковые значения. ЛНЗ основан на двух принципах. Он назначает каждому значению уникальное идентифицирующее число – его номер значения. Предполагается, что два выражения порождают одно и то же значение, если в них используется один и тот же оператор и номера значений операндов одинаковы. Эти простые правила позволяют ЛНЗ находить широкий класс избыточных операций – любая операция, порождающая уже существующий номер значения, избыточна.

Опираясь на эти два правила, ЛНЗ может доказать, что $2 + a$ имеет то же значение, что $a + 2$ или $2 + b$, коль скоро у a и b одинаковые номера значений. Но он не может доказать, что выражения $a + a$ и $2 \times a$ имеют одинаковые значения, потому что в них различны операторы. Аналогично нельзя доказать, что $a + 0$ и a имеют одинаковые значения. Поэтому мы дополнили ЛНЗ алгебраическими тождествами, позволяющими обработать многие корректно определенные случаи, не подпадающие под оригинальное правило. На рис. 8.3 приведены тождества, которые ЛНЗ умеет обрабатывать. СНЗ обобщает оптимизацию на более широкие области.

С другой стороны, ОПК полагается на имена, чтобы доказать, что два значения имеют один и тот же номер. Видя выражения $a + b$ и $a + c$, ОПК предполагает, что их значения различны, потому что различны имена b и c. Он пользуется лексическим сравнением – тожественностью имен. Лежащий в основе этого алгоритма анализ потока данных не может непосредственно работать с понятием тождества значений; задачи о потоке данных имеют дело с предопределенным пространством имен и распространяют факты об этих именах по ГПУ. Тот вид ситуативных сравнений, который применяется в ЛНЗ, не укладывается в каркас потока данных.

Компилятор может повысить эффективность ОПК, закодировав тождественность имен в пространство имен кода еще перед применением ОПК. Сначала компилятор применяет технику нумерации значений, например НЗД из следующего подраздела, чтобы определить номера значений. Затем он переписывает код в терминах номеров значений. И наконец, применяет ОПК к преобразованному коду. В получившемся коде найдут отражение сильные стороны обоих подходов, он сможет находить возможности улучшения, ускользающие от каждого подхода по отдельности.

10.5.2. Нумерация значений на основе доминирования



Фрагмент ГПУ

В главе 8 было описано два алгоритма нумерации значений, ЛНЗ и СНЗ. И хотя СНЗ находит больше избыточностей, чем ЛНЗ, он все же пропускает некоторые вещи, потому что распространяет информацию по путям только через расширенные простые блоки (РПБ). Например, для фрагмента ГПУ во врезке СНЗ будет обрабатывать пути $\langle B_0, B_1, B_2\rangle$ и $\langle B_0, B_1, B_4\rangle$. Следовательно, он оптимизирует B_2 и B_4 в контексте префиксного пути $\langle B_0, B_1\rangle$. Поскольку B_3 образует свой собственный вырожденный РПБ, СНЗ оптимизирует B_3 без предшествующего контекста. СНЗ не может распространить информацию через точку соединения в B_3.

С алгоритмической точки зрения, СНЗ начинает каждый блок таблицей, которая включает результаты всех предшественников на пути к РПБ. Блок B_3 образует вырожденный РПБ без предшественников, поэтому не имеет предшествующего контекста. Чтобы улучшить эту ситуацию, мы должны ответить на вопрос: на какое состояние мог бы положиться B_3?

B_3 не может полагаться на значения, вычисленные в B_2 или B_4, потому что ни один не лежит на любом пути, достигающем B_3. С другой стороны, B_3 может полагаться на значения, вычисленные в B_0 и B_1, потому что они встречаются на любом пути, достигающем B_3. Таким образом, мы могли бы дополнить нумерацию значений для B_3 информацией о вычислениях в B_0 и B_1. Однако мы должны учитывать влияние присваиваний в промежуточных блоках, B_2 и B_4.

Рассмотрим выражение $x + y$, которое встречается в конце B_1 и еще раз в начале B_3. Если ни B_2, ни B_4 не переопределяет ни x, ни y, то вычисление $x + y$ в B_3 избыточно, и оптимизатор может повторно использовать значение, вычисленное в B_1. С другой стороны, если в каком-нибудь из этих блоков переопределяется x или y, то результатом вычисления $x + y$ в B_3 будет значение, отличное от вычисленного в B_1, и вычисление не является избыточным. Нам нужен алгоритмический способ различить эти два случая.

У этого обсуждения есть одно тонкое следствие. Задачи о потоке данных, в которых используются множества Kill, например VarKill или ExprKill, можно переформулировать в контексте SSA-имен и тем самым исключить эти множества.

Решить получающуюся систему, возможно, будет значительно дешевле.

По счастью, в пространстве SSA-имен как раз такое различие и закодировано. В форме SSA имя, которое используется в некотором блоке B_i, может попасть в B_i только двумя способами. Либо имя определено φ-функцией в начале B_i, либо оно определено в каком-то блоке, доминирующем над B_i. Таким образом, присваивание x в B_2 или B_4 создает новое имя для x и приводит к вставке φ-функции для x в начало B_3. Эта φ-функция создает новое SSA-имя для x, а процесс переименования изменяет SSA-имя, используемое при последующем вычислении $x + y$ в B_3. Таким образом, форма SSA кодирует присутствие

или отсутствие промежуточного присваивания в B_2 или B_4 непосредственно в пространстве имен. Алгоритм нумерации значений с использованием SSA-имен может избежать осложнений, вызванных этими промежуточными присваиваниями.

Другой важный вопрос, на который мы должны ответить, прежде чем сможем обобщить СНЗ: если задан блок, например B_3, то как найти самого ближнего предшественника, содержащего информацию, которую алгоритм может использовать? Отношение IDom, которое обсуждалось в разделе 9.2.1, именно это свойство и отражает. B_1 принадлежит IDom(B_3), так что это ближайшая вершина, которая встречается на любом пути из входной вершины в B_3. Для блока b алгоритм может воспользоваться таблицами из IDom(b) в качестве начального состояния.

SSA-имя – единственная ссылка на результат одной точки определения.

Метод нумерации значений на основе доминирования (НЗД) основан на идеях СНЗ. В нем используется связка таблиц для отображения выражений на номера значений. Он создает новую таблицу для каждого блока и отбрасывает ее, когда она больше не нужна. Чтобы упростить внутреннюю кухню, НЗД просто использует SSA-имена в качестве номеров значений. Таким образом, если $t_k \leftarrow a_i \times b_j$ – первое вычисление $a_i \times b_j$, то номер значения $a_i \times b_j$ будет равен t_k.

На рис. 10.6 показан сам алгоритм. Он имеет вид рекурсивной процедуры, которую оптимизатор вызывает для входного блока процедуры. Он следует как ГПУ процедуры, представленному своим деревом доминирования, так и потоку значений в форме SSA. Для каждого блока B НЗД выполняет три шага: обрабатывает все φ-функции, существующие в B, сопоставляет номера значений присваиваниям, а затем распространяет информацию в преемники B и рекурсивно вызывает себя для потомков B в дереве доминирования.

Обработка φ-функций в B

НЗД должен сопоставить каждой φ-функции p номер значения. Если p бессмысленна, т. е. все ее аргументы имеют один и тот же номер значения, то НЗД полагает ее номер значения равным номеру значения одного из ее аргументов и удаляет p. Если p избыточна, т. е. порождает тот же номер значения, что и какая-то другая φ-функция q в B, то НЗД сопоставляет p номер значения q. Затем НЗД удаляет p.

В противном случае φ-функция вычисляет новое значение. НЗД сопоставляет ему номер значения – SSA-номер ее цели. Если код достигает этой точки, то либо у φ-функции уникальная комбинация аргументов, либо у некоторых ее аргументов еще нет номера значения. В любом случае НЗД не может упростить эту φ-функцию.

```
DVNT(B)
  выделить память под новую таблицу для B и включить ее в связку
  for каждой φ-функции p вида "n ← φ(...)" в B do
    if p бессмысленна then
      VN[n] ← номер значения для первого аргумента p
      удалить p из кода
    if p избыточна из-за какой-то φ-функции q в B then
      VN[p] ← VN[q]
      удалить p из кода
    else
      VN[n] ← n
      добавить p в хеш-таблицу с номером значения n

    for всех присваиваний "x ← y op z" в B по порядку do
      заменить y на VN[y]
      заменить z на VN[z]
      положить expr ← "y op z"
      if expr можно упростить до expr' then
        переписать присваивание в виде "x ← expr'"
        expr ← expr'
      if expr имеет номер значения v в хеш-таблице then
        VN[x] ← v
        удалить присваивание из кода
      else
        VN[x] ← x
        добавить expr в хеш-таблицу с номером значения x

    for каждого преемника s блока B do
      скорректировать входы φ-функции в s
    for каждого сыновнего узла c блока B в дереве доминирования
      DVNT(c)
    освободить память, выделенную для таблицы в B
```

Рис. 10.6 Метод нумерации значений на основе доминирования

Обработка присваиваний в B

Напомним, что алгоритм построения формы SSA не допускает неинициализированных имен.

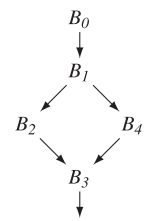
НЗД обходит присваивания в *B* и обрабатывает их так же, как в ЛНЗ или СНЗ. С использованием SSA-имен в качестве номеров значений связана одна тонкость. Встретив предложение *x* ← *y op z*, алгоритм может просто заменить *y* на *VN[y]*, потому что имя в *VN[y]* содержит то же самое значение, что *y*.

Распространение информации в преемники B

Обратите внимание на сходство между этим шагом и соответствующим шагом на этапе переименования при построении формы SSA.

Обработав все φ-функции и присваивания в *B*, НЗД посещает каждого преемника *B* в ГПУ, *s*, и обновляет аргументы φ-функций, которые соответствуют зна-

чениям, перетекающим по ребру (B, s). Он запоминает текущий номер значения для аргумента φ-функции путем замены SSA-имени аргумента. Затем алгоритм рекурсивно вызывает себя для каждого сыновнего узла B в дереве доминирования. Наконец, он освобождает хеш-таблицу, которая использовалась в B.



В этой рекурсивной схеме применения НЗД дерево доминирования обходится в прямом порядке, это гарантирует, что необходимые таблицы строятся перед посещением блока. Такой порядок обхода может противоречить интуиции; для ГПУ, изображенного справа, алгоритм мог бы посетить B_3 раньше B_2 и B_4. Поскольку единственные факты, которые алгоритм может использовать в B_3, – те, что были установлены во время обработки B_0 и B_1, относительный порядок B_2, B_4 и B_3 не только не задан, но и не существен.

ИТОГИ РАЗДЕЛА

Устранение избыточности подразумевает, что повторно использовать значение быстрее, чем заново вычислять его. Алгоритмы ОПК и НЗД находят избыточные вычисления и устраняют дублирующиеся. В ОПК используется тождественность на основе имен, а в НЗД – тождественность на основе значений. Из-за различных определений тождественности алгоритмы могут находить разные избыточности.

И ОПК, и НЗД устраняют избыточные вычисления. ОПК устраняет *вычисление* избыточных и частично избыточных выражений, но не устраняет присваивания. Алгоритм нумерации значений устраняет также присваивания, но не находит частичные избыточности. Чтобы объединить сильные стороны обоих методов, некоторые компиляторы используют нумерацию значений, чтобы закодировать тождественность значений в пространство имен, и технику на основе имен, например ОПК, при переписывании кода.

ВОПРОСЫ

1. НЗД не распространяет значение вдоль ребра, замыкающего цикл, – обратного ребра в графе вызовов. ЛНЗ распространяет информацию по таким ребрам. Напишите несколько примеров избыточных выражений, которые может найти истинно «глобальный» метод типа ЛНЗ, но не может НЗД.
2. Алгоритм НЗД напоминает фазу переименования в алгоритме построения формы SSA. Можете ли вы переформулировать фазу переименования, так чтобы она нумеровала значения, как переименовывает значения? Какое влияние такое изменение оказало бы на размер формы SSA процедуры?

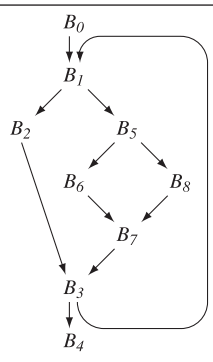
10.6. Обеспечение других преобразований

Часто оптимизатор включает проходы, главная цель которых – создать или раскрыть возможности для других преобразований. В одних случаях преобразование изменяет форму кода, чтобы сделать ее более пригодной для

оптимизации. В других случаях преобразование создает в коде точку, где выполняются условия, при которых другое преобразование становится безопасным или целесообразным. Явно создавая нужную форму кода, такие обеспечивающие преобразования снижают чувствительность оптимизатора к форме исходной программы.

Несколько обеспечивающих преобразований описаны в других частях книги. Большая часть преимуществ, свойственных разворачиванию цикла (раздел 8.5.2) и встраиванию (раздел 8.7.1), проистекают из создания контекста для других оптимизаций. (В каждом случае преобразование устраняет какие-то накладные расходы, но его основной эффект состоит в возможности последующего применения других оптимизаций.) Алгоритм балансировки деревьев по высоте (раздел 8.4.2) не устраняет никаких операций, но создает форму кода, которая может улучшить результаты планирования команд. В этом разделе представлено четыре обеспечивающих преобразования: *клонирование суперблока*, *клонирование процедуры*, *расчленение цикла* и *переименование*.

10.6.1. Клонирование суперблока



ГПУ для клонирования суперблока

Часто способность оптимизатора преобразовать код ограничена зависящей от пути информацией. Представьте себе СНЗ на ГПУ, показанном во врезке. Тот факт, что у блоков B_3 и B_7 несколько предшественников, может помешать оптимизатору улучшить эти блоки. Если, например, в B_6 переменной x было присвоено значение 7, а в B_8 ей же было присвоено значение 13, то при использовании x в B_7 будет казаться, что она получила значение $\perp$, хотя на самом деле ее значение известно и предсказуемо на любом пути, ведущем в B_7.

При таких обстоятельствах компилятор может клонировать блоки, чтобы создать код, лучше приспособленный для преобразования. В данном случае он мог бы создать две копии B_7, скажем B_{7a} и B_{7b}, и перенаправить входящие ребра, проведя вместо них $\langle B_6, B_{7a}\rangle$ и $\langle B_8, B_{7b}\rangle$. После такого изменения оптимизатор мог бы распространить значение 7 переменной x в B_{7a}, а значение 13 – в B_{7b}.

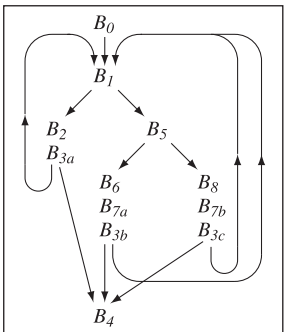
Обратное ребро
ребро ГПУ (i, j) такое, что номер i в порядке обхода в глубину меньше или равен номеру j.

Ребро, замыкающее цикл
ребро ГПУ (i, j) такое, что $j \in \text{Dom}(i)$.

В качестве дополнительного бонуса, поскольку у B_{7a} и B_{7b} по одному предшественнику, компилятор может объединить блоки и создать из B_6 и B_{7a} один блок, а из B_8 и B_{7b} – другой. При таком изменении исключаются безусловные переходы, завершающие блоки B_6 и B_8, и потенциально создается почва для последующих улучшений в ходе оптимизации.

Но возникает проблема: когда компилятор должен прекратить клонирование? Метод *клонирования суперблока* широко применяется, чтобы создать дополнительный контекст для планирования команд. В этом случае оптимизатор отправляется от заголовка цикла – точки входа в цикл – и клонирует все пути, пока не наткнется на обратное ребро.

Применительно к нашему примеру ГПУ этот метод порождает модифицированный ГПУ, показанный во врезке. B_1 – заголовок цикла. У каждой вершины, принадлежащей телу цикла, один предшественник. Если компилятор применит суперлокальную оптимизацию (основанную на РПБ), то каждый найденный им путь будет охватывать одну итерацию тела цикла. (Для нахождения более длинных путей оптимизатору, возможно, пришлось бы развернуть цикл перед клонированием суперблока.)



ГПУ после клонирования суперблока

Клонирование суперблока обеспечивает другие оптимизации тремя основными способами:

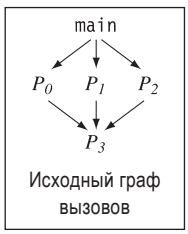
1) *создаются более длинные блоки.* Чем длиннее блок, тем больше контекста доступно локальной и суперлокальной оптимизациям. ЛНЗ без труда обобщается с блока на бóльшие регионы, но о других оптимизациях такого не скажешь. В преобразованном ГПУ локальные методы можно применить непосредственно к объединенным блокам. Например, компилятор мог бы применить алгоритм балансировки дерева по высоте к $\langle B_6, B_{7a}, B_{3b}\rangle$, вместо того чтобы балансировать $\langle B_6, B_{7a}\rangle$ и B_{3b} по отдельности;
2) *устраняются условные переходы.* Объединение двух блоков устраняет условный переход между ними. Переходы требуют времени. Кроме того, они могут создавать помехи критическим для производительности механизмам процессора, в частности механизму предвыборки команд. Общий эффект – уменьшение числа выполняемых операций и повышение эффективности аппаратных механизмов предсказания;
3) *создаются точки, в которых возможна оптимизация.* Устраняя точку соединения в ГПУ, клонирование создает в программе точку, где компилятор может вывести более точные знания о контексте времени выполнения. Преобразованный код может открыть новые возможности для оптимизации, которых не было в исходном коде.

Разумеется, у клонирования есть цена. Оно дублирует отдельные операции, что приводит к увеличению размера кода. Увеличенный код может работать быстрее, потому что обходится без переходов, завершающих блоки. Но может работать и медленнее, если станет больше непопаданий в кеш. В приложениях, где пользователю важнее размер кода, чем скорость выполнения, клонирование суперблока может оказаться контрпродуктивным.

10.6.2. Клонирование процедуры

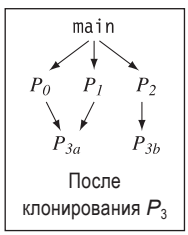
Встраивание, описанное в разделе 8.7.1, дает эффекты, похожие на клонирование суперблока. Для вызова q из p оно создает уникальную копию q и объединяет ее с местом вызова в p. При этом достигаются такие же результаты, как при клонировании суперблока, включая специализацию для конкретного контекста, устранение некоторых операций управления потоком и увеличение размера кода.

В некоторых случаях клонирование процедуры может обеспечить некоторые преимущества встраивания при меньшем увеличении размера кода. Ключевое наблюдение – то, что для клонирования процедуры не нужно создавать отдельный клон для каждого места вызова. Достаточно создать по клону для каждого интересного контекста оптимизации. Компилятор создает несколько копий вызываемой стороны и сопоставляет вызовам с похожим контекстом один и тот же клон.



Исходный граф вызовов

Рассмотрим, к примеру, простой граф вызовов, показанный во врезке. Предположим, что поведение P_3 сильно зависит от одного из ее входных параметров; для значения, равного 1, компилятор может сгенерировать код, обеспечивающий эффективный доступ к памяти, а для остальных – гораздо более длинный и медленный код. Далее предположим, что P_0 и P_1 передают P_3 значение 1, а P_2 – значение 17.



После клонирования P_3

Распространение констант по графу вызовов здесь не поможет, потому что оно должно вычислить параметр P_3 как $1 \wedge 1 \wedge 17 = \perp$, что не допускает специализации в P_3. Клонирование процедуры может создать место, где параметр всегда равен 1, как в P_{3a} в графе во врезке. Вызовам, которые подавляют оптимизацию, – (P_2, P_3) в оригинальном графе вызовов – сопоставляется P_{3b}. Компилятор может сгенерировать оптимизированный код для P_{3a} и общий для P_{3b}.

10.6.3. Расчленение цикла

В случае расчленения цикла (loop unswitching) инвариантные относительно цикла операции управления потоком выносятся из цикла. Если предикат в конструкции `if-then-else` инвариантен, то компилятор может переписать цикл, вынеся из него `if-then-else` и сгенерировав специальные копии цикла в каждой половине нового `if-then-else`. Простой пример показан на рис. 10.7.

Расчленение – это обеспечивающее преобразование; оно позволяет компилятору изменять тела циклов способами, реализовать которые иначе было бы затруднительно. После расчленения в получившихся циклах меньше операций управления потоком. Они выполняют меньше ветвлений и операций, необходимых для их поддержки. Это может улучшить планирование команд и распределение регистров, а также ускорить выполнение. Если бы исходный цикл содержал инвариантный код внутри `if-then-else`, то ОПК не смог бы

вынести его из цикла. Но после расчленения ОПК легко найдет и устранит такие избыточности.

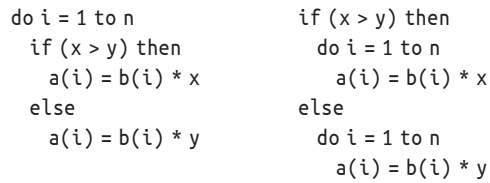


Рис. 10.7 Расчленение короткого цикла

У расчленения также имеется простой и прямой эффект, способный улучшить программу: оно выносит за пределы цикла ветвление в инвариантном условном предложении. Вынести поток управления из цикла трудно. Методы, основанные на анализе потока данных, как, например, ОПК, испытывают сложности с такими конструкциями, потому что преобразование изменяет ГПУ, на которое опирается анализ. Методы типа СНЗ и НЗД могут распознать случаи, когда предикаты, управляющие конструкциями `if-then-else`, постоянны или избыточны, но не способны ни упростить поток управления, ни вынести его из цикла.

10.6.4. Переименование

Большинство скалярных преобразований переписывают или переупорядочивают операции в коде. На примере нескольких контекстов мы видели, что выбор имен может скрыть или раскрыть возможности для улучшения. Например, в ЛНЗ преобразование имен в блоке в SSA-имена раскрывало несколько возможностей, которые иначе было бы трудно заметить.

Для многих преобразований тщательное построение «правильного» пространства имен может раскрыть дополнительные возможности – либо сделав больше фактов видимыми для анализа, либо избежав некоторых побочных эффектов из-за повторного использования памяти. Как было сказано в разделе 10.5.1, автор компилятора может повысить эффективность ЛНЗ путем кодирования тождественности значений в самом пространстве имен, тогда основанный на именах подход ЛНЗ к анализу потока данных сможет обнаружить избыточность и выполнить ее устранение и перемещение кода. Тщательное переименование раскрывает больше возможностей для ЛНЗ и делает его более эффективным.

Точно так же имена важны для планирования команд. В планировщике имена кодируют зависимости по данным, которые ограничивают размещение операций в спланированном коде. Когда повторное использование имени отражает фактический поток значений, оно несет критическую информацию, необходимую для обеспечения корректности. Если же повторное использование имени возникло, потому что предшествующий проход уплот-

нил пространство имен, то оно необязательно накладывает какие-то ограничения на планирование. Например, распределитель регистров помещает различные значения в один и тот же физический регистр, чтобы улучшить использование регистров. Если компилятор выполняет распределение раньше планирования, то распределитель может привнести кажущиеся ограничения на планирование, которых исходный код, строго говоря, не требует.

Кажущееся ограничение, объясняемое именованием, часто называют *ложным разделением*.

Переименование – тонкая материя. Отдельные преобразования могут выиграть от наличия пространств имен с разными свойствами. Авторы компиляторов уже давно поняли, что перемещение и переписывание операций могут улучшить программу. Точно так же они должны осознать, что переименование способно повысить эффективность оптимизатора. Как показала технология SSA, компилятору совсем необязательно ограничиваться пространством имен, определенным программистом или фазой анализа. Переименование – плодородная почва для будущей работы.

ИТОГИ РАЗДЕЛА

Как мы видели в главе 7, форма ПП влияет на код, который может сгенерировать компилятор. Методы, описанные в этом разделе, позволяют переписать код, создав новые возможности для оптимизаций. В них используются репликация, избирательное переписывание и переименование, чтобы создать в коде места, которые можно улучшить, применив специальные преобразования.

Клонирование, будь то на уровне блока или процедуры, достигает эффекта путем устранения точек схождения потоков управления. Оно способно упрощать пути и объединять блоки или процедуры. Расчленение цикла выносит управляющие конструкции из цикла, делая их более пригодными для оптимизации. Переименование – мощная идея, имеющая широкое применение, включая устранение избыточности, снижение стоимости операций, планирование выполнения и распределение регистров.

ВОПРОСЫ

1. Клонирование суперблока создает новые возможности для других оптимизаций. Рассмотрите балансировку дерева по высоте. Как здесь может помочь клонирование суперблока? Каким вы видите преобразование, следующее за клонированием суперблока, которое раскрыло бы дополнительные возможности для балансировки дерева? Сопоставьте результаты применения СНЗ вслед за клонированием с результатами применения ЛНЗ к тому же коду.
2. Клонирование процедуры нацелено на устранение тех же неэффективностей, что и встраивание. Есть ли место для обоих преобразований в одном и том же компиляторе? Каковы потенциальные преимущества и риски каждого преобразования? Как компилятор мог бы сделать выбор?

SSA-ГРАФ

В некоторых алгоритмах рассмотрение SSA-формы кода как графа упрощает либо обсуждение, либо реализацию. Алгоритмы разреженного условного распространения констант (SSCP) и снижения стоимости операций работают с SSA-формой как с графом.

В SSA-форме каждому имени соответствует ровно одно определение. Использование этого имени в операции *i* можно интерпретировать как цепочку, ведущую из *i* обратно к определению имени. В процессе построения SSA-формы компилятор легко может построить отображения использований на их определения, а определений – на их использования. Эти отображения можно интерпретировать как граф, в котором ребра соединяют определения с использованиями и отражают поток значений в оригинальном коде.

При изображении SSA-графов ребра направлены от использования к соответствующему определению, чтобы представить отношение, индуцированное SSA-именами. Компилятору может быть интересен проход по ребрам в обоих направлениях. SCCP распространяет значения от определений к использованиям. OSR преимущественно движется от использований к определениям. Автор компилятора легко может построить граф, допускающий обход в любом направлении.

10.7. Дополнительные темы

Большинство примеров в этой главе призваны проиллюстрировать какой-то конкретный эффект, который компилятор может использовать для ускорения работы программы. Иногда совместное выполнение двух оптимизаций может дать результаты, недостижимые путем применения их по отдельности в любой комбинации. В следующем подразделе приведен один такой пример: комбинирование распространения констант с устранением недостижимого кода. В разделе 10.7.2 приведен второй, более сложный пример специализации: снижение стоимости операторов путем *замены теста линейной функции* (linear-function test replacement – LFTR). Алгоритм, который мы представим, OSR, проще предыдущих алгоритмов, потому что опирается на свойства SSA-формы. Наконец, в разделе 10.7.3 обсуждаются некоторые вопросы, возникающие при выборе конкретного порядка применения преобразований оптимизатором.

10.7.1. Комбинирование оптимизаций

Иногда включение двух разных оптимизаций в объединенный каркас и совместное их применение может давать результаты, которые невозможно получить путем применения их по отдельности в любой комбинации. Например, рассмотрим алгоритм разреженного простого распространения констант (SSCP) из раздела 9.3.6. Он назначает решеточное значение результату каждой операции в SSA-форме программы. После остановки все определения помечены решеточными значениями, равными $\top$, $\bot$ или известной константе.

Конечный тег $\top$ показывает, что значение зависит от неинициализированной переменной или встречается в недостижимом блоке.

SSCP сопоставляет решеточное значение операнду, используемому в условном переходе. Если это значение равно $\bot$, то достижима любая цель пере-

хода. Если значение не равно ни ⊥, ни ⊤, то операнд должен иметь известное значение, и компилятор может заменить условный переход безусловным на одну из двух его целей, что упростит ГПУ. Поскольку при этом из ГПУ удаляется одно ребро, блок, который был целью условного перехода, может оказаться недостижимым. Распространение констант может игнорировать любые эффекты недостижимого блока. В SSCP нет механизма, позволяющего воспользоваться этим знанием.

Мы можем обобщить алгоритм SSCP на основе этих наблюдений. Получающийся алгоритм *разреженного условного распространения констант* (sparse conditional constant propagation – SCCP) показан на рис. 10.8–10.10.

```
CFGWorkList ← { ребра, исходящие из n0 }
SSAWorkList ← ∅
for каждого ребра ГПУ e do
  пометить e как невыполненное
for каждого определения и использования x в процедуре do
  Value(x) ← ⊤
while (CFGWorkList ≠ ∅ or SSAWorkList ≠ ∅) do
  if CFGWorkList ≠ ∅ then
    удалить ребро e = (m,n) из CFGWorkList
    if e помечено как невыполненное then
      пометить e как выполненное
      EvaluateAllPhisInBlock((m,n))
      if никакое другое ребро, входящее в n, не помечено как выполненное then
        if n – присваивание then
          EvaluateAssign(n)
          положить o равным преемнику n в ГПУ
          добавить (n,o) в CFGWorkList
        else EvaluateConditional(n)

  if SSAWorkList ≠ ∅ then
    удалить ребро e = (s,d) из SSAWorkList
    c ← вершина ГПУ, в которой d используется
    if какое-либо ребро, входящее в c, помечено как выполненное then
      if d – аргумент φ-функции then
        EvaluatePhi((s,d))
      else если c - присваивание then
        EvaluateAssign(c)
      else EvaluateConditional(c)
```

Рис. 10.8 Разреженное условное распространение констант

Концептуально SCCP прост. Он инициализирует структуры данных, обходит два графа, ГПУ и SSA, распространяет информацию о доступности по ГПУ и информацию о значениях по SSA-графу. Алгоритм останавливается, когда информация о значениях достигает неподвижной точки; поскольку решетка распространения значения мелкая, останавливается он быстро. Объединяя эти два вида информации, SCCP может обнаружить недостижимый код и по-

стоянные значения, которые компилятор не мог бы обнаружить применением любой комбинации SSCP и устранения недостижимого кода.

```
EvaluateAssign(m)        /* m – вершина ГПУ */
  for каждого значения y, используемого выражением в m do
    положить (x, y) равным ребру SSA-графа, которое поставляет y
    Value(y) ← Value(x)
  положить d равным имени значения, порождаемого m
  if Value(d) ≠ ⊥ then
    v ← результат вычисления m на решеточных значениях
    if v ≠ Value(d) then
      Value(d) ← v
      for каждого ребра SSA-графа (d, u) do
        добавить (d, u) в SSAWorklist

EvaluateConditional(m)   /* m – вершина ГПУ */
  положить (s,d) равным ребру SSA-графа, на которое есть ссылка в m
  if Value(d) ≠ ⊥ then
    if Value(d) ≠ Value(s) then
      Value(d) ← Value(s)
      if Value(d) = ⊥ then
        for каждого ребра ГПУ (m, n) do
          добавить (m, n) в CFGWorkList
      else
        положить (m, n) равным ребру SSA-графа, соответствующему Value(d)
        добавить (m, n) в CFGWorkList
```

Рис. 10.9 Вычисление присваиваний и условных предложений

Чтобы упростить объяснение SCCP, будем предполагать, что в каждой вершине ГПУ n находится одно предложение плюс факультативные φ-функции. Если у n один предшественник, то в нем находится присваивание или условный переход. Если у n несколько предшественников, то он также может иметь φ-функции перед присваиванием или условным переходом.

Но в деталях SCCP сложнее, чем SSCP или устранение недостижимого кода. Использование двух графов делает внутреннюю кухню более запутанной. Зависимость потока значений от достижимости прибавляет алгоритму работы. В итоге мы получаем более мощный, но и более сложный алгоритм.

Вначале алгоритм инициализирует каждое поле *Value* значением $\top$ и помечает каждое ребро ГПУ как «невыполненное». Рабочий список ребер ГПУ инициализируется множеством ребер, которые исходят из входной вершины процедуры, n_0. Рабочий список ребер SSA-графа инициализируется пустым множеством.

После инициализации алгоритм в цикле выбирает ребро из одного из рабочих списков и обрабатывает его. Для ребра ГПУ (m, n) SCCP определяет, помечено ли (m, n) как выполненное. Если да, то SCCP больше

В этом обсуждении блок называется *достижимым*, если хотя бы одно входящее в него ребро ГПУ помечено как выполненное.

ничего для (m, n) не делает. Если (m, n) помечено как невыполненное, то SCCP помечает его как выполненное и вычисляет все φ-функции в начале блока n. Затем SCCP проверяет, был ли уже произведен вход в n по другому ребру. Если нет, то SCCP вычисляет присваивание или условное предложение в n. В процессе этой обработки в любой рабочий список могут быть добавлены дополнительные ребра.

```
EvaluatePhi((s, d))              /* (s, d) – ребро SSA-графа */
  положить p равной φ-функции, которую использует d
  EvaluateOperands(p)
  EvaluateResult(p)

EvaluateAllPhisInBlock((m, n))   /* (m,n) – ребро графа ГПУ */
  for каждой φ-функции p в блоке n do
    EvaluateOperands(p)
  for каждой φ-функции p в блоке n do
    EvaluateResult(p)

EvaluateOperands(phi)
  положить w равным имени, определенному φ-функцией phi
  if Value(w) ≠ ⊥ then
    for каждого параметра p φ-функции phi do
      положить c равным ребру ГПУ, соответствующему p
      положить (x,y) равным ребру SSA-графа, использованием которого является p
      if c помечено как выполненное then
        Value(x) ← Value(y)

EvaluateResult(phi)
  положить x равным имени, определенному φ-функцией phi
  if Value(x) ≠ ⊥ then
    v ← результат вычисления phi на решеточных значениях
    if Value(x) ≠ v then
      Value(x) ← v
      for каждого ребра SSA-графа (x, y) do
        добавить (x, y) в SSAWorkList
```

Рис. 10.10 Вычисление φ-функций

Для ребра SSA-графа алгоритм сначала проверяет, достижим ли конечный блок. Если да, то SCCP вызывает одну из функций *EvaluatePhi*, *EvaluateAssign* или *EvaluateConditional*, в зависимости от вида операции, в которой используется SSA-имя. Когда SCCP должен вычислить присваивание или условное предложение на решеточных значениях, он следует той же схеме, что SSCP (см. обсуждение в разделе 9.3.6). Всякий раз, как решеточное значение для определения изменяется, все использования этого имени добавляются в рабочий список SSA.

Поскольку SCCP распространяет значения только в блоки, для которых уже была доказана исполнимость, он избегает обработки недостижимых блоков. Поскольку каждый шаг распространения значений защищен проверкой флага исполнения для входящего ребра, значения из недостижимых блоков не утека-

ют за пределы этих блоков. Таким образом, значения из недостижимых блоков не играют никакой роли для задания решеточных значений в других блоках.

После шага распространения необходим еще один, последний, проход, чтобы заменить операции, среди операндов которых есть снабженные тегами *Value*, а не $\perp$. Этот последний проход может специализировать многие из таких операций. Он также должен переписать условные переходы с известным исходом операциями безусловного перехода. Последующий проход может удалить недостижимый код (см. раздел 10.2.3). Алгоритм не может переписать код, пока распространение не завершилось.

Тонкости операций вычисления и переписывания

При моделировании отдельных операций возникают некоторые тонкости. Например, встретив операцию умножения с операндами $\top$ и $\perp$, алгоритм мог бы заключить, что операция порождает $\perp$. Однако это было бы преждевременно. Последующий анализ мог бы понизить $\top$ до константы 0, так что умножение дает значение 0. Используя правило $\top \times \perp \rightarrow \perp$, SCCP создает возможность немонотонного поведения – значение операции умножения могло бы последовательно изменяться как $\top$, $\perp$, 0, что увеличило бы время работы SCCP. Не менее важно и то, что он мог бы некорректно привести другие значения к $\perp$, из-за чего SCCP пропустил бы возможности для улучшения.

Чтобы решить эту проблему, SCCP должен использовать три правила для умножений, в которые входит $\perp$: $\top \times \perp \rightarrow \top$, $\alpha \times \perp \rightarrow \perp$ для $\alpha \neq \top$ и $\alpha \neq 0$ и $0 \times \perp \rightarrow 0$. Тот же эффект имеет место для любой операции, для которой значение одного аргумента может полностью определить результат. Другими примерами могут служить сдвиг на величину, большую длины слова, логическое И с нулем и логическое ИЛИ со значением, состоящим из одних единиц.

У некоторых переписываний имеются непредвиденные последствия. Например, замена $4 \times s$ для неотрицательных s сдвигом замещает коммутативную операцию некоммутативной. Если впоследствии компилятор попытается переупорядочить выражения, пользуясь коммутативностью, это преждевременное переписывание закроет такую возможность.

Этот вид взаимодействия может оказывать значительное влияние на качество кода. Чтобы решить, когда компилятору следует преобразовывать $4 \times s$ в сдвиг, автор компилятора должен учитывать порядок применения оптимизаций.

Эффективность

SCCP может находить константы, которых SSCP не видит. Аналогично он может обнаружить недостижимый код, ускользающий от любой комбинации алгоритмов из раздела 10.2. Его мощь проистекает из комбинирования анализа достижимости с распространением решеточных значений. Он может устранить некоторые ребра ГПУ, потому что решеточных значений достаточно, чтобы определить, по какому пути пойдет ветвление. Он может игнорировать ребра SSA-графа, появившиеся в результате недостижимых

операций (путем инициализации этих определений значением ⊤), потому что эти операции будут вычислены, если блок окажется помечен как достижимый. Мощь SCCP является следствием взаимодействия между этими видами анализа – распространения констант и достижимости.

Если достижимость не повлияла на конечные решеточные значения, то тех же эффектов можно было бы достичь, выполнив распространение констант (и заменив условные переходы с постоянными значениями безусловными) с последующим устранением недостижимого кода. Если распространение констант не сыграло никакой роли в достижимости, то тех же эффектов можно было бы достичь при другом порядке – сначала устранение недостижимого кода, затем распространение констант. Способность SCCP находить упрощения, помимо этих двух комбинаций, объясняется именно тем, что эти оптимизации взаимозависимы.

10.7.2. Снижение стоимости

Снижение стоимости операций – это преобразование, которое заменяет последовательность дорогих («сильных») операций последовательностью дешевых («слабых») операций, вычисляющих те же значения. Классический пример – замена умножений на переменную цикла эквивалентными сложениями. Этот конкретный случай естественно возникает при раскрытии ссылок на элементы массивов и структур в цикле. На рис. 10.11(a) показан ILOC-код, который мог бы быть сгенерирован для следующего цикла:

```
sum ← 0
for i ← 1 to 100
   sum ← sum + a(i)
```

```
        loadI  0       ⇒ r_s0
        loadI  1       ⇒ r_i0
        loadI  100     ⇒ r_100
l1:     phi    r_i0,r_i2 ⇒ r_i1
        phi    r_s0,r_s2 ⇒ r_s1
        subI   r_i1,1   ⇒ r_1
        multI  r_1,4    ⇒ r_2
        addI   r_2,@a   ⇒ r_3
        load   r_3      ⇒ r_4
        add    r_s1,r_4 ⇒ r_s2
        addI   r_i1,1   ⇒ r_i2
        cmp_LE r_i2,r_100 ⇒ r5
        cbr    r_5      → l1,l2
l2:     ...
```

(a) Исходный код

```
        loadI 0        ⇒ r_s0
        loadI @a       ⇒ r_t6
        addI r_t6, 396 ⇒ r_lim
l1:     phi r_t6,r_t8  ⇒ r_t7
        phi r_s0,r_s2  ⇒ r_s1
        load r_t7      ⇒ r_4
        add r_s1,r_4   ⇒ r_s2
        addI r_t7,4    ⇒ r_t8
        cmp_LE r_t8,r_lim ⇒ r_5
        cbr 5 → 1,2
l2:     ...
```

(b) Код после снижения стоимости

Рис. 10.11 Пример снижения стоимости

Код представлен в полуусеченной SSA-форме; чисто локальные значения (r_1, r_2, r_3, r_4 и r_5) не имеют ни индексов, ни φ-функций. Обратите внимание, как ссылка на `a(i)` раскрывается в четыре операции: `subI`, `multI` и `addI`, которые вычисляют выражение `(i-1)` × 4 + `@a`, и `load`, которая определяет r_4.

На каждой итерации эта последовательность операций с чистого листа вычисляет адрес a(i) как функцию от переменной цикла i. Рассмотрим последовательность значений, которые принимают r_{i1}, r_1, r_2 и r_3.

```
r_i1: { 1, 2, 3, ..., 100 }
r_1:  { 0, 1, 2, ..., 99 }
r_2:  { 0, 4, 8, ..., 396 }
r_3:  { @a, @a + 4, @a + 8, ..., @a + 396 }
```

Единственное назначение r_1, r_2 и r_3 – вычислить адрес для операции load. Если бы код мог вычислить каждое значение r_3 по предыдущему, то появилась бы возможность устранить операции, определяющие r_1 и r_2. Конечно, r_3 нужно было бы инициализировать и обновлять, а также понадобилась бы φ-функция в l_1 и, возможно, в l_2.

На рисунке (b) показан код после снижения стоимости, LFTR и устранения мертвого кода. Он вычисляет значения, которые раньше были в r_3, непосредственно в r_{t_7} и использует r_{t_7} в операции load. Проверка конца цикла, где в исходном коде использовался r_{i_2}, теперь модифицирована и использует r_{t_8}. В результате вычисления r_1, r_2, r_3, r_{i_0}, r_{i_1} и r_{i_2} оказываются мертвыми. После их удаления мы получаем окончательный код. Теперь цикл содержит всего пять операций, не считая φ-функций, тогда как в исходном коде их было восемь. (При трансляции кода в SSA-форме φ-функции переписываются в виде операций копирования, многие из которых можно объединить.)

Если операция multI дороже addI, то экономия будет больше. Исторически именно высокая стоимость умножения стала обоснованием алгоритма снижения стоимости. Однако даже если стоимости умножения и сложения одинаковы, цикл со сниженной стоимостью может быть предпочтительнее, потому что его форма лучше подходит для последующих этапов компилятора. В частности, если в целевой машине имеется автоинкрементный режим адресации, то операцию addI в цикле часто можно сделать частью load или store. Для умножения такой возможности нет.

Далее в этом разделе представлен простой алгоритм снижения стоимости операций (operator strength reduction – OSR), а затем схема LFTR, которая отодвигает проверки конца цикла дальше от переменных, которые в противном случае были бы мертвыми. OSR работает с SSA-формой кода, которая рассматривается как граф. На рис. 10.12 показан код нашего примера в SSA-форме, а рядом с ним SSA-граф.

Предварительные сведения

Алгоритм снижения стоимости ищет контексты, в которых операция, например умножение, выполняется внутри цикла, а ее операнды являются либо (1) значениями, не изменяющимися в цикле (*региональными константами*), либо (2) значениями, систематически изменяющимися от итерации

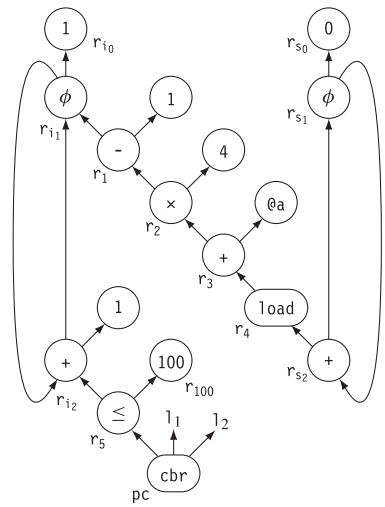
Региональная константа
Значение, которое не изменяется внутри данного цикла, называется *региональной константой* для этого цикла.

Индуктивная переменная
Значение, которое увеличивается или уменьшается на постоянную величину на каждой итерации цикла, называется *индуктивной переменной*.

к итерации (*индуктивными переменными*). Обнаружив такую ситуацию, алгоритм создает новую индуктивную переменную, чтобы вычислить ту же самую последовательность значений более эффективно. Ограничения на форму операндов операции умножения гарантируют, что эту новую индуктивную переменную можно вычислить с помощью сложений, а не умножений.

```
     loadI  0          ⇒ rs0
     loadI  1          ⇒ ri0
     loadI  100        ⇒ r100
l1:  phi    ri0, ri2   ⇒ ri1
     phi    rs0, rs2   ⇒ rs1
     subI   ri1, 1     ⇒ r1
     multI  r1, 4      ⇒ r2
     addI   r2, @a     ⇒ r3
     load   r3         ⇒ r4
     add    rs1, r4    ⇒ rs2
     addI   ri1, 1     ⇒ ri2
     cmp_LE ri2, r100  ⇒ r5
     cbr    r5         → l1, l2
l2:  ...
```

(a) Пример в форме SSA



(b) Соответствующий SSA-граф

Рис. 10.12 Соотношение между SSA в ILOC и SSA-графом

```
x ← c × i
x ← i × c
x ← c + i
x ← i + c
x ← i - c
```
Операции-кандидаты

Операция, которую алгоритм может упростить описанным способом, называется *операцией-кандидатом*. Чтобы упростить изложение OSR, мы будем рассматривать только операции-кандидаты, имеющие один из форматов, показанных во врезке, где c – региональная константа, а i – индуктивная переменная. Ключом к нахождению и упрощению операций-кандидатов является эффективная идентификация региональных констант и индуктивных переменных. Ограничение пятью представленными формами важно.

Региональная константа может быть либо литеральной константой, например 10, либо инвариантным относительно цикла значением, т. е. таким, которое не изменяется в цикле. Если код представлен в форме SSA, то компилятор может определить, является ли аргумент инвариантным относительно цикла, проверив, где находится его единственное определение, – блок, содержащий определение, должен доминировать над входом в цикл, где определена индуктивная переменная. OSR может проверить оба этих условия за постоянное время. Распространение констант и отложенное пере-

мещение кода перед снижением стоимости могут выявить дополнительные региональные константы.

Интуитивно понятно, что индуктивная переменная характеризуется тем, что значения, принимаемые ей в цикле, образуют арифметическую прогрессию. Но для целей этого алгоритма мы можем использовать более конкретное и узкое определение: индуктивная переменная – это компонента сильной связности (КСС) SSA-графа, в которой любая операция, обновляющая ее значение, – одна из: (1) индуктивная переменная плюс региональная константа, (2) индуктивная переменная минус региональная константа, (3) φ-функция или (4) копирование типа регистр-регистр другой индуктивной переменной. Хотя это определение гораздо менее общее, чем традиционное, алгоритму OSR его достаточно для нахождения и упрощения операций-кандидатов. Чтобы найти индуктивные переменные, OSR ищет КСС в SSA-графе и обходит каждую КСС с целью установить, имеет ли каждая операция в ней одну из четырех вышеперечисленных форм.

Поскольку OSR ищет индуктивные переменные в SSA-графе, а региональные константы – относительно циклов в ГПУ, проверка того, что значение постоянно относительно цикла, содержащего конкретную индуктивную переменную, затруднена. Рассмотрим операцию *o* вида `x ← i × c`, где `i` – индуктивная переменная. Чтобы *o* была кандидатом на снижение стоимости, `c` должна быть региональной константой относительно самого внешнего цикла, в котором `i` изменяется. Чтобы проверить, обладает ли `c` этим свойством, OSR должен соотнести КСС для `i` в SSA-графе с циклом в ГПУ. Для упрощения этой проверки OSR добавляет в каждую вершину SSA-графа поле, содержащее ее «заголовок».

OSR находит вершину SSA-графа с наименьшим номером в порядке реверсивного обратного обхода в той КСС, где определена `i`. Он рассматривает эту вершину как заголовок КСС и записывает этот факт в поле заголовка каждой вершины КСС. (Для любой вершины SSA-графа, не являющейся частью индуктивной переменной, поле заголовка устанавливается в *null*.) В SSA-форме заголовок индуктивной переменной – это φ-функция в начале самого внешнего цикла, в котором эта переменная изменяется. В операции `x ← i × c`, где `i` – индуктивная переменная, `c` является региональной константой, если блок ГПУ, содержащей ее определение, доминирует над блоком ГПУ, содержащим заголовок `i`. Это условие гарантирует, что `c` инвариантна в самом внешнем цикле, где `i` изменяется. Для проверки этого условия алгоритм построения формы SSA должен строить отображение каждой вершины SSA-графа на блок ГПУ, от которого она произошла.

Поле заголовка играет важную роль в определении того, можно ли упростить операции. Когда OSR встречает операцию вида `x ← y × z`, он может определить, является ли `y` индуктивной переменной, проследовав по ребру SSA-графа к определению `y` и заглянув в его поле заголовка. Значение *null* в этом поле означает, что `y` не является индуктивной переменной. Если и `y`, и `z` содержат *null* в поле заголовка, то снизить стоимость операции невозможно.

Если поле заголовка в y или z ненулевое, то OSR может воспользоваться заголовком, чтобы узнать, является ли второй операнд региональной константой. Предположим, что заголовок y отличен от *null*. По заголовку y OSR ищет в отображении SSA-графа в ГПУ вход в самый внешний цикл, в котором y изменяется. Если блок ГПУ, содержащий определение z, доминирует над блоком ГПУ заголовка y, то z является региональной константой относительно индуктивной переменной y.

Алгоритм

Чтобы снизить стоимость, OSR должен исследовать каждую операцию и определить, является ли один из ее операндов индуктивной переменной, а другой – региональной константой. Если операция отвечает этим критериям, то OSR может упростить ее, создав новую индуктивную переменную, которая вычисляет необходимые значения, и заменив исходную операцию копированием типа регистр-регистр из этой новой переменной. (Следует избегать создания дубликатов индуктивных переменных.)

Из предыдущего обсуждения мы знаем, что OSR может идентифицировать индуктивные переменные с помощью поиска КСС в SSA-графе. Региональные константы он может обнаружить, исследовав определение значения. Если значение определено в результате непосредственной операции или его блок в ГПУ доминирует над блоком заголовка индуктивной переменной, то оно является региональной константой. Теперь главное – соединить все эти идеи в эффективный алгоритм.

OSR основана на алгоритме Тарьяна поиска компонент сильной связности. Как показано на рис. 10.13, функция *OSR* принимает SSA-граф в качестве аргумента и в цикле применяет к нему функцию поиска компоненты сильной связности, *DFS*. (Этот процесс останавливается, когда *DFS* посетит все вершины *G*.)

DFS выполняет поиск в глубину в SSA-графе. Она сопоставляет каждой вершине номер, соответствующий порядку посещения этой вершины. Она помещает каждую вершину в стек и помечает ее номером самой нижней вершины, которую можно достичь из одного из ее непосредственных потомков поиском в глубину. Если после возврата *DFS* из обработки потомков самая нижняя вершина, достижимая из *n*, имеет номер, сопоставленный *n*, то *n* является заголовком КСС. *DFS* выталкивает вершины из стека, пока не достигнет *n*; все эти вершины принадлежат КСС.

```
x ← c × i
x ← i × c
x ← c + i
x ← i + c
x ← i - c
```

Операции-кандидаты

DFS удаляет КСС из стека в порядке, который упрощает оставшуюся часть *OSR*. Когда КСС выталкивается из стека и передается *Process*, *DFS* уже посетила всех ее потомков в SSA-графе. Если мы интерпретируем SSA-граф, так что его ребра направлены от использования к определению, как на рис. 10.12, то операции-кандидаты посещаются только после того, как их операнды были переданы *Process*. Когда *Process* встречает операцию, являющуюся кандидатом на снижение стоимости, ее операнды уже классифицированы. Таким образом, *Process*

может исследовать операции, идентифицировать кандидатов и вызывать *Replace*, чтобы переписать их в упрощенной форме со сниженной стоимостью, прямо в процессе поиска в глубину.

```
OSR(G)
  nextNum ← 0
  while существуют непосещенные n ∈ G do
    DFS(n)

DFS(n)
  n.Num ← nextNum++
  n.Visited ← true
  n.Low ← n.Num
  push(n)
  for каждого операнда o вершины n do
    if o.Visited = false then
      DFS(o)
      n.Low ← min(n.Low, o.Low)
    if o.Num < n.Num and
      o находится в стеке then
      n.Low ← min(n.Low, o.Num)
  if n.Low = n.Num then
    SCC ← ∅
    repeat until x = n do
      x ← pop( )
      SCC ← SCC ∪ { x }
    Process(SCC)

Process(N)
  if N имеет только один член n then
    if n является "кандидатом" then
      Replace(n, iv, rc)
    else n.Header ← null
  else ClassifyIV(N)

ClassifyIV(N)
  IsIV ← true
  for каждой вершины n ∈ N do
    if n – не является допустимым
      обновлением индуктивной
      переменной then
      IsIV ← false
  if IsIV then
    header ← n ∈ N с наименьшим номером
      в порядке обхода РОП
    for каждой вершины n ∈ N do
      n.Header ← header
  else
    for каждой вершины n ∈ N do
      if n является "кандидатом" then
        Replace(n, iv, rc)
      else n.Header ← null
```

Рис. 10.13 Алгоритм снижения стоимости операторов

Когда *Process* идентифицирует *n* как операцию-кандидата, она находит как ее индуктивную переменную, *iv*, так и региональную константу, *rc*.

DFS передает каждую КСС *Process*. Если КСС состоит из единственной вершины *n*, имеющей форму операции-кандидата, показанную во врезке, то *Process* передает *n* функции *Replace* вместе с ее индуктивной переменной *iv* и региональной константой *rc*. *Replace* переписывает код, как описано в следующем разделе. Если КСС содержит несколько вершин, то *Process* передает КСС функции *ClassifyIV*, чтобы та определила, является она индуктивной переменной или нет.

ClassifyIV исследует каждую вершину, принадлежащую КСС, сопоставляя ее с множеством допустимых обновлений для индуктивной переменной. Если все обновления допустимы, то КСС является индуктивной переменной, и *Process* устанавливает поле заголовка каждой вершины равным вершине в КСС с наименьшим номером в порядке обхода РОП. Если КСС не является индуктивной переменной, то *ClassifyIV* повторно посещает каждую вершину в КСС, проверяя ее на предмет операции-кандидата, и либо передает ее *Replace*, либо устанавливает заголовок, чтобы показать, что это не индуктивная переменная.

Переписывание кода

В оставшейся части *OSR* реализован шаг переписывания. И *Process*, и *ClassifyIV* вызывают *Replace*, чтобы выполнить переписывание. На рис. 10.14 показан код *Replace* и вспомогательных функций *Reduce* и *Apply*.

```
Replace(n, iv, rc)
  result ← Reduce(n.op, iv, rc)
  заменить n копией из result
  n.header ← iv.header

Reduce(op, iv, rc)
  result ← Lookup(op, iv, rc)
  if result не равно "not found" then
    result ← NewName( )
    Insert(op, iv, rc, result)
    newDef ← Clone(iv, result)
    newDef.header ← iv.header
    for каждого операнда o вершины newDef
    do
      if o.header = iv.header then
        заменить o на Reduce(op, o, rc)
      else if op совпадает с × или
          newDef.op - φ-функция then
        заменить o на Apply(op, o, rc)
  return result

Apply(op, o1, o2)
  result ← Lookup(op, o1, o2)
  if result is "not found" then
    if o1 – индуктивная переменная и
      o2 – региональная константа then
      result ← Reduce(op, o1, o2)
    else if o2 – индуктивная переменная и
      o1 – региональная константа then
      result ← Reduce(op, o2, o1)
    else
      result ← NewName( )
      Insert(op, o1, o2, result)
      найти блок b, над которым
        доминируют определения o1 и o2
      создать "op o1, o2 ⇒ result"
        в конце b и положить ее заголовок
        равным null
  return result
```

Рис. 10.14 Алгоритм для переписывания кода

Replace принимает три аргумента: вершину SSA-графа *n*, индуктивную переменную *iv* и региональную константу *rc*. Последние два являются операндами *n*. *Replace* вызывает *Reduce*, чтобы переписать операцию, представленную *n*. Затем она заменяет *n* операцией копирования из результата, возвращенного *Replace*. Она устанавливает поле заголовка *n* и возвращается.

Основную часть работы делают *Reduce* и *Apply*. Они используют хеш-таблицу, чтобы не вставлять дубликаты операций. Поскольку *OSR* работает с SSA-именами, достаточно одной глобальной хеш-таблицы. Ее можно инициализировать в *OSR* перед первым обращением к *DFS*. *Insert* добавляет записи в хеш-таблицу, а *Lookup* опрашивает ее.

План, реализуемый *Reduce*, прост. Она принимает код операции и два ее операнда и либо создает новую индуктивную переменную для замены вычисления, либо возвращает имя индуктивной переменной, ранее созданной для той же комбинации кода операции и операндов. Чтобы избежать лишней работы, она справляется с хеш-таблицей. Если требуемой индуктивной переменной еще нет в хеш-таблице, то она создает ее, выполняя два шага. Сначала она вызывает *Clone*, чтобы копировать определение *iv* индуктивной переменной в упрощаемой операции. Затем она рекурсивно вызывает себя с операндами этого нового определения.

Эти операнды можно отнести к двум категориям. Если операнд определен внутри КСС, то он является частью *iv*, поэтому *Reduce* рекурсивно вызывает себя с этим операндом. При этом образуется новая индуктивная переменная путем клонирования КСС исходной индуктивной переменной *iv*. Операнд, определенный вне КСС, должен быть либо начальным значением *iv*, либо значением, на которое увеличивается *iv*. Начальное значение должно быть аргументом φ-функции за пределами КСС; *Reduce* вызывает *Apply* для каждого такого аргумента. *Reduce* может оставить инкремент индуктивной переменной в покое, если только операция-кандидат не является умножением. Для умножения *Reduce* должна вычислить новый инкремент как произведение старого на исходную региональную константу *rc*. Для генерирования этого вычисления она вызывает *Apply*.

Apply принимает код операции и два операнда, находит подходящую точку в коде и вставляет эту операцию. Она возвращает новое SSA-имя для результата операции. Некоторые детали нуждаются в пояснениях. Если эта новая операция сама является кандидатом, то *Apply* вызывает для ее обработки *Reduce*. В противном случае *Apply* получает новое имя, вставляет операцию и возвращает результат. (Если *o1* и *o2* – константы, то *Apply* может вычислить результат операции и вставить команду непосредственной загрузки.) *Apply* находит подходящий для новой операции блок, пользуясь информацией о доминировании. Интуитивно понятно, что новая операция должна попасть в блок, над которым доминируют блоки ее операндов. Если один операнд – константа, то *Apply* может продублировать эту константу в блоке, определяющем другой операнд. Иначе оба операнда должны иметь определения, доминирующие над блоком заголовка, и одно должно доминировать над другим. *Apply* может вставить операцию сразу после этого последнего определения.

Возвращаясь к примеру

Рассмотрим, что произойдет, если *OSR* встретится пример на рис. 10.12. Предположим, что она начинает с вершины, помеченной r_{s2}, и посещает левых потомков раньше, чем правых. Она рекурсивно спускается вниз по цепочке операций, которые определяют r_4, r_3, r_2, r_1 и r_{i1}. В r_{i1} она рекурсивно переходит к r_{i2}, а затем к r_{i0}. Она находит две КСС с одной вершиной, которые содержат литеральную константу 1. Ни одна не является кандидатом, поэтому *Process* помечает их как неиндуктивные переменные, сбрасывая заголовки в *null*.

Первая нетривиальная КСС, которую обнаруживает *DFS*, содержит r_{i1} и r_{i2}. Все операции являются допустимыми обновлениями для индуктивной переменной, поэтому *ClassifyIV* помечает каждую вершину как индуктивную переменную и записывает в поле заголовка указатель на r_{i1} – вершину КСС с наибольшим номером в порядке обхода в глубину.

Теперь *DFS* возвращается к вершине для r_1. Ее левый потомок является индуктивной переменной, а правый потомок – региональной константой, поэтому *DFS* вызывает *Reduce*, чтобы создать индуктивную переменную. В этом

случае r_1 равна $r_{i1} - 1$, поэтому начальное значение на единицу меньше начального значения старой индуктивной переменной, т. е. 0. Инкремент тот же самый. На рис. 10.15 под меткой «для r_1» показана КСС, созданная *Reduce* и *Apply*. Наконец, определение r_1 заменяется операцией копирования, $r_1 \leftarrow r_{t1}$. Эта операция копирования помечена как индуктивная переменная.

Далее *DFS* находит КСС, состоящую из вершины для r_2. *Process* классифицирует ее как кандидата, потому что ее левый операнд (копирование, которое теперь определяет r_1) является индуктивной переменной, а правый операнд – региональной константой. *Process* вызывает *Replace*, чтобы создать индуктивную переменную со значением $r_1 \times 4$. *Reduce* и *Apply* клонируют индуктивную переменную для r_1, корректируют инкремент с учетом умножения и добавляют копирование в r_2.

После этого *DFS* передает вершину для r_3 функции *Process*. При этом создается еще одна индуктивная переменная с `@a` в качестве начального значения, и ее значение копируется в r_3.

Process обрабатывает операцию `load`, за которой следует КСС, которая вычисляет сумму. Она находит, что ни одна из этих операций не является кандидатом.

Наконец, *OSR* вызывает *DFS* для еще не посещенной вершины для операции `cbr`. *DFS* посещает сравнение, ранее помеченную индуктивную переменную и константу 100. Никаких упрощений не происходит.

В SSA-графе на рис. 10.15 показаны все индуктивные переменные, созданные этим процессом. Индуктивные переменные, помеченные «для r_1» и «для r_2», мертвы. Индуктивная переменная для `i` была бы мертва, если бы не проверка в конце цикла, где она все еще используется. Чтобы устранить эту индуктивную переменную, компилятор может применить алгоритм LFTR и перенести проверку в индуктивную переменную для r_3.

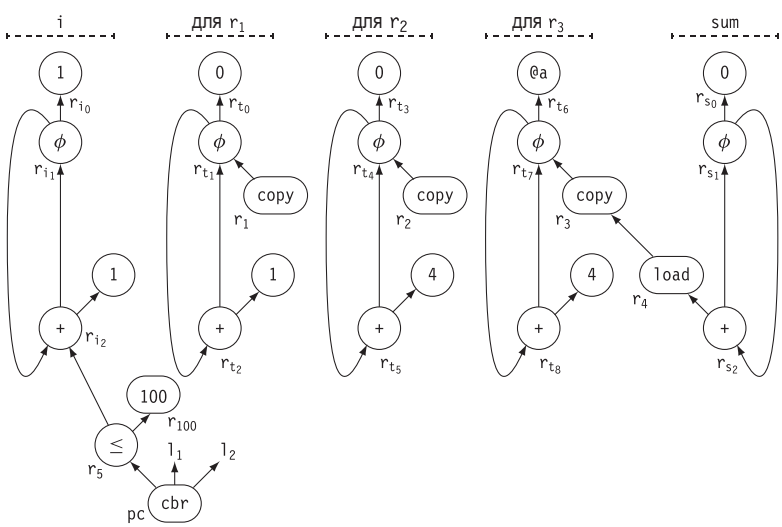


Рис. 10.15 Преобразованный SSA-граф для примера

Замена теста линейной функции

Алгоритм снижения стоимости часто устраняет все использования индуктивной переменной, кроме проверки в конце цикла. В этом случае компилятору иногда удается переписать проверку в конце цикла, так чтобы в ней использовалась другая индуктивная переменная, найденная в цикле. Если компилятор сможет удалить это последнее использование, то сможет устранить и оригинальную индуктивную переменную как мертвый код. Это преобразование называется заменой теста линейной функции (linear-function test replacement – LFTR).

Чтобы выполнить LFTR, компилятор должен (1) найти сравнения, в которых используются индуктивные переменные, больше ни для чего не нужные, (2) найти подходящую новую индуктивную переменную, которую можно было бы использовать в сравнении, (3) вычислить правильную региональную константу для переписанной проверки и (4) переписать код. Организовав кооперацию LFTR с *OSR*, можно упростить все эти задачи и получить быстрое и эффективное преобразование.

Операции, на которые нацелен LFTR, сравнивают значение индуктивной переменной с региональной константой. *OSR* исследует каждую операцию в программе, чтобы определить, является ли она кандидатом на снижение стоимости. Она может легко и недорого построить список всех операций сравнения, которые включают индуктивную переменную. После того как *OSR* закончит работу, LFTR должен заново посетить каждое из этих сравнений. Если стоимость индуктивной переменной, являющейся аргументом сравнения, была снижена *OSR*, то LFTR должен перенаправить сравнение, так чтобы использовалась новая индуктивная переменная.

Чтобы упростить этот процесс, функция *Reduce* может запомнить используемое ей арифметическое соотношение для вывода каждой новой индуктивной переменной. Она может вставить специальное LFTR-ребро из каждой вершины в оригинальной индуктивной переменной, ведущее в соответствующую вершину упрощенной операции, и пометить его кодом операции и региональной константой операции-кандидата, отвечающей за создание этой индуктивной переменной.

На рис. 10.16 эти дополнительные ребра выглядят как пунктирные линии. Последовательность упрощений в примере создает цепочку помеченных ребер. Отправляясь от оригинальной индуктивной переменной, мы находим метки `-1`, `x4` и `+@a`.

Найдя сравнение, подлежащее замене, LFTR может проследовать по ребрам, идущим от аргумента индуктивной переменной к конечной индуктивной переменной, которые образовались из цепочки одного или нескольких упрощений. В сравнении должна использоваться эта индуктивная переменная с подходящей новой региональной константой.

Метки на LFTR-ребрах описывают преобразование, которое следует применить к оригинальной региональной константе, чтобы вывести новую. В нашем примере цепочка ребер ведет из r_{i2} в r_{t8} и порождает значение

(100 - 1) × 4 + @a для преобразованной проверки. На рис. 10.16 показаны ребра и переписанная проверка.

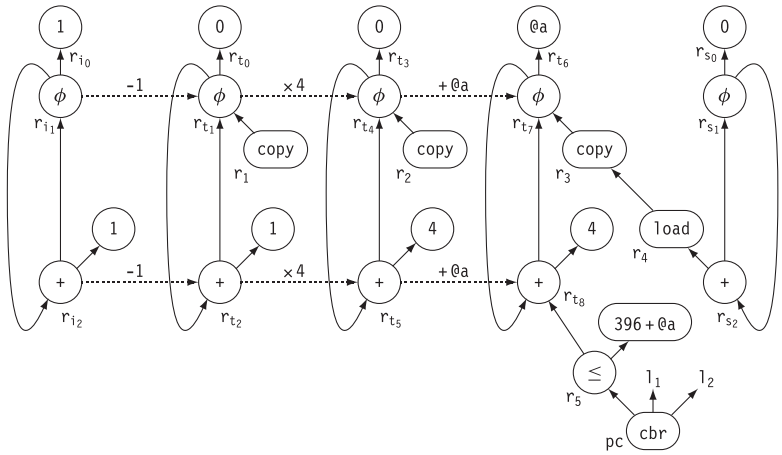


Рис. 10.16 Пример после применения замены теста линейной функции

Эта версия LFTR проста и эффективна. Она опирается на тесную кооперацию с *OSR* с целью нахождения сравнений, которые можно переписать, и запоминания упрощений по мере их применения. Используя эти две структуры данных, LFTR может найти пригодные сравнения, место, куда их следует перенести, и необходимое преобразование постоянного аргумента сравнения.

10.7.3. Выбор последовательности оптимизаций

Последовательность оптимизаций множество оптимизаций и порядок их применения.

Эффективность оптимизатора применительно к заданному коду зависит от последовательности оптимизаций – конкретных преобразований и порядка их применения. Традиционные оптимизирующие компиляторы предлагали пользователю на выбор несколько последовательностей (например, -O, -O1, -O2, ...), реализующих компромисс между временем компиляции и степенью оптимизации. Но больше затраченных на оптимизацию усилий не гарантирует улучшения.

Проблема последовательности оптимизаций возникает, потому что эффективность любого преобразования зависит от нескольких факторов.

1. Присутствует ли в коде возможность, на которую нацелено преобразование? Если нет, преобразование ничего не даст.
2. Замаскировало ли эту возможность предыдущее преобразование? Например, ЛНЗ может преобразовать 2 × a в операцию сдвига. Однако умножение коммутативно, а сдвиг – нет. Любое преобразование, нуждающееся в коммутативности, обнаружит, что после применения ЛНЗ возможность улучшения исчезла.

3. Устранили ли неэффективность другие преобразования? У преобразований бывают перекрывающиеся и неожиданные эффекты; например, ЛНЗ достигает некоторых эффектов глобального распространения констант, а последствия разворачивания цикла частично пересекаются с клонированием суперблока. Автор компилятора тем не менее мог бы включить оба преобразования ради тех эффектов, которые не пересекаются.

Взаимодействия между преобразованиями не позволяют точно предсказать, какое улучшение воспоследует в результате применения одного преобразования или их последовательности.

Исследователи изучали автоматизированные способы нахождения хороших последовательностей оптимизаций. Подходы различаются уровнем детализации и методиками. В разных системах последовательности ищутся на уровне блоков, на уровне исходных файлов и на уровне программы в целом. В большинстве систем используется тот или иной вид поиска в пространстве последовательностей оптимизаций.

В общем случае эти методы слишком дороги для практического использования в реальных программах, поскольку пространство потенциальных последовательностей слишком велико. Например, если компилятор выбирает последовательность 10 преобразований из возможных 15, то будет сгенерировано 10^{15} возможных последовательностей – исследовать их все невозможно. Поэтому компиляторы, которые все же ищут хорошие последовательности, применяют эвристические методы к выборочным подмножествам пространства поиска. Эти методы можно отнести к трем категориям: (1) генетические алгоритмы, приспособленные для проведения интеллектуального поиска, (2) рандомизированные алгоритмы поиска и (3) методы статистического машинного обучения. Все три подхода оказались многообещающими.

Несмотря на гигантское пространство поиска, хорошо настроенные алгоритмы могут находить хорошие последовательности оптимизаций за 100–200 попыток. Хотя и такое число еще слишком велико для практического использования, дальнейшее уточнение может уменьшить количество попыток до приемлемого уровня.

В этом контексте *хорошей* считается последовательность, которая дает результат, отличающийся от оптимального не более чем на 5 %.

Самое интересное приложение этих методов – вывести последовательности, активируемые при задании флагов командной строки компилятора, например `-O2`. Автор компилятора должен подобрать репрезентативные приложения для определения хороших общих последовательностей, а затем включить их в компилятор по умолчанию. Более агрессивный подход, применяемый в нескольких системах, – вывести несколько хороших последовательностей для разных групп приложений, поручить компилятору попробовать каждую из них и оставить лучший из полученных результатов.

10.8. Итоги и перспективы

Проектирование и реализация оптимизирующего компилятора – непростое предприятие. В этой главе мы познакомились с концептуальным подходом к рассуждениям о преобразованиях – таксономией эффектов. Каждая категория этой таксономии представлена несколькими примерами в этой главе или в других местах книги.

Проблемой для автора компилятора является выбор набора преобразований, которые будут хорошо работать совместно и порождать качественный код, отвечающий потребностям пользователя. Таксономия, представленная в этой главе, лежит в основе принятия таких решений. Конкретные преобразования, реализованные в компиляторе, в значительной степени определяют, для каких видов программы будет порождаться хороший код.

Замечания к этой главе

Хотя представленные в этой главе алгоритмы современные, многие фундаментальные идеи были хорошо известны уже в 1960–1970-х годах. Устранение мертвого кода, перемещение кода, снижение стоимости операций и устранение избыточности описаны в работах Аллена [12] и Кокке и Шварца [98]. В нескольких обзорных статьях описано состояние этой области в разные моменты времени [17, 29, 31, 328]. В книгах Моргана [277] и Мучника [279] обсуждаются проектирование, структура и реализация оптимизирующих компиляторов. Вольф [364] и Аллен и Кеннеди [21] акцентируют внимание на анализе на основе зависимостей и преобразованиях для векторных и параллельных вычислений.

Алгоритм *Dead*, реализующий устранение мертвого кода методом пометки и очистки, был впервые описан в работах Кеннеди [226, 228]. Он напоминает алгоритм пометки Шорра–Уэйта [319]. Наше изложение *Dead* основано на работе Цитрона и др. [120, раздел 7.1] с небольшими модификациями. Алгоритм *Clean* разработан и реализован в 1992 году Робом Шиллингсбургом [262].

ОПК усовершенствует классический алгоритм Мореля и Ренвуазе частичного устранения избыточности [276]. Эта статья вдохновила много улучшений [88, 141, 144, 333]. Алгоритм ОПК Кноопа, Рютинга и Штеффена [236] улучшил размещение кода; в разделе 10.3 используются уравнения из работы Дрехслера и Штаделя [145]. Бодик, Гупта и Соффа объединили этот подход с репликацией, чтобы находить и удалять весь избыточный код [47]. Алгоритм НЗД взят из диссертации Симпсона [59, 326]. Он реализован в целом ряде компиляторов.

Поднятие описано в каталоге Аллена–Кокке как метод уменьшения пространства кода [17]. Формулировка с использованием предсказуемости встречается в нескольких местах, включая статью Фишера и Лебланка [157]. Спуск, или перекрестный переход, описан в работе Вулфа и др. [368].

Идеи оконной оптимизации и устранения хвостовой рекурсии восходят к началу 1960-х годов. Оконная оптимизация впервые описана в статье Маккимана [268]. Устранение хвостовой рекурсии старше, ходят устные рассказы о том, что Маккарти описал ее на доске во время лекции в 1963 году. Диссертация Стила [335] – классическая ссылка на тему устранения хвостовой рекурсии.

Клонирование суперблока впервые описано в работе Ху и др. [211]. Такие оптимизации цикла, как расчленение и разворачивание, изучались во многих работах [21, 29]; Кеннеди использовал разворачивание, чтобы избежать операций копирования в конце цикла [225]. Цитрон, Лоури и Задек представили интересную альтернативу расчленению [121]. В работах Маккинли и др. приведены практические соображения о влиянии оптимизаций памяти на производительность [101, 269].

Комбинирование оптимизаций, как в SCCP [358, 359], часто приводит к улучшениям, которые невозможно получить применением исходных оптимизаций по отдельности [89, 91]. В алгоритме нумерации значений объединены устранение избыточности, распространение констант и упрощение алгебраических тождеств [59]. В ОПК устранение избыточностей и частичных избыточностей объединено с перемещением кода [236]. В работе Клика и Купера [91] объединен алгоритм разбиения Альперна [22] с SCCP [359]. Многие авторы объединяли распределение регистров с планированием команд [44, 54, 173, 278, 285, 286, 294, 318]. Но не все пары оптимизаций получают выигрыш от объединения [89, 91].

Снижение стоимости операций имеет богатую историю. Одно семейство таких алгоритмов появилось на основе работ Аллена, Кокке и Кеннеди [20, 95, 97, 227, 264]. Алгоритм OSR входит в это семейство [117]. Другое семейство алгоритмов выросло из подхода к оптимизации на основе анализа потока данных – примером может служить алгоритм ОПК; методы, принадлежащие этому семейству, описаны в ряде источников [138, 140, 142, 187, 220, 231, 237]. Версия OSR в разделе 10.7.2 упрощает только умножения. В статье Аллена и др. [20] показаны последовательности упрощения для многих других операторов; обобщить OSR на эти случаи несложно. Более слабая форма снижения стоимости предполагает замену целочисленного умножения более быстрыми операциями [251].

Упражнения

К разделу 10.1

1. Одна из основных функций оптимизатора – избавиться от накладных расходов, привнесенных компилятором в процессе трансляции исходного языка в ПП.

   a. Приведите четыре примера неэффективностей, которые, по вашему мнению, оптимизатор должен устранить, а также конструкции исходного языка, их породившие.

b. Приведите четыре примера неэффективностей, которые, по вашему мнению, оптимизатор пропустит. Объясните, почему оптимизатору трудно улучшить код в этих случаях.

К разделу 10.2

2. На рис. 10.1 показан алгоритм *Dead*. Проход пометки – классическое вычисление с неподвижной точкой.
   a. Объясните, почему это вычисление завершается.
   b. Является ли найденная им неподвижная точка единственной? Обоснуйте свой ответ.
   c. Выведите строгую верхнюю границу времени работы алгоритма.

3. Рассмотрите алгоритм *Clean* из раздела 10.2. Он удаляет бесполезные управляющие конструкции и упрощает ГПУ.
   a. Почему этот алгоритм завершается?
   b. Дайте оценку времени работы алгоритма.

К разделу 10.3

4. Устранение избыточности оказывает различные эффекты на генерируемый компилятором код.
   a. Как ОПК влияет на спрос на регистры в преобразуемом коде? Обоснуйте свой ответ.
   b. Как ОПК влияет на размер кода, сгенерированного для процедуры? (Можете предполагать, что спрос на регистры не изменяется.)
   c. Как поднятие влияет на спрос на регистры в преобразуемом коде? Обоснуйте свой ответ.
   d. Как поднятие влияет на размер кода, сгенерированного для процедуры? (Можете предполагать, что спрос на регистры не изменяется.)

К разделу 10.4

5. Простая форма снижения стоимости операций заменяет один экземпляр дорогой операции последовательностью более дешевых операций. Например, некоторые операции целочисленного умножения можно заменить последовательностью сдвигов и сложений.
   a. Какие условия должны быть выполнены, чтобы компилятор мог безопасно заменить целочисленную операцию x ← y × z одной операцией сдвига?
   b. Напишите набросок алгоритма, который заменяет перемножение известной константы и целого без знака последовательностью сдвигов и сложений.
   c. Как это преобразование можно было бы включить в алгоритм ЛНЗ?

6. И оптимизация хвостовой рекурсии, и встраивание стремятся уменьшить накладные расходы, связанные с соглашением о связях.
   a. Может ли компилятор встроить хвостовой вызов? Что этому мешает? Как можно было бы обойти препятствия?
   b. Сравните код, порожденный вашей модифицированной схемой встраивания, с тем, который порожден оптимизацией хвостовой рекурсии.

К разделу 10.5

7. Компилятор может искать и устранять избыточные вычисления многими способами. Среди них – алгоритмы НЗД и ОПК.

a. Приведите два примера избыточностей, которые НЗД устраняет, а ОПК не находит.

b. Приведите пример избыточности, которую ОПК находит и устраняет, а НЗД найти не может.

К разделу 10.6

8. Напишите набросок алгоритма, который переименовывает значения и переменные в процедуре таким образом, что в тождественности имен закодирована тождественность значений.

9. Клонирование суперблока может привести к значительному увеличению размера кода.

a. Как компилятор мог бы смягчить это увеличение, сохранив выигрыш настолько, насколько это возможно?

> **Указание:** вспомните алгоритм размещения блоков из главы 8.

b. Какие проблемы могли бы возникнуть, если бы оптимизатор разрешал продолжать клонирование суперблока через условный переход, замыкающий цикл?

Глава 11

Выбор команд

АННОТАЦИЯ

И фаза анализа компилятора, и оптимизатор работают с кодом в форме ПП. Чтобы создать исполняемый код, компилятор должен транслировать ПП в систему команд процессора. Этот процесс называется выбором команд. Поскольку типичный процессор предлагает несколько способов выразить большинство вычислений, селектор должен выбрать наилучшую последовательность из нескольких возможных.

В этой главе приведено введение в два разных подхода к выбору команд. Первый основан на оконной оптимизации, классической технике улучшения кода на поздней стадии. Во втором используются алгоритмы сопоставления с деревом-образцом для поиска множества операций, реализующих программу в форме ПП. Оба подхода поддержаны надежными инструментами. Оба широко распространены в реальных компиляторах.

КЛЮЧЕВЫЕ СЛОВА

Выбор команд, сопоставление с деревом-образцом, оконная оптимизация

11.1. Введение

Чтобы транслировать программу из промежуточного представления, например абстрактного синтаксического дерева или низкоуровневого линейного кода, в исполняемую форму, компилятор должен отобразить каждую конструкцию ПП в эквивалентную конструкцию в машинных командах целевого процессора. В зависимости от относительных уровней абстракции ПП и архитектуры системы команд (ISA) целевой машины эта трансляция может включать раскрытие деталей, скрытых в ПП программы, или объединение нескольких операций ПП в одну машинную команду. Конкретные решения, принимаемые компилятором, могут оказать прямое влияние на общую эффективность откомпилированного кода.

Сложность выбора команд проистекает из того факта, что в типичной ISA даже простые операции можно реализовать несколькими способами.

В 1970-х годах компьютер DEC PDP/11 имел компактную систему команд, поэтому хороший компилятор, например BLISS-11, мог произвести выбор команд на простом написанном вручную проходе. По мере усложнения ISA рост количества возможных способов реализовать заданную последовательность операций ПП стал неуправляемым. Этот комбинаторный взрыв привел к появлению систематических подходов к выбору команд, в частности представленных в этой главе, и инструментов для их реализации.

Концептуальная дорожная карта

Выбор команд, отображающих код из созданного компилятором ПП в команды целевой машины, – это задача сопоставления с образцом. В простейшем случае компилятор мог бы сопоставить одну последовательность ассемблерных команд каждой операции ПП. В итоге был бы порожден правильный, хотя и несколько шаблонный, код, неэффективно использующий ресурсы целевой машины. Лучше рассматривать несколько последовательностей команд для каждой операции ПП с учетом ее контекста и выбирать из них ту, для которой ожидаемая стоимость наименьшая.

В этой главе представлено два подхода к выбору команд: первый основан на оконной оптимизации, второй – на сопоставлении с деревом-образцом. В первом случае ПП транслируется в низкоуровневое линейное ПП, которое систематически улучшается, а затем отображается в команды целевой машины. Во втором случае ПП, генерируемое компилятором, и ISA целевой машины представляются древовидными структурами, и путем сопоставления с образцом ищется реализация ПП программы в терминах ISA. Оба метода могут порождать высококачественный код, адаптированный к локальному контексту. Оба включены в состав инструментов, которые принимают описание целевой машины и выдают работающий селектор команд.

Несколько слов о времени

Сам селектор команд работает на этапе компиляции и транслирует ПП программы, созданное фазой анализа и оптимизатором, в код для ISA целевой машины. Порожденный селектором код может еще не быть корректной ассемблерной программой; во многих компиляторах в нем еще предполагается, что запас виртуальных регистров не ограничен. Распределитель регистров, который работает после селектора команд, завершает задачу отображения промежуточного кода на набор регистров целевой ISA.

Во многих компиляторах используются селекторы команд на основе оконной оптимизации (раздел 11.3) или сопоставления с деревом-образцом (раздел 11.4). На этапе проектирования автор компилятора создает описания ПП компилятора и целевой ISA для этих инструментов. На этапе сборки кодогенератор анализирует эти описания и порождает код селектора команд, который затем компилируется и включается в состав компилятора. В этой главе нас будут интересовать алгоритмы этапа компиляции, осуществляющие выбор команд.

Краткий обзор

Систематические подходы к генерированию кода упрощают переориентацию компилятора. Цель – минимизировать усилия, необходимые для переноса компилятора на новый процессор. В идеале фаза анализа и оптимизатор нуждаются в минимальных изменениях, да и большую часть фазы синтеза можно использовать повторно. При такой стратегии инвестиции в сборку, отладку и сопровождение повторно используемых частей компилятора окупаются.

Выбор, планирование и распределение

Три основных процесса, составляющих фазу синтеза, – выбор команд, планирование выполнения и распределение регистров. Все три напрямую влияют на качество сгенерированного кода и взаимодействуют друг с другом. Выбор команд меняет задачу планирования. Он диктует как время выполнения операции, так и функциональные блоки, на которых она может выполняться. Планирование может повлиять на выбор команд. Если кодогенератор способен реализовать операции ПП одной из двух ассемблерных операций и эти операции потребляют разные ресурсы, то ему, возможно, требуется понимать конечный план, чтобы сделать наилучший выбор.

С распределением регистров выбор команд взаимодействует несколькими способами. Если целевой процессор располагает однородным набором регистров, то селектор команд может предполагать, что запас регистров не ограничен, и полагаться на то, что распределитель вставит операции загрузки и сохранения, необходимые для того, чтобы значения уместились в набор регистров. Если, с другой стороны, у целевой машины сложные правила использования регистров, то селектору, возможно, придется уделять пристальное внимание конкретным физическим регистрам. Это может затруднить выбор и предопределить некоторые или все решения, касающиеся распределения. В такой ситуации кодогенератор мог бы воспользоваться сопрограммой для локального распределения регистров в процессе выбора команд.

Разделив – насколько возможно – выбор, планирование и распределение, мы можем упростить реализацию, тестирование и отладку каждого процесса. Но поскольку каждый процесс может налагать ограничения на два других, автор компилятора должен знать о взаимодействиях, чтобы разделение процессов не ухудшило качество кода.

Значительная часть ответственности за поддержку нескольких целевых архитектур ложится на селектор команд. Типичный компилятор пользуется общим ПП для всех целей и в той мере, в какой это возможно, для всех поддерживаемых исходных языков. Он оптимизирует ПП, опираясь на предположения, которые верны для большинства или даже для всех целевых машин. Наконец, он использует фазу синтеза, в которой автор приложил все усилия для изоляции и выделения зависящих от целевой архитектуры деталей.

На практике новый язык может потребовать включения новых операций в ПП. Однако цель состоит в том, чтобы расширить существующее ПП, а не изобрести новое.

Хотя планировщику и распределителю регистров нужна архитектурно-зависимая информация, хороший дизайн может изолировать эти знания в конкретное описание целевой машины и ее ISA. Такое описание могло бы включать размер набора

регистров, описание каждой операции, количество, возможности и операционные задержки функциональных блоков, ограничения на выравнивание памяти и соглашение о вызове процедур. Затем алгоритмы планирования и распределения параметризуются этими характеристиками системы и используются для различных ISA и систем.

Таким образом, ключом к переносимости компилятора является реализация селектора команд. Селектор получает созданное компилятором ПП и порождает ассемблерный код для целевой машины. Переносимый селектор команд состоит из движка сопоставления с образцом вкупе с информацией о ПП, ISA и отображении между ними.

Генератор фаз синтеза иногда называют *генератором кодогенераторов*.

Если мы умеем автоматизировать конструирование движка сопоставления, то конструирование селекторов команд на основе спецификации может следовать той же базовой модели, что для сканеров и парсеров. В этих системах автор компилятора создает описание синтаксиса и действий, а затем вызывает генератор сканеров или парсеров для построения той или иной компоненты компилятора. Аналогично автор компилятора может создать описание целевой машины и вызвать *генератор фаз синтеза* для построения селектора команд. Этот генератор запускается автономно на этапе сборки.

При таком подходе стоимость и сложность выбора команд скрыты в генераторе фаз синтеза. Как и в случае LR-парсеров, мы можем позволить себе использовать в генераторе алгоритмы, требующие больше времени, чем те, что работают на этапе компиляции. Фактически основанные на инструментах селекторы команды могут быть чрезвычайно эффективны во время компиляции.

Второй ключ к переносимости компилятора – изолировать машинно-зависимый код в той мере, в какой это возможно. В идеале весь машинно-зависимый код должен быть сосредоточен в селекторе команд, планировщике и распределителе регистров; к сожалению, реальность почти всегда далека от этого идеала. Некоторые машинно-зависимые детали неизбежно прокрадываются в более ранние части компилятора. Например, ограничения на выравнивание в записях активации (ЗА) могут зависеть от целевой машины, что изменяет смещения значений, хранящихся в ЗА. У компилятора может возникнуть необходимость представлять такие вещи, как предикатное выполнение, слоты отсрочки ветвлений и операции над несколькими словами памяти явно, чтобы ими можно эффективно воспользоваться. И все же перенос машинно-зависимых деталей на уровень выбора команд может сократить количество изменений в других частях компилятора при переносе на новый целевой процессор.

На этапе проектирования автор компилятора должен решить, с каким объемом настройки кода можно смириться в фазе синтеза. Мы полагаем, что селектор команд должен отыскать хорошее локальное отображение кода, порожденного оптимизатором, на систему команд целевой машины. Основная ответственность за эффективность ложится на оптимизатор, а не

на фазу синтеза. Логика и опыт подсказывают, что следует отделять цель оптимизации вычислений от цели эффективного отображения вычислений на архитектуру целевой машины.

В этой главе рассматривается два подхода к построению селекторов команд. В разделе 11.2 мы приведем предварительные сведения, связанные с выбором команд, и опишем сквозной пример этой главы. В двух следующих разделах будут представлены различные способы применения метода сопоставления с образцом к преобразованию последовательностей кода ПП в ассемблерный код. В разделе 11.3 изложен подход на основе оконной оптимизации, а в разделе 11.4 – подход на основе сопоставления с деревом-образцом. Оба метода основаны на описаниях. Автор компилятора создает описание целевой ISA, а затем инструмент строит селектор, который будет использоваться на этапе компиляции. Тот и другой методы успешно использовались в переносимых компиляторах.

11.2. Предварительные сведения

Селектор команд транслирует код из формы ПП в машинные команды. Если уровни абстракции ПП и ISA схожи, то такая трансляция может пройти безболезненно – как в случае трансляции ILOC-кода на простую RISC-машину. Если ПП более абстрактно, чем ISA, то селектор должен предоставить недостающие детали – как в случае отображения близкого к исходному коду АСД на микропроцессор для бытовых устройств. Наконец, если ПП менее абстрактна, чем ISA, как в случае языка межрегистровых пересылок GCC, то селектору придется объединять несколько операций ПП в одну машинную.

Авторы компилятора тоже вносят свою лепту. Автор должен выбрать компромисс между сложностью фазы анализа компилятора и производительностью откомпилированного кода. Для порождения тонко настроенного и оптимизированного кода требуется более сложный компилятор, чем для сравнительно медленного шаблонного кода. В зависимости от контекста оба подхода имеют право на существование.

В главе 5 обсуждались вопросы, возникающие, когда компилятор транслирует исходный код в начальное ПП. Выбор команд отличается от генерирования ПП в двух важных отношениях. Во-первых, результатом генерирования ПП должна быть версия программы, которая хорошо *компилируется* и *оптимизируется*, тогда как в результате выбора команд должна получиться версия, которая хорошо *работает*. Во-вторых, автор компилятора контролирует дизайн ПП, что может упростить задачу отображения кода на ПП. Что же касается целевой ISA, то это фиксированный язык, спроектированный другими людьми и располагающий ограниченными ресурсами. Из-за этих различий при генерировании кода в фазе синтеза используются более сложные подходы и алгоритмы, чем при генерировании ПП в фазе анализа.

Сложность выбора команд связана не с конкретной методологией или алгоритмом, а с самой природой проблемы – процессоры обычно предла-

гают несколько способов реализовать конструкцию ПП, и у каждого своя стоимость и свои ограничения. Кодогенератор должен сделать выбор, исходя из знаний о стоимости операций и окружающем контексте. Если выбор удачный, то процесс должен породить эффективный код, который хорошо настроен под окружающие операции.

Сравнение хорошо работающего и хорошо оптимизируемого кода

Различие между генерированием ПП на ранней стадии компиляции и генерированием ассемблерного кода на поздней связано с предполагаемым использованием кода. Цель ПП программы – упростить оптимизацию и генерирование кода. Цель ассемблерного кода – обеспечить быстрое выполнение. Чтобы понять, почему эти различные цели определяют различные последовательности кода, рассмотрим следующий абстрактный цикл:

```
for i = 1 to 1000
    for j = 1 to 1000
        ... A[i,j] ...
```

В разделе 7.3.2 было показано два способа вычислить адрес `A[i,j]` в предположении, что массив `A` определен как `A`[low_1:$high_1$, low_2:$high_2$] и хранится по строкам, начиная с адреса `@A`:

(1) `@A` + (`i` − low_1) × len_2 × w + (`j` − low_2) × w;

(2) `@A` + (`i` × len_2 × w) + (`j` × w) − (low_1 × len_2 × w + low_2 × w).

В полиномиальной форме (1) меньше операций; она исполняется быстрее альтернативы. В полиномиальной форме (2) разделены подвыражения, зависящие от `i` и `j`, и отдельно присутствует член, содержащий только константы. Хороший оптимизатор мог бы вынести член `i` из внутреннего цикла и вычислить постоянный член на этапе компиляции. Таким образом, форма (1) хорошо работает, а форма (2) хорошо оптимизируется.

Основанные на спецификации инструменты могут перенести большую часть сложности генерирования последовательностей команд с этапа компиляции на этапы проектирования и сборки. Такие инструменты автоматически конструируют эффективные селекторы команд, которые справляются со сложностью как в ПП, так и в ISA. Они могут упростить конструирование эффективной фазы синтеза точно так же, как генераторы сканеров и парсеров упрощают конструирование фазы анализа.

11.2.1. Влияние дизайна ISA на выбор команд

Напомним, что ILOC-код в наших примерах предельно простой.

Сложность выбора команд в значительной степени является следствием свойств системы команд типичной целевой машины. Селектор должен учитывать два фактора, вызывающих комбинаторный рост числа вариантов: несколько механизмов решения одной и той же задачи и изобилие режимов адресации в командах доступа к памяти и арифметических операций.

Подобные возможности усложняют выбор команд, потому что расширяют контекст, увеличивают сложность и количество вариантов. Автоматизированные методы построения селектора команд появились в значительной степени как ответ на рост сложности процессоров. Благодаря более точному описанию отображения ПП на ISA и алгоритмическому преобразованию этого описания в код инструменты упрощают задачу построения эффективных и мощных селекторов.

Множественность реализаций

Если бы целевая ISA предлагала только один способ выполнения каждой операции ПП, то компилятор мог бы просто заменить каждую операцию ПП эквивалентной последовательностью машинных команд. Но в большинстве контекстов целевая машина предлагает несколько способов реализации каждой конструкции ПП.

Рассмотрим, к примеру, простую операцию копирования $r_i \rightarrow r_j$. Предположим, что целевой процессор использует ILOC в качестве системы команд. Язык ILOC прост, но даже на его примере можно продемонстрировать сложность генерирования кода. В очевидной реализации $r_i \rightarrow r_j$ использовалась бы команда `i2i` $r_i \Rightarrow r_j$; такое копирование регистра в регистр обычно является одной из самых дешевых процессорных операций. Но и в других реализациях нет недостатка, например:

```
addI ri,0  ⇒ rj      subI ri,0    ⇒ rj      multI ri,1   ⇒ rj
divI ri,1  ⇒ rj      lshiftI ri,0 ⇒ rj      rshiftI ri,0 ⇒ rj
and  ri,ri ⇒ rj      orI ri,0     ⇒ rj      xorI ri,0    ⇒ rj
```

Например, в процессоре ARM V8 имеется специальный нулевой регистр `XZR`.

И этим возможности еще не исчерпываются. Если у процессора имеется регистр, значение которого всегда равно 0, то будет работать еще целый набор операций с использованием `add`, `sub`, `lshift`, `rshift`, `or` и `xor`. Если включить в рассмотрение последовательности двух операций, то выбор еще расширится.

Программист отбросил бы большинство этих альтернатив, если не вообще все. Операция копирования регистра в регистр типа `i2i` простая, быстрая и очевидная. Но автоматизированный процесс должен рассмотреть все возможности и их стоимость. Тот факт, что ISA может достичь нужного эффекта несколькими способами, затрудняет выбор команд. Даже в ILOC, особенно простой ISA, есть много способов выполнить операцию копирования.

Режимы адресации

Во всех арифметических операциях, перечисленных как способы выполнить копирование, предполагалась модель адресации регистр-регистр. Некоторые ISA предоставляют команды, для которых один или более операндов находятся в памяти. А такие команды, как загрузка и сохранение, всегда

адресуют память. Для повышения эффективности в ISA введено понятие *режима адресации*, которое включает наиболее распространенные способы вычисления адреса.

RISC, CISC и выбор команд

Первые поборники архитектур RISC предполагали, что их системы команд приведут к упрощению компиляторов. В ранних RISC-машинах, таких как IBM 801, было намного меньше режимов адресации, чем в современных им CISC-машинах, например серии DEC VAX-11. В них присутствовали операции вида регистр-регистр и отдельные операции сохранения и загрузки для перемещения данных между регистрами и памятью. С другой стороны, VAX-11 позволял использовать в качестве операндов регистры и память; для многих операций поддерживалась двух- и трехадресная форма. RISC-машины действительно упростили выбор команд. Они предоставляли меньше способов реализовать данную операцию. И у них было меньше ограничений на использование регистров. Однако архитектура загрузки и сохранения повышала важность распределения регистров.

Напротив, у CISC-машин были операции, инкапсулирующие гораздо более сложную функциональность в одну операцию. Чтобы эффективно использовать эти операции, селектор должен распознавать более длинные паттерны в более длинных фрагментах кода. Это повышает важность систематического выбора команд; автоматизированные методы, описанные в этой главе, более важны для CISC-машин, но применимы и к RISC-машинам.

ILOC предлагает несколько режимов адресации для операций `load` и `store`.

Операция	Семантика
`load` $r_1 \Rightarrow r_2$	MEMORY(r_1) $\Rightarrow$ r_2
`loadAI` $r_1, c_2 \Rightarrow r_3$	MEMORY($r_1 + c_2$) $\Rightarrow$ r_3
`loadAO` $r_1, r_2 \Rightarrow r_3$	MEMORY($r_1 + r_2$) $\Rightarrow$ r_3
`loadI` $c_1 \Rightarrow r_2$	$c_1 \Rightarrow r_2$

Операции сохранения поддерживают режимы непосредственной адресации (`AI`) и адресации со смещением (`AO`). Арифметические и логические операции также поддерживают непосредственный режим (`I`), в котором один операнд является литеральной константой, встроенной в поток команд.

Реальные процессоры часто поддерживают более широкий и сложный набор режимов адресации, например арифметические операторы с адресацией память-память, одно- и двухадресные операции, которые перезаписывают некоторые свои входные аргументы, и операции с неявными аргументами, например вершиной аппаратного стека. Селектор должен эффективно использовать все режимы адресации, поддерживаемые ISA.

Аналогичные проблемы возникают с управляющими конструкциями. Условные и безусловные переходы могут поддерживать абсолютные адреса, относительные адреса и адреса с различной длиной в битах. У этих операций разная длина, и они могут занимать разное число тактов. Поскольку выбор лучшего варианта условного или безусловного перехода зависит от многих факторов, включая расстояние и направление между начальным и конечным

адресами, выбор режима адресации в операции условного перехода требует особого внимания.

Уровень абстракции

Индивидуальные операции в ILOC имеют относительно низкий уровень абстракции. Современные процессоры часто предоставляют смесь низкоуровневых и более сложных операций. Например:

- операция перемещения строки позволяет программе легко описать сложную последовательность, включающую неявный итеративный цикл;
- операция вызова процедуры может автоматизировать крупные части последовательности вызова, в т. ч. управления регистрами, сохраняемыми вызывающей стороной;
- операция сложения с умножением для чисел с плавающей точкой может использовать меньше циклов и меньше регистров для вычисления $(r_i \times r_j) + r_k$, чем операции сложения и умножения порознь;
- операция «загрузить несколько» или «сохранить несколько» может перемещать значения в несколько соседних регистров или из них.

Наличие подобных операций может означать, что селектор команд должен объединять несколько низкоуровневых операций в одну высокоуровневую.

Использование регистров

Например, в архитектуре IA-32 операция умножения 16-разрядных целых со знаком читает один аргумент из регистра `AX`, а 32-разрядный результат записывает в регистры `AX` и `DX`.

Иногда процессоры ограничивают использование регистров странными способами (см. раздел 13.2.4). Операции могут ожидать, что один или несколько операндов находятся в определенных местах. Некоторые процессоры требуют, чтобы значения двойной точности хранились в паре регистров одинарной точности, которая начинается регистром с четным номером. При использовании деструктивной одно- или двухадресной операции значение одного из операндов может быть перезаписано, что усложняет выделение и использование регистров (см. раздел 4.4).

Стоимость

У каждой операции имеется стоимость. В большинстве современных машин простые операции типа сложения и сдвига занимают один такт. А такие операции, как загрузка или деление, могут выполняться дольше. Величина задержки умножения и деления может зависеть от битового состава операндов. Задержка операции с памятью зависит от многих факторов, в т. ч. от текущего состояния подсистемы памяти процессора. Задержка условного перехода может зависеть от того, насколько хорошо процессор умеет предсказывать ее исход.

В большинстве случаев автор компилятора хочет, чтобы фаза синтеза порождала быстрый код. Но возможны и другие метрики. Например, если конечный код работает на устройстве, питающемся от аккумулятора, то

компилятор мог бы рассмотреть типичное энергопотребление каждой операции; разные операции потребляют разное количество энергии. Компилятор, пытающийся анализировать энергопотребление, мог бы использовать совершенно другие метрики, чем тот, что оптимизирует скорость. Аналогично если больше важно потребление памяти, то автор компилятора мог бы назначать последовательности операций стоимость, зависящую только от ее длины. Или же он мог бы просто не рассматривать последовательности нескольких операций, достигающие того же эффекта, что одна операция.

Поскольку более короткая последовательность кода выбирает меньше байтов из оперативной памяти, уменьшение длины кода заодно экономит энергию.

Хотя выбор команд может играть важную роль при определении качества кода, автор компилятора должен помнить о гигантском размере пространства поиска, которое предстоит исследовать селектору. Даже скромная по размерам система команд может породить пространства поиска, состоящие из миллионов операций. Очевидно, что компилятор не может позволить себе исчерпывающий поиск в таких пространствах. Методы, которые будут описаны ниже, исследуют альтернативные последовательности в строго дисциплинированном порядке и либо ограничивают поиск, либо заранее вычисляют достаточно информации, чтобы сделать глубокий поиск эффективным.

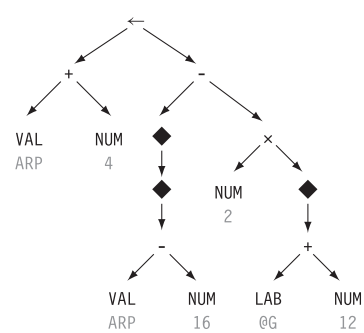
11.2.2. Поясняющий пример

Выбор команд методами оконной оптимизации и сопоставления с деревом-образцом поясняется на одном и том же примере. В обоих случаях мы изучаем, как сгенерировать код простого предложения присваивания: a ← b - 2 × c. На рис. 11.1 этот пример показан в двух разных ПП. На рисунке (a) приведена форма с четверками; с нее начинается обсуждение выбора команд методом оконной оптимизации. На рисунке (b) показано низкоуровневое АСД этого предложения; оно используется при обсуждении сопоставления с деревом-образцом.

Этот пример впервые был показан на рис. 4.1.

Op	Arg_1	Arg_2	Result
×	2	c	t_1
-	b	t_1	a

(a) Четверки



(b) Низкоуровневое АСД

Рис. 11.1 Низкоуровневое ПП для a ← b - 2 × c

В примере используются три переменные с разными способами вычисления адреса: a – локальная переменная, располагающаяся со смещением 4 от УЗА; b – переданный по ссылке параметр, располагающийся со смещением –16 от УЗА; c – глобальная переменная.

Вычислить адрес c сложно. Значение расположено со смещением 12 от метки global. Поскольку метке сопоставляется полноразмерный адрес, результат его разрешения хранится в некотором глобальном пуле констант. Пул констант начинается с адреса @CP, а смещение от его начала задается величиной @G. Мы предполагаем, что @CP слишком велик для размещения непосредственно в операции add или sub, но достаточно мал для размещения в поле непосредственного операнда операции loadI.

Поскольку пример состоит из одного исходного предложения, никакого потока управления в нем нет. Таким образом, мы будем предполагать, что значения кодов условий, определяемые операциями, не используются и потому мертвы.

Четверки

На рис. 11.1(a) показан наш пример кода, выраженный в классической форме с четверками. Имена переменных выглядят как ссылки, а дополнительные детали хранятся в таблице символов. Операции – простой трехадресный код. Сгенерированное компилятором временное имя используется для передачи результата умножения в операцию вычитания. Порядок четверок определяет порядок выполнения.

В этом представлении мало явных деталей. Предполагается, что в таблице символов хранятся данные о типе и местоположении именованных значений (см. разделы 4.5 и 5.4). Как мы увидим в разделе 11.3, селектор команд на базе локальной оптимизации расширяет этот код, чтобы сделать все необходимые детали явными и видными в самом ПП.

Низкоуровневое АСД

Чтобы раскрыть достаточно деталей для выбора команд, в АСД на рис. 11.1(b) выбран низкий уровень абстракции. Несколько узлов дерева нуждаются в пояснении.

Константы представлены узлами трех типов:

- узел типа NUM представляет константу, которая помещается в поле непосредственного операнда команды с тремя операндами (например, multI или loadAI);
- узел типа CON, не показанный в примере, представляет константу, которая достаточно мала, чтобы поместиться в loadI, но слишком велика для NUM;
- узел типа LAB представляет перемещаемый символ, обычно ассемблерную метку кода или данных.

Различие между этими видами констант критично для выбора команды. Значение в узле типа CON или LAB не может встретиться в качестве непо-

средственного операнда операции `multI`. Значение, представленное узлом типа `LAB`, предполагается слишком большим для операции `loadI`; компилятор хранит такие значения в пуле констант в памяти, расположенном начиная с адреса `@CP`.

Смысл двух других узлов этого дерева не очевиден. Узел типа `VAL` представляет значение, которое заведомо находится в регистре, например УЗА, хранящийся в регистре r_{arp}, или результат вычисления общего подвыражения, найденного оптимизатором. Узел ◆ обозначает уровень косвенности; его сыновним узлом является адрес, и он порождает значение, хранящееся по этому адресу.

Низкоуровневые детали АСД позволяют селектору команд адаптировать свои решения к конкретному контексту. Поддеревья, описывающие адресные выражения для `a`, `b` и `c`, выглядят похоже; как мы увидим, они генерируют совершенно разный код из-за конкретных базовых адресов и смещений. Адаптируя конечный ассемблерный код к контексту, компилятор может породить эффективный код для каждого поддерева.

11.2.3. Ситуативное сопоставление

Автор компилятора может построить селектор команд на базе написанного вручную ситуативного сопоставителя. Такой селектор распознавал бы отдельные конструкции ПП или небольшие множества взаимосвязанных конструкций и транслировал их прямо на язык ассемблера (ASM) целевой машины. В случае древовидного ПП такой подход мог бы вылиться в простой обратный порядок обхода, как в синтаксически управляемых схемах трансляции из главы 5. В случае линейного ПП автор компилятора мог бы написать селектор, который выполняет линейное сканирование ПП и генерирует код для каждой операции.

Селекторы команд, построенные на базе ситуативного сопоставления, обычно порождают единообразный шаблонный код. Чтобы лучше использовать особенности ISA целевой машины, нужно включить в рассмотрение контекст из ПП программы. Селектор может эффективно задействовать сложные режимы адресации, только когда строит какое-то представление, явное или неявное, вычисления адреса; он может воспользоваться этой моделью, чтобы выбрать наилучшую операцию загрузки или сохранения. Воспользоваться операциями с непосредственной адресацией селектор может, только если понимает, пусть и ограниченно, поток постоянных значений в коде и абсолютные величины этих значений.

Наконец, ситуативное сопоставление слабо поддерживает (или не поддерживает вовсе) перенос компилятора на другой процессор. Переносимость – главное обоснование отделения селектора команд от оптимизации, распределения регистров и планирования команд. Инструментальный подход к конструированию селекторов команд на основе спецификации должен упростить задачу переноса компилятора на новый процессор или систему.

ИТОГИ РАЗДЕЛА

Селектор команд транслирует код в форме ПП в ассемблерный код целевой машины. Для каждой конструкции ПП он должен выбрать эффективную реализацию на языке ассемблера, адаптированную к окружающему контексту. Сам селектор должен работать быстро.

Выбрать команды трудно из-за очень большого числа вариантов, с которыми сталкивается селектор. Он должен делать выбор из нескольких операций. Он должен эффективно использовать режимы адресации целевой машины, должен моделировать ограничения на использование ресурсов процессора и гарантировать, что сгенерированный код их соблюдает. Поскольку выбор влияет на эффективность откомпилированного кода, селектор играет очень важную роль для производительности приложения.

ВОПРОСЫ

1. Перечислите как можно больше способов, которыми в ILOC можно поместить в регистр r_i целое значение 1.
2. Рассмотрите операцию `move` $r_i \Rightarrow r_k$, которая копирует четыре байта из памяти по адресу r_i в память по адресу r_k, потребляя на один такт больше, чем эквивалентная операция `load`. Какие преимущества дает такая операция?

11.3. Выбор команд с помощью оконной оптимизации

Оконная (peephole) оптимизация – метод, применяемый на последней стадии оптимизации, который был разработан в 1960–1970-е годы. В некоторых компиляторах оконная оптимизация выступает в роли парадигмы выбора команд. В них используется селектор, который транслирует ПП программы в низкоуровневое ПП (НУПП), систематически применяет к нему оконную оптимизацию, а затем использует сопоставление, чтобы отобразить НУПП в команды целевой машины. Оконная оптимизация собирает факты, необходимые для эффективного использования возможностей процессора. Она также систематически устраняет мелкие неэффективности, которые оптимизация на более высоком уровне абстракции могла пропустить.

11.3.1. Оконная оптимизация

Основная предпосылка успеха оконной оптимизации проста: компилятор может эффективно находить возможности для локальных улучшений, исследуя короткие последовательности соседних операций. На первый взгляд, эта предпосылка противоречит высказанному ранее утверждению о том, что оптимизатор хорошо справился со своей работой. Это тонкий момент. Возможности, найденные в ходе оконной оптимизации, часто находятся на более низком уровне абстракции, чем ПП, видное оптимизатору.

Изначально предполагалось, что оконный оптимизатор работает на последнем проходе компилятора – читает и записывает ассемблерный код. Оптимизатор перемещал скользящее окно, или «глазок», по коду. На каждом шаге он исследовал попавшие в окно операции в поисках конкретных паттернов, которые мог бы улучшить. Найдя подходящий паттерн, он заменял

код более эффективной последовательностью команд. Благодаря ограниченности набора паттернов и размера окна эти оптимизаторы работали быстро.

Классический пример паттерна – сохранение, за которым следует загрузка из того же адреса в памяти. Загрузку можно заменить операцией копирования.

```
storeAI r1    ⇒ rarp, 8          ⇒  storeAI r1 ⇒ rarp,8
loadAI rarp,8 ⇒ r15                 i2i r1      ⇒ r15
```

Неэффективности, похожие на эту, часто возникают из-за трансляции предложение за предложением, а также как результат других оптимизаций, например устранения мертвого кода и свертывания констант. Другие паттерны, пригодные для улучшения в процессе оконной оптимизации, включают алгебраические тождества, например

```
subI r2,0  ⇒ r7
mult r4,r2 ⇒ r10   ⇒  mult r4,r7 ⇒ r10
```

и случаи, когда целью операции перехода является другая операция перехода:

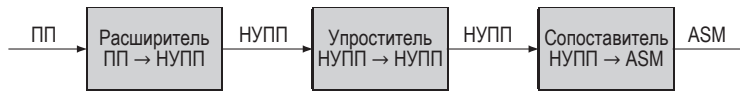
```
    jumpI → L10          jumpI → L11
L10: jumpI → L11   ⇒  L10: jumpI → L11
```

Если при этом устраняется последний переход на метку `L10`, то блок с меткой `L10` становится недостижимым и может быть устранен. К сожалению, чтобы доказать, что операция с меткой `L10` недостижима, нужно больше анализа, чем обычно возможно на стадии оконной оптимизации (см. раздел 10.2.2).

В ранних оконных оптимизаторах использовался ограниченный набор написанных вручную паттернов. Они производили исчерпывающий поиск, чтобы найти паттерны, но работали быстро, потому что использовали небольшой набор паттернов и окно малого размера – обычно на две или три операции.

С тех пор оконная оптимизация шагнула вперед и уже не ограничивается небольшим числом паттернов. Более сложные ISA открыли дополнительные возможности, что привело к более систематическому подходу к оптимизации. Современные оконные оптимизаторы улучшают код благодаря систематическому применению символической интерпретации и упрощения.

Оконный селектор команд строится на этой философской основе. Он разбивает процесс выбора на три задачи: расширение, упрощение и сопоставление.



Структурно эта система выглядит как компилятор. Она принимает на входе созданное компилятором ПП, а на выходе порождает код, который напоминает язык ассемблера целевой машины (ASM). Предположим, что селектор обрабатывает по одному простому блоку за раз.

Мы осторожно говорим «напоминает», потому что селектор команд может порождать код, в котором используются виртуальные, а не физические регистры.

Расширитель транслирует ПП в последовательность низкоуровневых операций (НУПП), которая улавливает все эффекты блока – по крайней мере те из них, что влияют на поведение программы. Например, если операция `add` r_i, $r_j \Rightarrow r_k$ устанавливает код условия, то код НУПП должен записать $r_i + r_j$ в r_k и установить правильное значение кода условия. Обычно расширитель переписывает операции на управляемом шаблоном проходе, игнорируя контекст.

Упроститель проходит по НУПП. Он сдвигает по НУПП небольшое окно – скажем, на три-четыре операции. В этом окне он пытается улучшить код путем систематического применения четырех преобразований: прямая подстановка, алгебраическое упрощение, вычисление константных выражений и устранение мертвых эффектов, таких как создание неиспользуемых кодов условий. Он подвергает все детали кода НУПП единому уровню локальной оптимизации.

Сопоставитель сравнивает упрощенное НУПП с библиотекой паттернов и ищет паттерн, который лучше всего улавливает все эффекты в НУПП. Конечная последовательность операций может производить эффекты, помимо обязательных; например, она может создавать новые, хотя и мертвые, значения кодов условий. Однако она должна сохранять эффекты, необходимые для корректности. Она не может устранять живое значение независимо от того, где оно хранится: в памяти, в регистре или в таком неявно устанавливаемом месте, как код условия.

Напомним, что `a` находится по адресу `ARP+4`, `b` – по адресу `ARP-16`, а `c` – со смещением 12 от `@G`.

На рис. 11.2 показано, как этот подход мог бы работать для примера на рис. 11.1(а). Все начинается в левом верхнем углу, где показан код в форме с четверками. Этот код вычисляет `a` ← `b - 2 × c`. Расширитель создает НУПП программы, показанное в правом верхнем углу. Упроститель сводит этот код к форме, показанной в правом нижнем углу. На основе этого НУПП сопоставитель строит ILOC-программу, показанную в левом нижнем углу. В этой последней программе для реализации вычисления `a` ← `b - 2 × c` используется десять операций.

11.3.2. Упроститель

Упроститель – сердце оконного оптимизатора. Сначала он заполняет окно из потока НУПП. Затем он пытается выполнить прямую подстановку определенных значений в места использования далее в окне. Он применяет алгебраическое упрощение и свертывание констант. Если в результате любого из этих преобразований какая-то операция в окне становится мертвой, то он устраняет ее и подтягивает вниз окна новую.

Когда упроститель больше не может улучшить код в окне, он генерирует код первой операции в окне и подтягивает новую операцию вниз окна. В результате окно «скользит» вниз по блоку, по одной операции за раз.

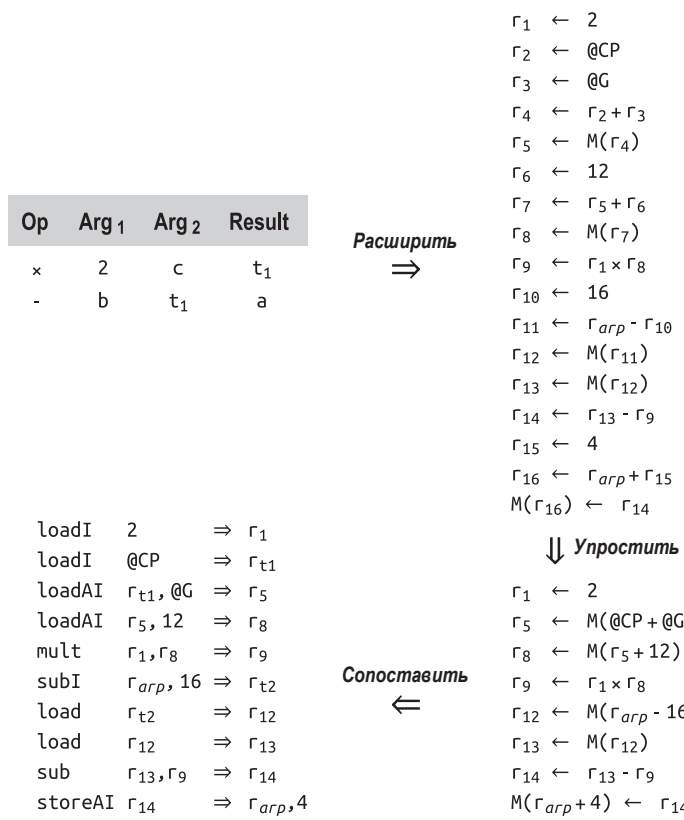


Рис. 11.2 Применение расширения, упрощения и сопоставления к примеру

На рис. 11.3 показано содержимое окна упростителя на каждом шаге примера. Предполагается, что окно вмещает три операции. Последовательность 1 отражает окно с первыми тремя операциями. Упрощение в нем невозможно. Упроститель выводит $r_1 \leftarrow 2$ и отбрасывает первую операцию. Затем он помещает вниз окна четвертую операцию, которая определяет r_4, создавая тем самым последовательность 2.

В последовательности 2 упроститель может выполнить прямую подстановку r_2 и r_3 в операцию сложения, в результате чего определения r_2 и r_3 оказываются мертвыми. Упроститель отбрасывает их и подтягивает следующие две операции в окно. В последовательности 3 он вставляет определение r_4 в операцию загрузки, отбрасывает определение r_4 и подтягивает следующую операцию в окно. Процесс продолжается. Во врезке показаны операции, которые генерирует упроститель, а также номер последовательности, после которой операция сгенерирована. Окончательный код на рис. 11.2 состоит именно из этих операций.

```
 1  r1 ← 2
 6  r5 ← M(@CP + @G)
 7  r8 ← M(r5 + 12)
10  r9 ← r1 × r8
11  r12 ← M(rarp - 16)
12  r13 ← M(r12)
15  r14 ← r13 - r9
16  M(rarp + 4) ← r14
```

Код, сгенерированный упростителем

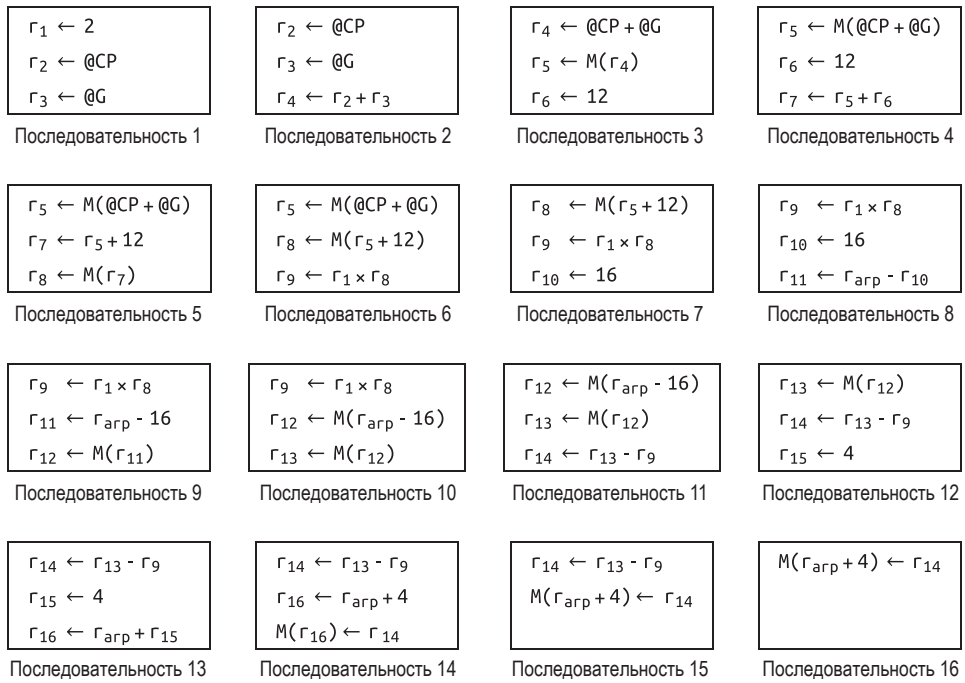


Рис. 11.3 Последовательности, порождаемые упростителем

На способность локального селектора улучшать код влияет несколько факторов. Способность обнаружить, что значение мертво, играет критическую роль для упрощения. Размер окна ограничивает способность оптимизатора объединять взаимосвязанные операции. Например, если бы окно было больше, то упроститель смог бы свернуть константу 2 в операцию умножения. Обработка операций потока управления определяет, что происходит на границах блока.

Распознавание мертвых значений

```
r12 ← 2
r14 ← r12 + r12
```

Сталкиваясь с последовательностью типа показанной на полях, упроститель может заменить использования r_{12} в операции сложения на 2 и вычислить сумму. Однако он не может исключить определение r_{12}, если не уверен, что r_{12} мертв после использования в операции сложения. Таким образом, способность распознать, когда значение мертво, критически важно для работы упростителя.

Анализ живучести обсуждается в разделах 8.6.1 и 9.2.2.

Компилятор может вычислить множества LiveOut для каждого блока, а затем, на обратном проходе по блоку, проследить, какие значения живы на каждой операции. Альтернативно он может воспользоваться идеей, лежащей в основе построения полуусеченной SSA-формы: идентифицировать имена, которые используются более чем в одном блоке, и считать, что каждое такое имя живо при выходе из каждого блока. Эта стратегия по-

зволяет избежать затрат на анализ живучести; она корректно идентифицирует любое значение, строго локальное в блоке, где оно определено. Все последствия работы расширителя строго локальны, поэтому этот более дешевый подход должен работать хорошо.

Этот анализ следует производить поочередно с расширением, так чтобы его результаты отражались на коде НУПП.

Зная либо множества LiveOut, либо множество глобальных имен, расширитель может пометить места последних использований в НУПП. Это возможно благодаря двум наблюдениям. Во-первых, расширитель может обрабатывать блоки снизу вверх; расширение – это простой процесс, управляемый шаблоном. Во-вторых, обходя блок снизу вверх, расширитель может строить множество значений, живых на каждой операции, LiveNow.

Вычислить LiveNow просто. Расширитель полагает начальное значение LiveNow равным либо множеству LiveOut для блока, либо множеству глобальных имен. Алгоритм обходит операции в блоке снизу вверх. На каждой операции $r_i \leftarrow r_j$ op r_k он удаляет из LiveNow r_i и добавляет r_j и r_k. На каждом шаге этот алгоритм порождает множество LiveNow не менее точное, чем начальная информация, использованная в конце блока.

На машине, где для управления условными переходами используются коды условий, многие операции устанавливают значение кода условия. В типичном блоке многие такие значения мертвы. Расширитель должен вставлять явные присваивания коду условия. Упроститель должен понимать, когда значение кода условия мертво, потому что лишние присваивания могут помешать сопоставителю сгенерировать некоторые последовательности команд.

Например, рассмотрим вычисление $r_i \times r_j + r_k$. Если × и + устанавливают код условия, то последовательность двух операций могла бы сгенерировать НУПП, показанное на полях. Первое присваивание cc мертвое. Если упроститель устранит это присваивание, то сможет объединить другие операции в операцию умножения со сложением в предположении, что в системе команд таковая имеется. Но если он не сможет устранить $cc \leftarrow f_x(r_i, r_j)$, то сопоставитель не сможет использовать умножение со сложением, потому что она не устанавливает cc дважды.

$r_{t1} \leftarrow r_i \times r_j$
$cc \leftarrow f_x(r_i, r_j)$
$r_{t2} \leftarrow r_{t1} + r_k$
$cc \leftarrow f_+(r_{t1}, r_k)$

НУПП для $r_i \times r_j + r_k$

Физические и логические окна

До сих пор мы говорили об окне, содержащем соседние операции в НУПП. Такой подход интуитивно понятен физически и делает всю концепцию конкретной. Однако соседние операции в НУПП могут оперировать не одними и теми же значениями. На самом деле, поскольку целевые машины реализуют все больше параллелизма на уровне команд, фаза анализа и оптимизатор должны генерировать ПП программ, содержащие как можно больше независимых чередующихся вычислений, чтобы функциональные блоки целевой машины не простаивали. В таком случае оконный селектор сможет найти очень мало возможностей для улучшения кода.

Чтобы улучшить ситуацию, оконный селектор может использовать логическое окно вместо физического. Логическое окно позволяет рассматривать операции, связанные потоком значений в коде, т. е. рассматривать вместе операции, которые определяют и используют одно и то же значение. Это открывает возможность объединять и упрощать взаимосвязанные операции, даже если они не соседствуют в коде.

В процессе расширения оптимизатор может связать каждое определение со следующим использованием указанного в нем значения в блоке. Упроститель использует эти связи для заполнения окна. Достигнув операции *i*, упроститель строит окно для *i*, подтягивая операции, связанные с результатом *i*. Использование логического окна в коде повышает эффективность упростителя, сокращая время компиляции и количество операций, оставшихся после упрощения.

В нашем примере упроститель не может свернуть константу 2 в операцию умножения, потому что обе операции никогда не оказываются в окне вместе. Но в упростителе с логическим окном операции умножения и непосредственной загрузки присутствовали бы в одном окне, поэтому упроститель смог бы подставить значение 2 в умножение и исключить операцию непосредственной загрузки.

Мы можем распространить эту идею на несколько блоков. Компилятор может попытаться упростить логически соседние операции, даже если они находятся в разных блоках. Использование разных блоков требует глобального анализа, который установит, каких мест использования может достичь каждое определение, как, например, делает анализ достигающих определений из раздела 9.2.4. Кроме того, упроститель должен распознать, что одно определение может достигать нескольких мест использования, а одно использование может ссылаться на значения, вычисленные в нескольких разных определениях. Таким образом, упроститель не может просто объединить определяющую операцию с одним использованием и бросить остальные операции на произвол судьбы. Он должен либо ограничиться рассмотрением простых ситуаций, например одиночным определением и одиночным использованием или несколькими использованиями с одним определением, либо произвести детальный анализ, чтобы выяснить, какие комбинации безопасны и целесообразны. Ограничение логического окна расширенными простыми блоками позволяет избежать некоторых из этих осложнений.

11.3.3. Сопоставитель

Последний компонент оконного селектора – сопоставитель, который отображает произвольную последовательность операций НУПП на ISA целевой машины. В оконных селекторах использовались различные технологии для реализации сопоставителя, от адаптированных к задаче парсеров до напи-

санного вручную кода. Выбор подхода сильно зависит от размера и сложности целевой системы команд и доступности инструментов.

Цель сгенерированного селектора команд – уменьшить объем работы, необходимой для переноса компилятора. При наличии управляемого спецификациями сопоставителя автор компилятора создает описание ISA и его связи с НУПП; затем на этапе сборки генератор строит сопоставитель, который будет работать на этапе компиляции. Автор компилятора должен гарантировать, что (1) описание покрывает все множество возможных последовательностей НУПП и (2) в описании эффективно используются возможности целевой ISA, в частности режимы адресации и специализированные операции.

Располагая написанным вручную сопоставителем, автор компилятора может модифицировать сопоставитель напрямую, так чтобы он генерировал подходящие операции целевой ISA. Эта задача звучит пугающе, однако различий между ISA процессоров обычно меньше, чем сходных черт. Логические и арифметические операции работают похоже во многих архитектурах; различаются же они способом именования операндов и побочными эффектами, влияющими на состояние системы (например, установкой кода условия).

Оконные селекторы использовались в нескольких важных системах компиляции. В системе Gnu GCC НУПП, называемая языком межрегистровых пересылок (RTL), используется для некоторых оптимизаций и генерирования кода. В фазе синтеза оконный селектор применяется для преобразования RTL в язык ассемблера целевой машины. Этап сопоставления интерпретирует RTL как деревья и использует сопоставитель с деревом-образцом, построенный на основе описания целевой машины. С другой стороны, сопоставитель в оконном оптимизаторе VPO Дэвидсона пользуется парсером, который рассматривает НУПП как линейную форму и генерирует машинный код в процессе разбора. Он строит селектор на основе описания целевой ISA и собственных знаний об используемой в системе НУПП.

Парсер должен обрабатывать неоднозначности, появляющиеся из-за наличия нескольких способов реализовать заданную последовательность операций НУПП.

Эти основанные на оконной оптимизации системы имеют два фундаментальных достоинства:

- систематическое упрощение на поздней стадии позволяет компилятору устранять неэффективности, оставшиеся после более ранних проходов трансляции;
- прямая подстановка и свертывание констант открывают возможности для отображения последовательностей операций НУПП на процессорные режимы адресации.

В итоге мы получаем программу, которая выполняет меньше операций и лучше использует особенности системы команд целевой машины.

ИТОГИ РАЗДЕЛА

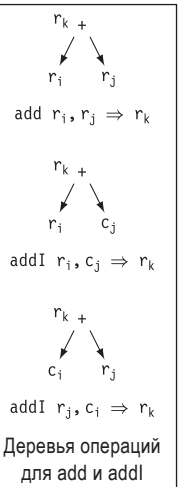
Оконная оптимизация может лечь в основу быстрого и эффективного селектора команд. Селектор состоит из управляемого шаблоном расширителя, который транслирует созданное компилятором ПП в более детальное низкоуровневое ПП, упростителя, который применяет прямую подстановку, алгебраические упрощения, распространение констант и устранение мертвого кода к небольшому скользящему окну, и сопоставителя, который отображает оптимизированное низкоуровневое ПП в целевую ISA.

Сила этого подхода – в упростителе; он удаляет межоперационные неэффективности, появившиеся в процессе трансляции. В сфере его внимания возможности, включающие локальные значения; многие из них на более ранних этапах трансляции не видны. Улучшения, которых удается в результате достичь, иногда вызывают удивление. Последний этап сопоставления прямолинейный; на нем используются разные технологии: от написанного вручную кода до LR-парсеров.

ВОПРОСЫ

1. Напишите набросок алгоритма упростителя. Какова сложность алгоритма? Как размер окна «глазка» влияет на временные границы?
2. Как можно было бы обобщить упроститель, чтобы он использовал алгебраические тождества на рис. 8.3 для улучшения кода НУПП?

11.4. Выбор с помощью сопоставления с деревом-образцом



Деревья операций для add и addI

Другой способ справиться со сложностью выбора команд дают инструменты сопоставления с деревом-образцом. Чтобы преобразовать генерирование кода в задачу сопоставления с деревом-образцом, ПП программы и систему команд целевой машины следует представить в виде деревьев. Компилятор может использовать низкоуровневое АСД в качестве детальной модели компилируемого кода. Похожие деревья он может использовать для представления операций целевого процессора. Например, операции сложения в ILOC можно было бы смоделировать деревьями операций типа показанных во врезке для `add` и `addI`. Систематически сравнивая такие деревья операций, или деревья-образцы, с поддеревьями АСД, компилятор может найти все потенциально возможные реализации поддерева.

Имея низкоуровневое АСД для кода и набор деревьев операций для целевой ISA, сопоставитель строит *плиточное покрытие* АСД деревьями операций. Плиткой (tile) называется пара $\langle x, y \rangle$, где x – узел АСД, а y – корень дерева операций. Покрытием называется множество пар, удовлетворяющее следующим ограничениям:

1) множество плиток покрывает все узлы АСД;

2) корень каждого дерева операций перекрывается с листом другого дерева операций, если только он не перекрывается с корнем АСД;
3) два перекрывающихся дерева операций *совместимы*, т. е. согласованы по классу хранения и типу значений;
4) любые два дерева операций в множестве могут перекрываться только по одному узлу.

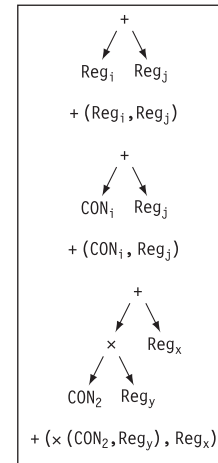
Множество пар, удовлетворяющее этим условиям, называется покрытием, оно *реализует* АСД.

Присутствие пары $\langle x, y \rangle$ в покрытии означает, что поддерево АСД с корнем в x можно реализовать деревом операций с корнем в y. Если узел x не является листовым, то выбор y будет зависеть от нахождения реализаций для поддеревьев x, которые работают с y. Чтобы построить покрытие, компилятор должен гарантировать, что дерево в целом и любое его поддерево можно реализовать заданным множеством деревьев операций.

Имея покрытие, реализующее АСД, компилятор может сгенерировать ассемблерный код в процессе обхода АСД снизу вверх. Следовательно, ключом к практическому использованию селекторов команд на основе сопоставления с деревом-образцом являются алгоритмы нахождения хороших покрытий АСД. Существуют различные инструменты такого рода: сопоставление деревьев, сопоставление текстов и системы восходящего переписывания (bottom-up rewrite systems – BURS). Инструменты назначают стоимость деревьям операций и порождают покрытия с минимальной стоимостью. В этом разделе мы опишем сопоставление деревьев, но основные идеи применимы также к сопоставлению текстов и BURS.

11.4.1. Представление деревьев

Нам необходима нотация для описания АСД и деревьев-образцов. Для этой задачи хорошо подходит префиксная нотация. Например, операция `add`, показанная во врезке, в префиксной форме записывается как + (Reg_i, Reg_j), а операция `addI` – как + (CON_i, Reg_j). Та же нотация пригодна для низкоуровневых АСД, представляющих исполняемый код, и деревьев-образцов, представляющих операции целевой машины. В этой главе мы ограничимся только примерами операций над целыми числами. Обобщение правил на другие типы данных привносит много новых образцов, но мало новых идей.



Деревья НУПП и их префиксные описания

Операндами поддерева являются поддеревья или листья. Поддеревья записываются в префиксной нотации, как в случае поддерева умножения в АСД, показанного во врезке: + (× (CON_2, Reg_y), Reg_x). Листьям назначаются символические имена, в которых закодирована информация о типе и местоположении операнда. Например, Reg_i обо-

значает значение, находящееся в регистре, в CON_j – постоянное значение. В АСД такие обозначения, как VAL, NUM, LAB и CON, дают более детальную информацию (см. стр. 605). Для обеспечения уникальности в имена включаются индексы. Мы можем следующим образом переписать АСД на рис. 11.1(b) в префиксной форме:

```
← (+ (VAL1, NUM1),
   -(◆(◆(- (VAL2, NUM2))), × (NUM3,◆(+ (LAB1, NUM4)))))
```

Хотя изображение в виде дерева интуитивно понятнее, такая линейная префиксная форма несет в точности ту же информацию.

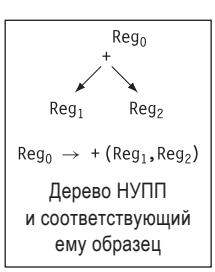
В этом разделе мы будем предполагать, что в ПП используются виртуальные имена регистров, т. е. что запас регистров не ограничен (см. раздел 1.4). После выбора команд распределитель регистров отобразит виртуальные имена на ограниченное множество физических регистров целевой машины (см. главу 13).

11.4.2. Правила переписывания

Для построения селектора команд на основе сопоставления с деревом-образцом автор компилятора создает множество правил переписывания, которые отображают АСД на ISA целевой машины. Каждое правило состоит из дерева-образца, шаблона кода и ассоциированной стоимости. Эти образцы описывают структуру АСД. На рис. 11.4 показано подмножество ILOC, которое мы будем использовать в последующих примерах.

Арифметические операции		Операции с памятью	
add	$r_1,r_2 \Rightarrow r_3$	store	$r_1 \Rightarrow r_2$
addI	$r_1,c_2 \Rightarrow r_3$	storeAO	$r_1 \Rightarrow r_2,r_3$
sub	$r_1,r_2 \Rightarrow r_3$	storeAI	$r_1 \Rightarrow r_2,c_3$
subI	$r_1,c_2 \Rightarrow r_3$	loadI	$c_1 \Rightarrow r_3$
rsubI	$r_2,c_1 \Rightarrow r_3$	load	$r_1 \Rightarrow r_3$
mult	$r_1,r_2 \Rightarrow r_3$	loadAO	$r_1,r_2 \Rightarrow r_3$
multI	$r_1,c_2 \Rightarrow r_3$	loadAI	$r_1,c_2 \Rightarrow r_3$

Рис. 11.4 Подмножество ILOC



Образцы похожи на продукции контекстно-свободной грамматики (КСГ). У каждого образца есть левая часть (ЛЧ) и правая часть (ПЧ); как и в КСГ, ЛЧ и ПЧ разделены символом вывода, →. Во врезке показаны дерево трехрегистрового сложения и соответствующий ему образец. Метки содержат имена различных определений и использований. Входными аргументами являются Reg_1 и Reg_2. Выходом операции является Reg_0.

Некоторые символы встречаются только в правых частях образцов. Они аналогичны терминальным символам в КСГ – представляют конкретные символы, которые могут быть листьями дерева. Другие символы встречаются и в левой, и в правой части образца. Они аналогичны нетерминальным символам КСГ – это синтаксические переменные, которые используются для связывания продукций в последовательности.

Набор правил включает взаимосвязанные правила, которые совместно могут покрыть дерево. На рис. 11.5 показан набор правил для низкоуровневого АСД, описанного в разделе 11.2.2. Каждое правило представляет небольшое АСД. Например, правило 8 было ранее показано во врезке. Его правая часть описывает узел + и его сыновние узлы. Левая часть правила соответствует метке в узле +: в дереве во врезке это Reg_0.

	Продукция	Стоимость	Шаблон кода
0	Goal → Goal Stmt	0	
1	Goal → Stmt	0	
2	Stmt → ← (Reg_1,Reg_2)	3	store $r_2 \Rightarrow r_1$
3	Stmt → ← ($T1_1$,Reg_2)	3	storeAO $r_2 \Rightarrow T1.r_1$,$T1.r_2$
4	Stmt → ← ($T2_1$,Reg_2)	3	storeAI $r_2 \Rightarrow T2.r$,$T2.n$
5	Reg → ◆(Reg_1)	3	load $r_1 \Rightarrow r_{new}$
6	Reg → ◆($T1_1$)	3	loadAO $T1.r_1$,$T1.r_2 \Rightarrow r_{new}$
7	Reg → ◆($T2_1$)	3	loadAI $T_2.r$,$T_2.n \Rightarrow r_{new}$
8	Reg → +(Reg_1,Reg_2)	1	add r_1,$r_2 \Rightarrow r_{new}$
9	Reg → +(Reg_1,$T3_2$)	1	addI r_1,T3 $\Rightarrow r_{new}$
10	Reg → +($T3_1$,Reg_2)	1	addI r_2,T3 $\Rightarrow r_{new}$
11	Reg → -(Reg_1,Reg_2)	1	sub r_1,$r_2 \Rightarrow r_{new}$
12	Reg → -(Reg_1,$T3_2$)	1	subI r_1,T3 $\Rightarrow r_{new}$
13	Reg → -($T3_1$,Reg_2)	1	rsubI r_2,T3 $\Rightarrow r_{new}$
14	Reg → ×(Reg_1,Reg_2)	2	mult r_1,$r_2 \Rightarrow r_{new}$
15	Reg → ×(Reg_1,$T3_2$)	2	multI r_1,T3 $\Rightarrow r_{new}$
16	Reg → ×($T3_1$,Reg_2)	2	multI r_2,T3 $\Rightarrow r_{new}$
17	Reg → CON_1	1	loadI $CON_1 \Rightarrow r_{new}$
18	Reg → NUM_1	1	loadI $NUM_1 \Rightarrow r_{new}$
19	Reg → LAB_1	4	loadI @CP $\Rightarrow r_{new1}$
			loadAI r_{new1},@L $\Rightarrow r_{new2}$
20	Reg → VAL_1	0	
21	T1 → + (Reg_1,Reg_2)	0	
22	T2 → + (Reg_1,$T3_2$)	0	
23	T2 → + ($T3_1$,Reg_2)	0	
24	T3 → NUM_1	0	

Рис. 11.5 Правила переписывания для покрытия низкоуровневого АСД ILOC-кодом

Правило 19 сталкивается с общей проблемой: коду нужно загрузить значение константы, слишком большое для встраивания в операцию непосредственной загрузки. Шаблон кода предлагает способ решения проблемы. Предполагается, что в каждой процедуре имеется уникальный, статиче-

Правило 19 слишком простое. На практике компилятор мог бы разместить пул констант по адресу с известным смещением от точки входа в процедуру. Код пролога либо загрузил бы @CP в регистр, либо сохранил бы его в области локальных данных.

ски инициализированный пул констант, начало которого обозначено символом @CP. Далее предполагается, что @L – положительное смещение LAB_1 от @CP. В этих предположениях сгенерированный код загружает @CP в регистр и использует его как базовый адрес для операции loadAI.

Правила на рис. 11.5 описывают набор потенциальных АСД для списка присваиваний. Не все правила порождают код. Например, правила 0 и 1 создают упорядоченный список предложений *Stmt* – нетерминальный символ для предложения присваивания. Правила 20–23 обрабатывают выбор режима адресации, а правило 24 используется в операциях с непосредственным режимом адресации.

Взаимодействия между образцами, закодированные в использовании символов в ЛЧ, определяют способы комбинирования поддеревьев. Например, в продукциях 5–20 Reg встречается в ЛЧ. Следовательно, каждое из этих правил описывает поддерево, которое может переписать Reg.

Символы в левой части кодируют знания о типе и классе хранения представленных ими значений. Например, Reg представляет значение в регистре. Reg может появиться из поддерева, которое порождает целое значение, хранящееся в регистре, как в правилах 5–16. Он также может появиться из CON, NUM или LAB в результате применения продукций 17, 18 и 19. Правило 20 дает способ преобразования узла VAL в Reg – по сути дела, оно именует значение. VAL могло бы быть глобальным значением, например ARP (УЗА) или результатом вычисления, произведенного в непересекающемся поддереве, такого как избыточное выражение, найденное оптимизатором. T1 и T2 представляют адреса – значения, которые могут быть вычислены в таких режимах адресации, как loadAO и loadAI. T3 представляет NUM, используемое как непосредственный операнд.

Наконец, заметим, что набор правил неоднозначен. У правил 9 и 22 одинаковые правые части, равно как и у правил 10 и 23. У этих правил разные цели и разные стоимости. Правило 9 представляет явную операцию сложения общего назначения, тогда как правило 22 – операцию сложения, которая выполняется оборудованием вычисления адресов в памяти.

Оценки стоимости

Некоторые системы допускают только фиксированную стоимость. Другие позволяют изменять стоимость во время сопоставления с учетом ранее выбранных вариантов.

У каждого правила есть стоимость; эта стоимость должна давать реалистичную оценку затрат на выполнение указанного в шаблоне кода. На рис. 11.5 предполагается, что сложение и вычитание занимают один такт, умножение – два, а операции с памятью – три такта, это разумная оценка при попадании в кеш первого уровня. Для правил, не генерирующих кода, таких как правило 22, стоимость нулевая. Оценка стоимости направляет выбор из альтернативных правил.

Полный набор правил мог бы включать правила, требующие более одной машинной операции, как правило 19. Алгоритм сопоставления с деревом-

образцом видит такое правило как один образец. При условии что оценки стоимости точны, присутствие последовательностей из нескольких операций не должно приводить к осложнениям.

Построение набора правил

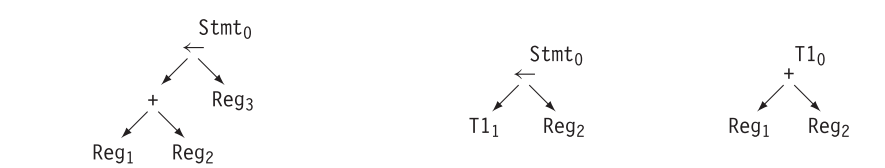
Распространенная стратегия построения набора правил – начать с минимального набора, покрывающего дерево. То есть автор компилятора может вывести набор правил, включающий образец для каждого узла АСД и правильную последовательность для каждого образца. После того как этот набор правил спроектирован, протестирован и отлажен, можно добавлять правила с образцами, в которых используются специализированные операции, для обработки более сложных случаев.

Ограничения на набор правил

Чтобы упростить код сопоставителя с деревом-образцом, мы введем два ограничения на набор правил. Во-первых, *каждый образец должен включать не более одного оператора*. Любое правило с несколькими операторами можно переписать в виде последовательности однооператорных правил.

Рассмотрим два набора правил для операции ILOC `storeAO`, показанных на рис. 11.6. На рисунке (a) показано одно правило, ← (+ (Reg_1,Reg_2),Reg_3). Оно специализирует общую операцию сохранения, описанную правилом 2, для случая, когда адрес является суммой двух регистров. На рисунке (b) показано два однооператорных образца, которые совместно покрывают то же поддерево. Для создания однооператорных правил автор компилятора ввел новое имя `T1`, которое встречается только при вычислении адреса в `storeAO` или `loadAO`. Правила на рисунках (a) и (b) эквивалентны; они покрывают одно и то же множество поддеревьев НУПП.

> Этот частный случай важен, если оборудование может вычислять сумму в устройстве обработки адресов, как в случае команды `storeAO`.



(a) Образец для `storeAO` (b) Однооператорные образцы для `storeAO`

Рис. 11.6 Однооператорные и многооператорные образцы

Ограничение одного оператора значительно упрощает код сопоставителя с деревом-образцом. Для сопоставления с образцом на рисунке (a) сопоставитель должен проверить вместе узлы + и ← либо переносить дальше явное

состояние, чтобы прослеживать такие соответствия. Любой подход увеличивает сложность и стоимость реализации.

В случае двух однооператорных образцов на рисунке (b) сопоставитель принимает чисто локальные решения. Поддерево АСД `+(Reg,Reg)` сопоставляется с общим правилом, описывающим `add` (правило 8), и с правилом, описывающим `T1` (правило 21). Поднимаясь к родителю этого поддерева, сопоставитель может породить сопоставления с правилом 2 для метки левого потомка `Reg` и с правилом 3 для метки левого потомка `T1`.

Ключ к тому, чтобы заставить однооператорные правила порождать желаемые результаты для сложных команд целевой машины, лежит в структуре стоимости набора правил. В случае однооператорных образцов сумма стоимостей однооператорных правил должна быть равна стоимости сложной команды. Если стоимость операции равна одному такту и для сопоставления с ней нужно два образца, то им можно назначить произвольные неотрицательные стоимости, в сумме дающие единицу, например ½ и ½ или ¼ и ¾ или 0 и 1. При таком подходе гарантируется, что сопоставление с низкой стоимостью генерирует последовательность команд с наименьшей локальной стоимостью.

Отметим, что в правиле 18 есть шаблон кода, а в правиле 24 – нет.

Второе ограничение заключается в том, что *листья дерева встречаются только в синглтонных правилах*, каковыми являются правила 17–20 и 24. И в этом случае цель – упростить сопоставитель. Если требуется переместить `NUM` в регистр, то с задачей справляется правило 18, `Reg` ← `NUM`. Появление `NUM` в поле непосредственного операнда сопоставляется с правилом 24, `T3` ← `NUM`. Здесь `T3` – символ в левой части, который встречается только в поле непосредственного операнда таких операций, как `addI` или `loadAI`. Если бы целевая ISA поддерживала несколько длин непосредственных операндов, то набор включал бы правила с символом для каждой длины в левой части.

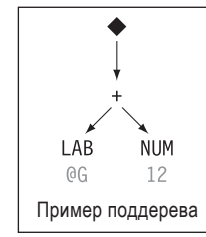
Второе ограничение позволяет избежать специальных случаев в коде сопоставителя с деревом-образцом. Рассматривая сыновние узлы оператора, сопоставитель может предполагать, что они уже аннотированы информацией о типе и стоимости посредством синглтонных правил. Код операторов остается простым и единообразным.

Любое ограничение или сразу оба можно отбросить ценой усложнения сопоставителя. В случае рукописного сопоставителя простота приводит к эффективным решениям. В случае таблично управляемой схемы автоматического генерирования возможны различные стоимостные компромиссы.

11.4.3. Вычисление покрытий

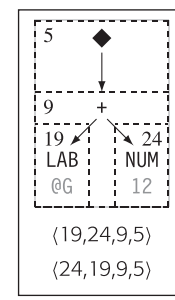
Имея набор правил, который отображает созданное компилятором ПП в ISA целевой машины, селектор команд должен отобразить конкретное АСД в последовательность команд. Для этого кодогенератор на основе сопоставите-

ля с деревом-образцом строит покрытие АСД образцами из набора правил. Имеется несколько методов построения такого покрытия, все они похожи концептуально, но различаются в деталях.



Чтобы лучше разобраться с покрытием, рассмотрим АСД во врезке, повторяющее поддерево на рис. 11.1(b). Оно описывает операцию с памятью, которая загружает слово по адресу со смещением 12 от метки `@G`. Пользуясь правилами на рис. 11.5, мы можем найти последовательность правил, покрывающую это дерево.

На рис. 11.7 показан один из способов покрыть это АСД. Правила трактуются как переписывания дерева. Как показано на рисунке (a), правило 19 сопоставляется с узлом `LAB`. Оно переписывает `LAB` в виде `Reg`, как показано на рисунке (b). Правило 24 сопоставляется с узлом `NUM` и переписывает его в виде `T3`, порождая АСД на рисунке (c). Одним из вариантов сопоставления с `+(Reg,T3)` на рисунке (c) является правило 9, которое переписывает это поддерево в виде `Reg`, как показано на рисунке (d). Правило 5 сопоставляется с АСД на рисунке (d) и переписывает его в виде одного узла `Reg` (не показано). Мы обозначаем эту последовательность перезаписей ⟨19, 24, 9, 5⟩; порядок в последовательности соответствует порядку применения правил.



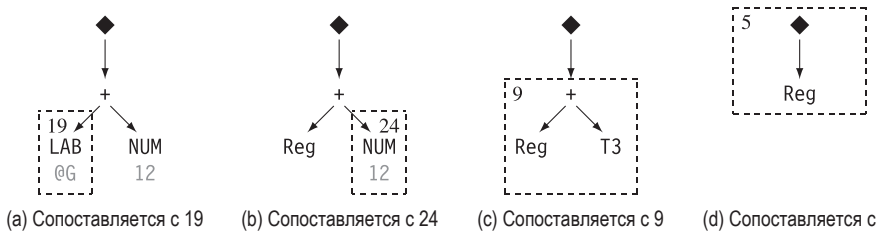


Рис. 11.7 Последовательность перезаписи простого дерева

Правила 19 и 24 можно применять в любом порядке. Следовательно, последовательность ⟨19, 24, 9, 5⟩ порождает то же покрытие, что ⟨24, 19, 9, 5⟩. Диаграмма во врезке показывает эти последовательности.

Это покрытие реализует АСД; оно отвечает всем четырем критериям, определенным на стр. 616–617. Оно покрывает каждый узел АСД одним или двумя узлами дерева операций. Корень каждого дерева-образца перекрывается листом его родителя. Например, корень дерева 19 – это лист дерева 9. Связи между образцами совместимы и ограничены одним узлом в каждом поддереве. Если два узла дерева операций покрывают узел АСД, то связанные узлы имеют одну и ту же нетерминальную метку. Таким образом, это покрытие реализует АСД.

Имея пригодное покрытие, селектор генерирует код, пользуясь шаблонами, ассоциированными с каждым правилом. Шаблон кода состоит из нуля

или более операций, которые совместно реализуют поддерево, покрытое правилом. Например, из последовательности ⟨19, 24, 9, 5⟩ вытекают четыре операции. Правило 19 порождает `loadI`, за которой следует `loadAI`; правило 24 не генерирует никакого кода; правило 9 генерирует `addI`, а правило 5 генерирует `load`. Результирующий код показан во втором столбце на рис. 11.8, где при создании потока значений используются абстрактные имена регистров. Заметим, что последовательность ⟨24, 19, 9, 5⟩ порождает тот же самый код, потому что правило 24 не порождает никакого кода. В общем случае последовательности, различающиеся порядком применения правил, могут генерировать операции в другом порядке.

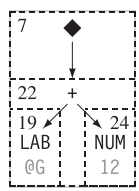
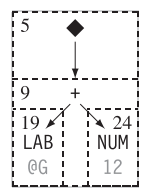
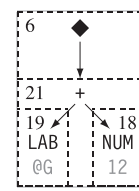
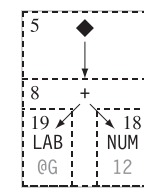
Покрытие				
Код	`loadI @CP` ⇒ r_i `loadAI` r_i`,@G` ⇒ r_j `loadAI` r_j`,12` ⇒ r_k Стоимость: 7 тактов	`loadI @CP` ⇒ r_i `loadAI` r_i`,@G` ⇒ r_j `addI` r_j`,12` ⇒ r_k `load` r_k ⇒ r_l Стоимость: 8 тактов	`loadI @CP` ⇒ r_i `loadAI` r_i`,@G` ⇒ r_j `loadI 12` ⇒ r_k `loadAO` r_j`,` r_k ⇒ r_l Стоимость: 8 тактов	`loadI @CP` ⇒ r_i `loadAI` r_i`,@G` ⇒ r_j `loadI 12` ⇒ r_k `add` r_j`,` r_k ⇒ r_l `load` r_l ⇒ r_m Стоимость: 9 тактов
Последовательности	⟨19,24,22,7⟩ ⟨24,19,22,7⟩	⟨19,24,9,5⟩ ⟨24,19,9,5⟩	⟨19,18,21,6⟩ ⟨18,19,21,6⟩	⟨19,18,8,5⟩ ⟨18,19,8,5⟩

Рис. 11.8 Множество всех покрытий поддерева из примера

Выбор одного из нескольких покрытий

Последовательности, в которых используются оба правила 18 и 19 – ⟨18, 19⟩ и ⟨19, 18⟩, – будут генерировать операции `loadI` в разном порядке.

АСД из нашего примера реализуют восемь разных последовательностей. На рис. 11.8 показаны все восемь; те, что различаются только порядком, показаны вместе. В верхней строке показаны покрытия, в средней – код, сгенерированный покрытием, и его стоимость, а в нижней – последовательности для каждого покрытия.

Чтобы сгенерировать код, селектор команд должен выбрать одну последовательность. Очевидный выбор – взять последовательность с наименьшей стоимостью. Если оценки стоимости правил отражают фактическую стоимость во время выполнения, то последовательность с наименьшей стоимостью будет самой быстрой. Отметим, однако, что стоимости могут отражать не скорость выполнения, а другие свойства. Например, в приложении, где потребление памяти критично, больше подходит метрика, равная длине последовательности кода в байтах.

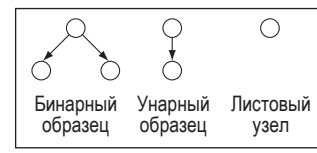
Алгоритм покрытия

На рис. 11.9 показан простой рекурсивный алгоритм, который вычисляет множество соответствий для каждого узла дерева. Чтобы упростить изложение, алгоритм предполагает, что АСД и набор правил состоят из узлов с нулем, одним или двумя операндами.

```
Tile(n)                          / * n – узел АСД * /
  if n листовый then
    Match(n,* ) ← { правила, реализующие n }
  else if n – унарный узел then
    Tile(child(n))
    Match(n,*) ← ∅          / * очистить множества Match для n * /
    for каждого правила r, где operator(r) = operator(n) do
      if (child(r), child(n)) совместимы then
        добавить r в Match(n, class(r))
  else if n – бинарный узел then
    Tile(left(n))
    Tile(right(n))
    Match(n,*) ← ∅          / * очистить множества Match для n * /
    for каждого правила r, где operator(r) = operator(n) do
      if (left(r), left(n)) and (right(r), right(n)) совместимы then
        добавить r в Match(n, class(r))
```

Рис. 11.9 Вычислить все сопоставления, чтобы покрыть АСД

Это ограничение позволяет алгоритму иметь дело только с тремя видами деревьев-образцов: бинарный узел и два его потомка, унарный узел и его потомок и листовый узел. В результате сопоставитель получается компактным и эффективным.



Функция *Tile* аннотирует каждый узел вектором правил, которые сопоставляют поддерево с корнем в этом узле. В векторе имеется по одному элементу для каждого символа в левой части, встречающегося в наборе правил. По самому замыслу, каждому символу в левой части соответствует пара ⟨класс хранения, тип⟩. Для узла *n* множество *Match(n,*)* содержит все правила, которые могут реализовать поддерево с корнем в *n*. *Match(n,c)* содержит все правила, которые могут реализовать поддерево с корнем *n*, породив результат *c*.

Для листового узла множество соответствий можно вычислить заранее. В нашем наборе правил для узла `NUM` множество *Match* всегда имеет вид, показанный во врезке. Правила предлагают один способ переписать `NUM` как `Reg` (правило 18) и один способ переписать `NUM` как `T3` (правило 24). У каждого листового узла свой собственный вектор.

Reg	Stmt	T1	T2	T3
18				24

Множество *Match* для узла `NUM`

Для унарного узла *n Tile* сначала рекурсивно переходит к потомку *n*, чтобы найти, какими способами он может быть реализован. Затем она исследует каждое правило *r*, реализующее оператор в узле *n*. Если у потомка име-

ется реализация, совместимая с *r*, помеченная в его множестве *Match*, то *Tile* добавляет *r* в *Match(n, class(r))*, где *class(r)* обозначает символ в левой части правила *r*.

Для проверки совместимости правила *r* с реализациями потомка узла АСД *n* алгоритм пользуется простым тестом, основанным на определении совместимости: правило и АСД должны иметь одинаковые операторы и класс хранения, а также тип значения потомка. Поддерево АСД $o_1(x)$ и дерево-образец $o_2(y)$ совместимы тогда и только тогда, когда (1) операторы в o_1 и o_2 одинаковы и (2) множество *Match(x, class(y))* не пусто. Первое условие гарантирует, что в дереве-образце и в поддереве АСД используется одна и та же операция, а второе – что потомок в АСД может быть реализован правилом, порождающим тот вид значений, который используется в правиле *r*.

Для бинарного узла *n Tile* следует тому же плану, что для унарного. Она рекурсивно вызывает себя для его левого и правого потомков в АСД и проверяет совместимость левого потомка с соответствующим термом в правиле *r* и правого потомка с соответствующим ему термом в *r*. Если в *r* используется тот же оператор, что в *n*, и *r* совместимо с покрытиями, вычисленными для потомков *n*, то *Tile* добавляет *r* в *Match(n, class(r))*.

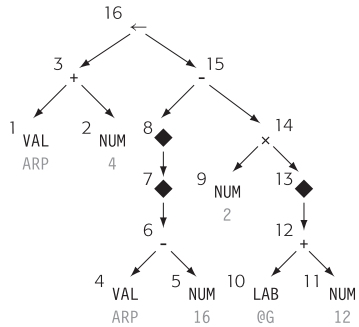
Проработанный пример

На рис. 11.10 показаны результаты применения *Tile* к нашему примеру. На рисунке (a) показано АСД для `a ← b - 2 × c`. Узлы АСД аннотированы их номерами в порядке обратного обхода; *Tile* обходит дерево в обратном порядке. На рисунке (d) показаны все соответствия, найденные *Tile* для каждого узла. Верхние индексы над номерами правил показывают наименьшую стоимость каждой записи. Запись в строке, имеющая наименьшую стоимость, выделена **полужирным** шрифтом.

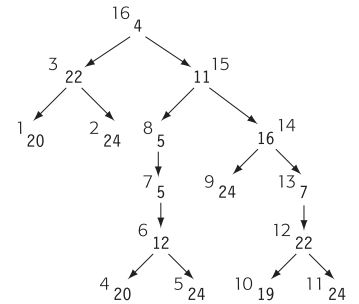
Чтобы лучше прочувствовать алгоритм, рассмотрим поддерево, которое начинается узлом 13 (см. рис. 11.8). При обходе в обратном порядке *Tile* будет вычислять множества *Match* для узлов 10 и 11. Они листовые, поэтому множества вычислены заранее. К узлу `LAB` применимо только правило 19, которое переписывает `LAB` как `Reg`. Запись в таблице отражает этот факт. Аналогично запись для узла 11 – это показанное ранее множество *Match*, которое означает, что правило 18 переписывает `NUM` как `Reg`, а правило 24 переписывает `Num` как `T3`.

Для узла 12 *Tile* находит четыре возможности. Она может породить `Reg` для сложения, пользуясь правилом 8 или 9. А может породить `T1` по правилу 21 или `T2` по правилу 22. Как показано на рис. 11.8, каждый из этих вариантов диктует вполне определенные соответствия для узлов 10 и 11.

Наконец, для узла 13 *Tile* находит три варианта: правила 5, 6 и 7. Каждое из них переписывает поддерево, вставляя `Reg`. Обход поддерева АСД сверху вниз с выбором совместимого набора правил из *Match* генерирует последовательности, показанные на рис. 11.8.



(a) Низкоуровневое АСД для a ← b - 2 × c



(b) Нисходящее назначение правил

Код	Rule	Emitted Code			
6	12	subI	r_{arp}, 16	⇒	r_a
7	5	load	r_a	⇒	r_b
8	5	load	r_b	⇒	r_c
10	19	loadI	@CP	⇒	r_d
		loadAI	r_d, @G	⇒	r_e
13	7	loadAI	r_e, 12	⇒	r_f
14	16	multI	r_f, 2	⇒	r_g
15	11	sub	r_c, r_g	⇒	r_h
16	4	storeAI	r_h	⇒	r_{arp}, 4

(c) Код, сгенерированный правилами

Код	Reg	Stmt	T1	T2	T3
1	**20**^{0}				
2	18^{1}				**24**^{0}
3	8^{2} 9^{1}		21^{1}	**22**^{0}	
4	**20**^{0}				
5	18^{1}				**24**^{0}
6	11^{2} **12**^{1}				
7	**5**^{4}				
8	**5**^{7}				
9	18^{1}				**24**^{0}
10	**19**^{4}				
11	18^{1}				**24**^{0}
12	8^{6} 9^{5}		21^{5}	**22**^{4}	
13	5^{8} 6^{8} **7**^{7}				
14	14^{10} **16**^{9}				
15	**11**^{17}				
16		2^{21} 3^{21} **4**^{20}			

(d) Полный набор соответствий и стоимостей

Рис. 11.10 Результаты применения *Tile* к низкоуровневому АСД для a ← b - 2 × c

Учет стоимостей

Имея набор соответствий и стоимостей для каждого правила, компилятор может вычислить вариант с наименьшей стоимостью в каждой категории в процессе простого обхода в обратном порядке. Он аккумулирует стоимости для каждой категории снизу вверх, для чего определяет стоимость выбора конкретного правила как сумму собственной стоимости этого правила

и стоимостей выбора в поддеревьях, которых требует правило. Сопоставитель может вычислить стоимость выбора каждого правила и сохранить самое дешевое в каждой категории.

Стоимость вычисления можно включить в *Tile*, как показано на рис. 11.11. Это несложное расширение кода на рис. 11.9.

```
Tile(n)  / * n – вершина АСД * /
  if n – листовый узел then
    Match(n,* ).rule ← { правило с наименьшей стоимостью,
                  сопоставляемое с n, в каждом классе}
    Match(n,* ).cost ← { соответствующая стоимость }

  else if n – унарный узел then
    Tile(child(n))
    Match(n ,*).rule ← invalid
    Match(n ,*).cost ← наибольшее целое число
    for каждого правила r, где operator(r) = operator(n) do
      if (child(r), child(n)) совместимы then
        NewCost ← RuleCost(r) + Match(child(n),class(child(r))).cost
        if (Match(n,class(r)).cost > NewCost) then
          Match(n,class(r)).rule ← r
          Match(n,class(r)).cost ← NewCost

  else if n – бинарный узел then
    Tile(left(n))
    Tile(right(n))
    Match(n ,*).rule ← invalid
    Match(n ,*).cost ← наибольшее целое число
    for каждого правила r, где operator(r) = operator(n) do
      if (left(r), left(n)) and (right(r), right(n)) совместимы then
        NewCost ← RuleCost(r) + Match(left(n), class(left(r))).cost
              + Match(right(n), class(right(r))).cost
        if (Match(n,class(r)).cost > NewCost) then
          Match(n,class(r)).rule ← r
          Match(n,class(r)).cost ← NewCost
```

Рис. 11.11 Вычисление дешевых соответствий для покрытия АСД

В таблице на рис. 11.10(d) показано, в каком месте процесса сопоставления производится выбор. Рассмотрим узел 3. Для порождения значения типа `Reg` подходят два образца. Полная стоимость правила 8 равна 2: его собственная стоимость 1 плюс 0 для `Reg` в узле 1 и 1 для `Reg` в узле 2. Полная стоимость правила 9 равна 1: его собственная стоимость 1 плюс 0 для `Reg` в узле 1 и 0 для `T3` в узле 2. Стоимостной алгоритм оставит сопоставление с правилом 9. В дополнение к этим соответствиям для `Reg` один образец сопоставляется с поддеревом для порождения каждого из `T1` и `T2`. Наименьшую стоимость среди этих соответствий для узла 3 имеет правило 22, которое порождает `T2`.

К моменту завершения работы сопоставитель аннотировал каждый узел правилом, порождающим соответствие с наименьшей стоимостью. Эти со-

ответствия показаны на рис. 11.10(b). Селектор команд может генерировать код в процессе обхода дерева снизу вверх в обратном порядке. На рис. 11.10(c) показан результирующий код с именами регистров, которые используются для связывания вместе различных шаблонов кода.

Этот процесс в каждой точке определяет выбор правила, порождающего последовательность кода с минимальной стоимостью в некоторой локальной окрестности этой точки в дереве. В литературе это свойство называется *локальной оптимальностью*.

Локальная оптимальность
Схема, при которой у компилятора нет лучшего варианта в каждой точке кода, называется *локально оптимальной*.

Локальная оптимальность не гарантирует, что решение оптимально на более высоком уровне, таком как целая процедура или вся программа. Вообще говоря, компиляторы не могут эффективно рассмотреть все детали, необходимые для достижения глобальной оптимальности.

Обратите внимание, как различается конечный код для ссылок на все три переменные. Сохранение в `a` со смещением 4 от УЗА целиком свернуто в одну операцию `storeAI`. Загрузка из `b`, переданного по ссылке параметра, указатель на который расположен на 16 байт выше УЗА, требует `subI` и двух операций `load`; режим адресации со смещением поддерживает только положительные смещения, и передача по ссылке приводит к дополнительной загрузке. Наконец, загрузка из глобальной переменной `c`, расположенной со смещением 12 от метки, требует операции `loadI` для нахождения пула констант и еще двух операций `loadAI`. Первая `loadAI` выбирает адрес метки, хранящийся в `@CP+@G`, а вторая – значение со смещением 12 от этого адреса. На уровне исходного кода все три ссылки текстуально похожи, но о сгенерированном для них коде этого не скажешь.

А можно ли применить сопоставление с деревом-образцом к четверкам?

Используемая терминология – *сопоставление с деревом-образцом* и *оконная оптимизация* – содержит неявные предположения о видах ПП, к которым эти методы применимы. Описание селекторов на основе сопоставления с деревом-образцом подразумевает, что селектор имеет дело с древовидным ПП. Аналогично оконные оптимизаторы впервые были предложены как последний проход для улучшения уже сгенерированного ассемблерного кода. Идея скользящего окна команд подразумевает линейное низкоуровневое ПП.

Оба метода можно адаптировать к большинству ПП. Компилятор может интерпретировать низкоуровневое линейное ПП типа ILOC как дерево. Каждая операция становится узлом дерева; ребра определяются повторным использованием операндов. Аналогично если компилятор назначит имя каждому узлу, то сможет интерпретировать дерево как линейную форму, полученную в результате его обхода в обратном порядке. Умелый разработчик сумеет адаптировать описанные в этой главе методы к широкому спектру реальных ПП.

И напоследок отметим, что селектор на основе сопоставления с деревом-образцом сворачивает константу 2 в операцию умножения. Напомним, что оконный селектор не смог выполнить такое переписывание, потому что присваивание значения 2 регистру r_1 и использование r_1 в операции умножения

отстояли слишком далеко друг от друга, что не позволило упростителю разглядеть такую возможность. А сопоставитель с деревом-образцом кодирует тот факт, что поддерево имеет постоянное значение, и переносит его в составе множества возможных соответствий до точки, где принимается решение о выборе между `mult` и `multI`.

11.4.4. Инструменты

Как мы видели, подход, подразумевающий обход дерева снизу вверх, может порождать эффективные селекторы команд. У автора компилятора есть несколько способов реализовать кодогенератор на основе следующих принципов.

1. Можно вручную написать сопоставитель, похожий на *Tile*, который явно проверяет подходящие правила в процессе покрытия дерева. Продуманная реализация может ограничить множество правил, просматриваемых в каждом узле. Это позволит избежать больших таблиц и получить компактный кодогенератор.
2. Поскольку задача конечна, автор компилятора может закодировать ее как конечный автомат сопоставления деревьев, похожий на ДКА. В этой схеме имеется справочная таблица, кодирующая функцию переходов автомата и неявно включающая всю необходимую информацию о состоянии. Такой подход, называемый системой восходящего переписывания (bottom-up rewrite system – BURS), используется в нескольких реальных системах.
3. Форма правил, похожая на грамматику, наводит на мысль об использовании методов разбора, обобщенных для обработки сильно неоднозначных грамматик, получающихся из описаний машины. Целью в этом случае является выбор способа разбора с наименьшей стоимостью.
4. Если компилятор линеаризует дерево в префиксную форму, то может воспользоваться алгоритмом сравнения строк для поиска потенциальных соответствий.

В практических инструментах реализованы последние три подхода. Автор компилятора создает описание системы команд целевой машины, а генератор порождает исполняемый код по этому описанию.

Детали автоматизированных инструментов различаются. Различна стоимость в расчете на одну сгенерированную команду. Одни генераторы быстрее, другие медленнее, но ни один не является настолько медленным, чтобы это заметно отразилось на скорости работы компилятора в целом. Разные методы по-разному подходят к определению модели стоимости. Некоторые системы требуют, чтобы стоимость каждого правила была фиксированной, но взаимен могут выполнять некоторые, а то и все виды анализа в процессе генерирования таблицы. Другие допускают более общие модели, в которых стоимость изменяется в процессе сопоставления, и должны манипулировать стоимостями во время выбора команд. Но, вообще говоря, все подходы порождают вполне эффективные селекторы команд.

ИТОГИ РАЗДЕЛА

При выборе команд с помощью сопоставления с деревом-образцом деревья используются для представления операций программы и операций целевой машины. Автор компилятора создает библиотеку деревьев-образцов, которые отображают конструкции ПП на операции целевой машины. Каждый образец состоит из небольшого дерева ПП, шаблона кода и стоимости. На одном проходе селектор находит локально оптимальное покрытие дерева. Далее в процессе обхода дерева в обратном порядке генерируется код по шаблонам, ассоциированным с плитками покрытия.

Для реализации проходов построения покрытия применялось несколько технологий: написанные вручную сопоставители, сопоставители на основе конечных автоматов, сопоставители на основе парсеров, работающих с неоднозначными грамматиками, и линейные сопоставители на базе алгоритмов быстрого сравнения строк. Все эти технологии продемонстрировали успешную работу в одной или нескольких системах. Результирующие селекторы команд работают быстро и порождают высококачественный код.

ВОПРОСЫ

1. Сопоставление с деревом-образцом кажется естественным для использования в компиляторе с древовидным ПП. Как разделение информации, т. е. использование ориентированного ациклического графа (ОАГ) вместо дерева, могло бы повлиять на алгоритм?
2. В наших примерах каждый образец имел фиксированную стоимость. Как компилятор мог бы воспользоваться способностью моделировать стоимости динамически?

11.5. Дополнительные темы

И оконные, и основанные на сопоставлении с деревом-образцом селекторы команд проектировались для достижения эффективности на этапе компиляции. Однако оба метода ограничены набором образцов, составленным автором компилятора. Чтобы отобрать лучшие образцы, автор компилятора должен рассмотреть использование различных методов поиска образцов.

Идея проста. Комбинации команд иногда дают удивительные эффекты. Поскольку результаты неожиданны, автор компилятора редко может предвидеть их и потому не включает в спецификацию целевой машины. Автоматическое обнаружение может выявить новые образцы.

В литературе описаны два разных подхода, в которых для обнаружения новых образцов используется исчерпывающий поиск. В первом оконная система обнаруживает новые образцы в процессе компиляции кода. Во втором применяется полный перебор в пространстве возможных команд.

11.5.1. Обучение оконным образцам

Для ускорения оконной оптимизации несколько систем заменили упростительдвижком переписывания, управляемым образцами. Движок по-прежнему работает с небольшим скользящим окном. Он использует библиотеку образцов, чтобы распознать возможности внутри окна, а не исследует все

возможные подстановки в каждом последующем окне. При таком подходе уменьшается время, затраченное на сокращение, но появляется компромисс между размером набора образцов и качеством результирующего кода.

Компромисс обменивает время автора компилятора, потраченное на разработку набора образцов, на качество кода, генерируемого селектором команд. Чем больше набор, тем больше возможных упрощений, но за это приходится расплачиваться временем составления набора. При небольшом наборе образцов возможно меньше упрощений, зато на его разработку уходит меньше времени.

Дэвидсон и Фрезер создали систему, в которой быстрый управляемый образцами упроститель сочетается с исчерпывающим упростителем [126]. Автор компилятора должен предоставить начальный набор образцов, чтобы перенести селектор команд на новую машину.

Эффективный способ сгенерировать явную таблицу образцов, необходимую быстрому оконному селектору, – объединить ее с оптимизатором, в котором применяется символический упроститель. Всякий раз, как упроститель находит улучшение, он запоминает начальную и упрощенную последовательности. На последующем проходе обнаруженные образцы проверяются на предмет применимости в общем случае и запоминаются для дальнейшего использования.

Применив символический упроститель к обучающему набору приложений, компилятор может найти большинство необходимых ему образцов. Затем компилятор может использовать эту таблицу как основу для быстрого селектора на базе сопоставления с образцами. Это позволит автору компилятора ускорить рутинное использование компилятора ценой увеличения компьютерного времени, потраченного на этапе проектирования. Сложность образцов, которые необходимо задать вручную, при этом значительно сокращается.

Расширение взаимодействия между двумя упростителями может еще повысить качество кода. На этапе компиляции быстрый сопоставитель с образцами встречает в НУПП пары, которым нет никакого соответствия в таблице. В таком случае он может вызвать символический упроститель для поиска улучшения, применив всю мощь поиска только к парам, для которых еще нет образца.

Чтобы этот подход имел практическую ценность, символический упроститель должен запоминать как успехи, так и неудачи. Эта стратегия позволит отвергать ранее встречавшиеся в НУПП пары без накладных расходов на символическую интерпретацию. В случае успешного улучшения пары упроститель должен добавить новый образец в таблицу селектора, чтобы при обработке ее будущих вхождений использовался более эффективный механизм.

У этого основанного на обучении подхода к генерированию образцов есть несколько преимуществ. Он тратит усилия только на ранее не встречавшиеся пары в НУПП. Он компенсирует прорехи в покрытии целевой машины обучающим набором. Он предоставляет скрупулезность более дорогой си-

стемы, почти не жертвуя скоростью системы, основанной на сопоставлении с образцами.

Однако, используя этот подход, автор компилятора должен решить, когда символическому оптимизатору следует обновлять таблицы образцов и как с этими обновлениями поступать. Разрешать произвольному акту компиляции переписывать таблицу образцов для всех пользователей вряд ли имеет смысл, поскольку неизбежно возникнут проблемы синхронизации и безопасности. Вместо этого автор компилятора мог бы выбрать периодические обновления – сохранять вновь найденные образцы и распространять их в составе очередного обновления компилятора.

11.5.2. Генерирование последовательностей команд

Подходу на основе обучения свойственна предвзятость: он предполагает, что низкоуровневые образцы должны направлять поиск эквивалентной последовательности команд. Некоторые компиляторы выбрали исчерпывающий подход к решению той же базовой проблемы. Вместо того чтобы пытаться синтезировать желаемую последовательность команд по низкоуровневой модели, они решили применить генерирование с последующим тестированием.

Идея проста. Компилятор или его автор идентифицирует короткую последовательность ассемблерных команд, которую следует улучшить. Затем компилятор генерирует все ассемблерные последовательности единичной стоимости и подставляет оригинальные аргументы в сгенерированную последовательность. Он тестирует каждую получившуюся последовательность, проверяя, дает ли она тот же эффект, что целевая. Исчерпав все последовательности заданной стоимости, компилятор увеличивает стоимость на единицу и продолжает в том же духе. Процесс продолжается до тех пор, пока (1) не будет найдена эквивалентная последовательность, (2) не будет достигнута стоимость оригинальной целевой последовательности или (3) не будет превышено внешнее ограничение на стоимость или время компиляции.

Это принципиально дорогой подход, но механизм, используемый для проверки эквивалентности, сильно влияет на время, необходимое для тестирования каждой последовательности-кандидата. Очевидно, необходим формальный подход с использованием низкоуровневой модели машинных эффектов, чтобы отфильтровать тонкие несоответствия, однако для отлавливания серьезных несоответствий, которые чаще всего и встречаются, можно воспользоваться более быстрой проверкой. Если компилятор просто генерирует и выполняет последовательность-кандидат, то может сравнить результаты с полученными на целевой последовательности. Этот простой подход, примененный к тщательно отобранным входам, должен исключить большинство непригодных последовательностей-кандидатов с помощью дешевой проверки.

Понятно, что этот подход слишком дорог для рутинного использования или для применения к большим фрагментам кода. Но при некоторых усло-

виях он заслуживает внимания. Если автор компилятора или сам компилятор может идентифицировать небольшой, критически важный для производительности участок кода, то выигрыш от использования очень хорошей последовательности команд может оправдать применение исчерпывающего поиска. Например, в некоторых встраиваемых приложениях код, требующий максимальной производительности, состоит из одного внутреннего цикла. Использование исчерпывающего поиска для небольших фрагментов кода – чтобы улучшить быстродействие или потребление памяти – может окупиться.

Исчерпывающий поиск также применялся как часть процесса переноса компилятора на новую архитектуру. В этом случае его цель – найти особенно эффективные реализации для последовательностей ПП, рутинно генерируемых компилятором. Поскольку затраты возникают на стадии переноса компилятора, использование поиска можно оправдать тем, что они распределятся по многим актам компиляции, ожидаемым после внедрения нового компилятора.

11.6. Итоги и перспективы

По сути своей выбор команд – это задача сопоставления с образцом. Ее трудность зависит от уровня абстракции созданного компилятором ПП, от сложности целевой машины и от ожидаемого качества сгенерированного кода. В некоторых случаях простой обход дерева дает приемлемые результаты. Но при более серьезных требованиях лучше использовать систематический поиск, направляемый либо сопоставлением с деревьями-образцами, либо оконной оптимизацией. Ручное написание селектора команд на основе обхода дерева, который смог бы достичь таких же результатов, потребовало бы гораздо больше работы. И хотя эти два подхода различаются почти во всех деталях, идея у них общая – использовать сопоставление с образцом для нахождения хорошей последовательности кода среди мириадов возможных для любого данного ПП программы.

Оконные селекторы систематически упрощают ПП программы и сопоставляют результат с набором образцов для целевой машины. Поскольку для них не существует явной модели стоимости, доказать их оптимальность невозможно. Они генерируют код, порождающий такие же эффекты, как ПП программы, а не буквальную реализацию этого ПП. С другой стороны, сопоставители с деревом-образцом находят покрытия с низкой стоимостью, делая в каждой точке принятия решений наиболее дешевый выбор. Результирующий код реализует вычисление, описываемое ПП программы. Из-за этого тонкого различия двух подходов мы не можем напрямую сравнивать их заявления о качестве. На практике с помощью каждого подхода были получены отличные результаты.

Практические достоинства этих методов были продемонстрированы в реальных компиляторах. И GCC, и LCC работают на многих платформах.

В первом используется оконный селектор, во втором – сопоставление с деревьями-образцами. Применение автоматизированных инструментов в обеих системах способствовало легкости освоения, простоте переноса и, в конечном счете, широкому признанию сообществом.

Читателю также важно понимать, что оба семейства автоматических сопоставителей с образцами можно применять и к другим задачам компиляции. Оконная оптимизация изначально была предложена как метод улучшения конечного кода, порожденного компилятором. Аналогично компилятор может применить сопоставление с деревом-образцом для распознавания вычислений в АСД и их переписывания. Технология BURS может оказаться особенно эффективным способом распознавания и улучшения простых паттернов, в т. ч. алгебраических тождеств, распознаваемых с помощью нумерации значений.

Замечания к этой главе

В большинстве ранних компиляторов применялись рукописные ситуативные методы выбора команд [27]. Когда система команд достаточно мала, а коллектив разработчиков достаточно велик, такой подход работает. Например, компилятор BLISS-11 генерировал отличный код для PDP/11 с ее ограниченным набором операций [368]. Благодаря небольшим системам команд ранних компьютеров и мини-компьютеров исследователи и авторы компиляторов могли игнорировать некоторые проблемы, возникающие на современных машинах.

Например, Сети и Ульман [321], а позднее Ахо и Джонсон [6] рассматривали задачу генерирования оптимального кода для деревьев выражений. В работе Ахо, Джонсона и Ульмана [7] их идеи обобщены на ОАГ выражений. В компиляторах, основанных на этой работе, для управляющих конструкций использовались ситуативные методы. Эртл показал, как использовать разбор дерева с целью нахождения оптимальных последовательностей кода для ОАГ [153].

В конце 1970-х годов две разные тенденции в компьютерной архитектуре вывели проблему выбора команд на передний край исследований в области компиляторов. Переход от 16- к 32-разрядным архитектурам знаменовал комбинаторный взрыв числа операций и режимов адресации, которые вынужден был рассматривать компилятор. Чтобы компилятор мог исследовать сколько-нибудь большую долю возможностей, был необходим более формальный и мощный подход. Как раз в то время только-только народившаяся операционная система Unix начала появляться на нескольких платформах. Естественно, возник спрос на компиляторы C и интерес к переносимым компиляторам [217]. Способность легко перенести селектор команд играет ключевую роль для простоты переноса компилятора на новую архитектуру. Эти две тенденции привели к всплеску исследований по выбору команд, которые начались в 1970-е годы и продолжались до середины 1990-х [78, 79, 143, 170, 176, 297, 298].

Благодаря успехам автоматизации сканирования и синтаксического разбора стала привлекательной идея положить в основу выбора команд спецификацию. Глэнвилл и Грэхэм преобразовали выбор команд на основе сопоставления с образцом в таблично управляемый разбор [170, 175, 177]. Ганапати и Фишер применили для решения проблемы атрибутные грамматики [166].

Первым оконным оптимизатором, похоже, является система Маккимана [268]. Бэгуэлл [31], Вульф и др. [368] и Лэм [246] описывают ранние оконные системы со скользящими окнами. Цикл расширения, упрощения и сопоставления, описанный в разделе 11.3.1, заимствован из работ Дэвидсона [125, 128]. Кэсслер также работал над выводом оконных оптимизаторов непосредственно из низкоуровневых описаний целевых архитектур [233]. Фрэзер и Вендт адаптировали оконную оптимизацию под задачу выбора команд [164, 165]. Подход на базе машинного обучения, описанный в разделе 11.5.1, был предложен в работе Дэвидсона и Фрэзера [126].

Кодогенераторы на основе сопоставления с деревом-образцом выросли из ранних работ по таблично управляемому генерированию кода [10, 46, 177, 195, 249] [83, 204]. Пелегри-Ллопарт формализовал многие из этих понятий в теории BURS [290]. Последующие авторы на основе этой работы создали разнообразные реализации, вариации и алгоритмы генерирования таблиц [162, 163, 297]. В системе Twig сопоставление с деревьями-образцами сочетается с динамическим программированием [3, 346].

Массалин предложил исчерпывающий подход, описанный в разделе 11.5.2 [265]. В несколько ограниченном виде он был применен в компиляторе GCC Гранлундом и Кеннером [180].

Упражнения

К разделу 11.3

1. Оконный селектор должен иметь дело с такими операциями управления потоком, как условные и безусловные переходы и помеченные предложения.
   a. Что должен делать оконный селектор, когда в окно оптимизации подтягивается условный переход?
   b. Изменится ли ситуация, если встретился безусловный переход?
   c. Что делать с помеченной операцией?
   d. Что может сделать оптимизатор для улучшения ситуации?

2. Оконные селекторы упрощают код при выборе его конкретной реализации. Предположим, что оконный селектор работает до планирования команд и распределения регистров и что селектор может использовать неограниченное множество имен виртуальных регистров.
   a. Может ли селектор изменить спрос на регистры?
   b. Может ли селектор изменить набор доступных планировщику возможностей для переупорядочения кода?

К разделу 11.4

3. Постройте низкоуровневое АСД для каждого из следующих выражений, взяв за образец дерево на рис. 11.1:

a. `y ← a × b + c × d`

b. `w ← a × b × c - 7`

Можете предполагать, что `a` расположено со смещением 8 от УЗА; что `b` и `c` находятся в регистрах (r_b и r_c); что расположено со смещением 12 от начала пула константа (по адресу `@CP`); что `w` расположено со смещением 12 от УЗА и что `y` находится в регистре (r_y).

Для покрытия этих деревьев используйте правило на рис. 11.5.

4. В любой схеме выбора команд с обходом дерева компилятор должен выбрать порядок вычисления поддеревьев. То есть решить, какое поддерево бинарного узла *n* – левое или правое – вычислять первым.

a. Влияет ли выбор порядка на число регистров, необходимое для вычисления всего поддерева?

b. Как можно включить этот выбор в восходящие схемы сопоставления с деревом-образцом?

5. Алгоритм на рис. 11.9 ограничивает правила однооператорными.

a. Модифицируйте этот алгоритм, так чтобы он правильно обрабатывал правила с двумя операторами, например `Reg → ◆(+ (Reg1, NUM2))`.

b. Как ваши изменения отразятся на времени работы алгоритма?

Глава 12

Планирование команд

АННОТАЦИЯ

Время, затраченное на выполнение набора операций, сильно зависит от порядка, в котором эти операции предъявлены для выполнения. Планировщик команд изменяет порядок операций в процедуре, чтобы уменьшить время их выполнения. Множество допустимых порядков ограничено необходимостью сохранить поток значений в оригинальном коде. Цель состоит в том, чтобы уменьшить количество тактов от начала выполнения кода до конца.

В этой главе приведено введение в самый распространенный метод планирования, применяемый в компиляторах: жадное планирование списка команд в одиночных простых блоках. Авторы компиляторов применяют различные вариации этой схемы для планирования списка команд в более широких областях. Будут представлены некоторые из них: суперлокальное планирование, планирование трасс и планирование циклов.

КЛЮЧЕВЫЕ СЛОВА

Планирование команд, граф зависимостей, планирование списка, планирование трасс, программный конвейер

12.1. Введение

На многих процессорах порядок предъявления операций для выполнения сильно влияет на время выполнения всей последовательности. Разные операции занимают разное число процессорных тактов. На типичном бытовом микропроцессоре сложение и вычитание целых чисел занимают меньше тактов, чем целочисленное деление; аналогично деление чисел с плавающей точкой занимает больше тактов, чем их сложение или вычитание. Умножение обычно оказывается посередине между сложением и делением. Стоимость загрузки из памяти зависит от того, в каком месте иерархии памяти находится загружаемое значение.

Чтобы помочь компилятору в управлении этой сложной ситуацией, большинство процессоров допускают совмещенное выполнение операций, прав-

да, с некоторыми ограничениями, гарантирующими предсказуемость результатов. Чтобы операции могли выполняться конкурентно, большинство процессоров предлагают несколько независимых функциональных блоков. Дабы уменьшить интервал между операциями, в большинстве процессоров применяется конвейерная модель выполнения. Оба механизма поддержки конкурентности увеличивают сложность модели выполнения.

Планирование команд – процесс систематического переупорядочения операций в блоке или процедуре, так чтобы их выполнение занимало меньше тактов. Планировщик принимает частично упорядоченный список операций на языке ассемблера целевой машины и порождает вполне упорядоченный список тех же операций. Он предполагает, что код уже оптимизирован, и не пытается продублировать работу оптимизатора. Вместо этого планировщик распределяет операции по имеющимся тактам и слотам запуска в функциональных блоках, чтобы уменьшить число холостых тактов.

Запуск
Процессор *запускает* операцию, когда начинает процесс ее декодирования и выполнения.

Обычно у процессора имеется фиксированное число слотов запуска в одном такте, как правило, по одному на каждый функциональный блок.

Концептуальная дорожная карта

Порядок, в котором процессор запускает операции, напрямую влияет на скорость выполнения кода. Поэтому большинство компиляторов включают планировщик команд, который изменяет порядок сгенерированных операций с целью повысить производительность. Доступные планировщику варианты ограничены потоком данных, задержками индивидуальных операций и возможностями целевого процессора. Планировщик должен учитывать все эти факторы, если хочет построить корректный и эффективный план выполнения откомпилированного кода. Почти в любом реалистичном сценарии планирование для одиночных простых блоков является NP-полной задачей.

Преобладающий метод планирования команд – жадный эвристический алгоритм, называемый планированием списка. Такие планировщики работают с линейным кодом и используют различные схемы ранжирования по приоритетам. Авторы компиляторов придумали целый ряд схем для планирования в более крупных регионах, чем простые блоки; такие планировщики регионов и циклов просто создают условия, при которых компилятор может применить планирование списка к более длинной последовательности операций.

Несколько слов о времени

Планирование команд – это действие на этапе компиляции, цель которого – улучшить качество порождаемого компилятором кода. Как и большинство оптимизаций, планировщик работает на этапе компиляции. Он анализирует

код и изменяет порядок операций, прежде чем они будут сгенерированы компилятором. Плюсы такого переупорядочения проявляются во время выполнения.

Краткий обзор

Чтобы ускорить выполнение программы, в архитектуру процессора включены такие дополнительные возможности, как конвейерное выполнение, несколько функциональных блоков и выполнение не по порядку. Из-за них процессоры и их модели выполнения становятся сложнее, но взамен становится возможно повысить производительность; правда, реализованная производительность оказывается более чувствительна к порядку запуска операций. Таким образом, планировщик команд является важным фактором, влияющим на производительность конечного откомпилированного кода.

Неформально говоря, планирование команд – это процесс, посредством которого компилятор изменяет порядок операций в конечном коде, чтобы уменьшить число занятых ими тактов. Планировщик принимает частично упорядоченный список команд и порождает упорядоченный список команд, построенный из того же набора операций. Планировщик предполагает, что набор операций фиксирован, он не занимается оптимизацией кода. Также предполагается, что значения, находившиеся в регистрах (или в оперативной памяти), там и останутся; планировщик не меняет ранее принятых решений о выделении памяти и о размещении в памяти.

Планировщик пытается минимизировать время работы блока. Но на практике он не гарантирует, что выбранный план оптимален.

Основная цель планировщика – вычислить порядок запуска операций в блоке такой, что (1) операнды доступны, когда в них возникает необходимость; (2) функциональные блоки в достаточной мере загружены работой; (3) количество тактов, необходимых для выполнения блока, сведено к минимуму.

Пример

Длина плана
Длиной плана для фрагмента кода называется количество тактов, необходимых для его полного выполнения.

Рассмотрим пример на рис. 12.1(а), где вычисляется выражение `(a + a) × b × c × d`; в первый раз этот пример был приведен в разделе 1.3.3. Мы подставили конкретные числа вместо смещений `@a`, `@b`, `@c` и `@d`. В столбце «Начало» показано, в каком такте запускается каждая операция. Предполагается, что процессор оснащен одним функциональным блоком, что операции загрузки и сохранения занимают три такта, операция умножения – два такта, а все остальные операции – один такт. Оригинальный код, показанный на рисунке (а), выполняется за 22 такта; эта величина называется *длиной плана*.

Начало	Операции			
1	loadAI	r_{arp},4	⇒	r_1
4	add	r_1,r_1	⇒	r_1
5	loadAI	r_{arp},8	⇒	r_2
8	mult	r_1,r_2	⇒	r_1
10	loadAI	r_{arp},12	⇒	r_2
13	mult	r_1,r_2	⇒	r_1
15	loadAI	r_{arp},16	⇒	r_2
18	mult	r_1,r_2	⇒	r_1
20	storeAI	r_1	⇒	r_{arp}, 4

(a) Оригинальный код

Начало	Операции			
1	loadAI	r_{arp},4	⇒	r_1
2	loadAI	r_{arp},8	⇒	r_2
3	loadAI	r_{arp},12	⇒	r3
4	add	r_1,r_1	⇒	r_1
5	mult	r_1,r_2	⇒	r_1
6	loadAI	r_{arp},16	⇒	r_2
7	mult	r_1,r_3	⇒	r_1
9	mult	r_1,r_2	⇒	r_1
11	storeAI	r_1	⇒	r_{arp},4

(b) Спланированный код

Рис. 12.1 Пример блока из главы 1

Спланированный код, показанный на рисунке (b), выполняется за 13 тактов. Он отделяет операции с длительной задержкой от операций, которые ссылаются на их результаты. Это разделение позволяет выполнять операции, не зависящие от результатов долгих операций, параллельно – еще до того, как эти долгие операции завершатся. Спланированный код запускает операции загрузки в тактах 1, 2 и 3; результаты становятся доступны в тактах 4, 5 и 6 соответственно. Совмещение операций маскирует задержку операций памяти. Тот же подход маскирует задержку первого умножения. Длина плана для переупорядоченного кода составляет 13 тактов, т. е. мы улучшили производительность на 41 %, не изменив множество операций. Платой за конкурентность стал один дополнительный регистр.

Дорожная карта

В этой главе рассматривается сама задача планирования команд и алгоритмы ее решения, применяемые в компиляторах. В разделе 12.2 обсуждаются средства процессора, затрудняющие планирование, и идеи, лежащие в основе алгоритмов планирования. В разделе 12.3 мы познакомимся с планированием списка – стандартной схемой, используемой в большинстве компиляторов. В разделе 12.4 будут рассмотрены методы, позволяющие применить эту схему к более крупным областям кода. В разделе «Дополнительные темы» рассматривается программный конвейер – метод планирования циклов.

12.2. Предварительные сведения

Чтобы компенсировать сложность моделей выполнения современных процессоров и обратить ее во благо, компиляторы систематически изменяют порядок операций в сгенерированном ими коде. В этом процессе перед компилятором стоит несколько целей. Он

Длина плана (*еще раз*)
Длиной плана блока называется количество тактов с момента запуска первой операции до момента завершения последней.

стремится уменьшить количество пустых, или холостых, слотов запуска операций, сократить время простоя функциональных блоков и снизить общую длину плана.

Граф зависимостей
В графе зависимостей $\mathcal{D}$ для блока b имеется по одной вершине для каждой операции в b и ребро (x, y), если операция y пользуется результатом x, т. е. если y зависит от x.

Планировщик команд анализирует код, чтобы определить, как поток значений ограничивает порядок выполнения. Он строит *граф зависимостей* $\mathcal{D}$, в котором представлены ограничения на порядок, индуцированные потоком значений. Планировщик изменяет порядок операций в рамках ограничений, закодированных в $\mathcal{D}$, чтобы воспользоваться ресурсами процессора и системы и по возможности замаскировать задержки последних.

Чтобы спроектировать и реализовать планировщик, автор компилятора должен понимать весь комплекс средств процессора, как они влияют на производительность откомпилированного кода и какие проблемы или, наоборот, возможности предлагают планировщику. В этом разделе описываются некоторые средства процессора, осложняющие планирование. Затем ставится задача планирования.

12.2.1. Архитектурные средства, влияющие на производительность

В этом разделе предлагается обзор средств, влияющих на планирование. Читателю, интересующемуся дополнительными деталями, рекомендуем обратиться к учебнику по компьютерной архитектуре [200].

Компьютерные архитекторы изучили и реализовали мириады средств, ставящих целью повышение пиковой производительности процессора. Некоторые из них, например конвейерное выполнение, одновременно влияют на планирование и зависят от него. Другие, например регистровые окна, практически не оказывают влияния на проектирование и эффективность планировщика. В этом разделе описываются три средства, которые взаимодействуют с планированием: конвейерное выполнение, операции с переменной задержкой и множественные функциональные блоки.

Если явно не оговорено противное, то предполагается, что все операции *неблокирующие* – процессор может запускать операции на функциональном блоке в каждом такте, даже если запущенная ранее операция еще не завершилась. Операции начинают выполнение в момент запуска, если только процессор не обнаруживает, что в операции используется значение, которое будет определено еще выполняющейся операцией. Обнаружив такой *конвейерный конфликт по данным*, процессор приостанавливает операцию до тех пор, пока операнд не станет доступен.

Конвейерное выполнение

Фундаментальная единица времени планирования – один такт тактового генератора процессора. Многие операции, например целочисленное сложение, непосредственная загрузка или логический сдвиг, занимают один такт. Более сложные операции, например умножение, деление, условный и безусловный переход и большинство операций с памятью, требуют больше тактов. Типичный процессор может запускать по одной операции в каждом такте каждого функционального блока, поэтому чем короче такт, тем больше операций может быть запущено в секунду.

Многотактовые операции часто *конвейеризуются*, т. е. операция разлагается на несколько более простых однотактовых шагов, которые, будучи выполнены последовательно, реализуют сложную операцию. Конвейер похож на сборочную линию: каждая ступень конвейера выполняет часть операции, а в конце такта результат передается следующей ступени.

Типичная операция читает свои операнды в начале такта, в котором запущена, и записывает результаты в конце такта, в котором завершилась.

Чтобы воспользоваться конвейерным выполнением, планировщик должен понимать, на какой ступени конвейера операция читает необходимые ей значения и на какой записывает вычисленные значения. В идеале спланированный код запускает каждую операцию после того, как станут доступны ее операнды. Чтобы планировщик мог заполнить все слоты запуска, пока откладывает операцию в ожидании готовности аргументов, код должен обладать достаточной степенью параллелизма на уровне команд (ILP).

Оптимизатор может повысить степень видимого параллелизма на уровне команд. См., например, раздел 8.4.2.

Операции с переменной задержкой

Для некоторых операций задержка непредсказуема. Например, задержка операции загрузки или сохранения зависит от состояния иерархии памяти. Для некоторых реализаций умножения и деления задержка может немного варьироваться в зависимости от битового состава аргументов. Если у операции имеется максимальная задержка, то планировщик может использовать ее и таким образом создать безопасный план.

Для операций с памятью максимальная задержка зачастую слишком велика для использования при планировании. Рассмотрим цикл из восьми операций, содержащий операцию загрузки. Если загрузка может занять от 2 (попадание в кеш первого уровня) до 100 (загрузка из оперативной памяти) тактов, то приостановка на 100 тактов приведет к большому числу холостых тактов, если операции загрузки характеризуются хотя бы скромной локальностью. Поэтому планировщик должен оценить разумную ожидаемую задержку и положиться на то, что оборудование обнаружит незавершенные операции и приостановит выполнение следующих за ними.

Что будет, если операнд не готов?

В некоторых ситуациях процессор может попытаться выполнить операцию раньше, чем готовы все ее операнды. Следующая последовательность демонстрирует проблему. Предположим, что процессор запустил команды `mult` и `add` в соседних тактах.

```
mult r1,r2 ⇒ r3
add  r1,r3 ⇒ r4
```

Если `add` занимает один такт, а `mult` – три такта, то значение r_3 не будет готово к моменту запуска `add`. Процессор может обнаружить такой *конфликт по данным*, прежде чем запустит операцию, сравнив имена аргументов с именами, которые будут определены операциями, которые еще находятся в процессе выполнения.

Большинство процессоров приостановят новую операцию, пока аргумент не станет доступен. Однако процессоры различаются по последствиям такой приостановки.

- *Статически планируемые, или поочередные, процессоры.* Обнаружив необходимость приостановки, такие процессоры прекращают запускать дополнительные операции, пока конфликт не исчезнет. Некоторые процессоры останавливают только один функциональный блок, другие – сразу все.
- *Динамически планируемые, или внеочередные, процессоры.* Такие процессоры организуют окно в потоке команд и запускают операции, когда их аргументы оказываются готовы, отдавая предпочтение более ранним операциям в окне.

Планировщик должен размещать операции в тактах, где они могут быть выполнены без приостановки. В примере выше он мог бы запланировать две независимые операции между `mult` и `add`, если они доступны.

Некоторые процессоры полагались на то, что компилятор обнаружит и устранит конфликты по данным. Хотя такой подход может упростить оборудование, он усложняет планировщик. В такой схеме компилятору, возможно, придется вставлять команды `nop`, чтобы обеспечить разнесение операций по времени.

Множественные функциональные блоки

Чтобы увеличить число операций, выполняемых в секунду, архитекторы конструируют процессоры с несколькими функциональными блоками. Каждый блок может инициировать выполнение операции в каждом такте. Чтобы обеспечить занятость функциональных блоков, компилятор должен запускать несколько независимых операций в одном такте, используя параллелизм на уровне команд. Планировщик должен понимать, какими ресурсами он располагает – количество функциональных блоков и набор выполняемых ими операций, – и дисциплину диспетчеризации, определяющую, как процессор запускает операции.

Суперскалярный
так называется процессор, имеющий один поток команд и несколько функциональных блоков, который запускает несколько операций в одном такте.

У суперскалярных процессоров имеется единственный поток команд. В каждом такте процессор смотрит на поток команд и запускает столько последовательных команд, сколько может. Рассмотрим простой поочередный процессор с двумя функциональными блоками. В каждом такте он выбирает первую операцию в окне

и запускает ее. Затем он исследует следующую операцию; если эту операцию можно запустить на неиспользуемом функциональном блоке, то процессор ее запускает. Если нет, то блок в текущем такте простаивает. У внеочередного процессора окно больше, и он исследует последовательные операции, пока не найдет такую, которую можно запустить без конфликта.

С другой стороны, процессор *со сверхдлинным командным словом* (very-long instruction-word – VLIW) управляет процессом запуска операций, кодируя одну операцию на каждом функциональном блоке в каждой команде. Эта стратегия упрощает процесс выбора и запуска операций. Но теперь компилятор должен полностью описать проведение программы в плане запуска операций. VLIW-команда может представлять холостые операции (`nop`) явно или неявно.

VLIW-процессор
У процессора со *сверхдлинным командным словом* формат команды предусматривает слот для каждого функционального блока.

У процессора с *упакованным* VLIW командное слово имеет переменную длину и может содержать до одной операции на каждый функциональный блок.

Между этими крайностями имеется много промежуточных вариантов. Суперскалярные процессоры могут быть как строго поочередными, так и внеочередными с большим окном предвыборки. У некоторых VLIW-процессоров командные слова длинные, но они допускают изменение длины, чтобы сэкономить память, занятую кодом. Чтобы спланировать поведение во время выполнения, планировщик должен понимать дисциплину диспетчеризации команд процессором.

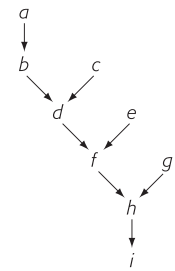
Разнообразие механизмов диспетчеризации размывает различие между операцией и командой. На машинах с VLIW и упакованным VLIW команда содержит несколько операций. На суперскалярных машинах мы обычно рассматриваем одну операцию как команду и говорим, что такие машины запускают несколько команд в одном такте. В этой книге мы используем термин *операция* для описания одного кода операции и соответствующих операндов, а термин *команда* – для описания агрегата, содержащего одну или несколько операций, запускаемых в одном такте. Отдавая дань традиции, мы тем не менее называем задачу, рассматриваемую в этой главе, *планированием команд*, хотя более правильно было бы называть ее *планированием операций*.

12.2.2. Задача планирования команд

Задача планирования команд определена для простого блока и его графа зависимостей $\mathcal{D}$. Вершины $\mathcal{D}$ представляют операции в блоке. Ребра $\mathcal{D}$ представляют поток значений между этими операциями. Ребра ориентированы; ребро (x, y) означает, что операция y использует значение, порожденное операцией x. Кроме того, каждая вершина имеет два атрибута, *type* и *delay*. Для вершины n операция, соответствующая n, должна выполняться на функциональном блоке, заданном атрибутом $type(n)$. Эта операция занимает $delay(n)$ тактов. На рис. 12.2 показан код нашего примера и его граф зависимостей.

```
a:  loadAI  r_arp, 4  ⇒ r1
b:  add     r1, r1    ⇒ r1
c:  loadAI  r_arp, 8  ⇒ r2
d:  mult    r1, r2    ⇒ r1
e:  loadAI  r_arp, 12 ⇒ r2
f:  mult    r1, r2    ⇒ r1
g:  loadAI  r_arp, 16 ⇒ r2
h:  mult    r1, r2    ⇒ r1
i:  storeAI r1        ⇒ r_arp, 4
```

(a) Пример кода



(b) Граф зависимостей для него

Рис. 12.2 Граф зависимостей для примера

Ребра $\mathcal{D}$ показывают поток значений. Если бы мы изменили направление ребер на противоположное, то $\mathcal{D}$ стал бы лесом ОАГ.

Мы изображаем $\mathcal{D}$ с корнями, расположенными сверху, чтобы рисунок отражал порядок выполнения кода.

$\mathcal{D}$ является лесом обращенных ориентированных ациклических графов (ОАГ). Поэтому для описания $\mathcal{D}$ мы пользуемся терминологией ОАГ. Вершины, не имеющие предшественников, например *a*, *c*, *e* и *g*, называются *листьями*. Листья можно планировать настолько рано, насколько возможно, потому что они не зависят ни от каких операций. Вершины, не имеющие преемников, например *i*, называются *корнями*. Корни можно планировать только после завершения всех их предшественников.

Временно будем предполагать, что задержки всех операций известны. Обсуждение планирования с переменными задержками отложим до стр. 658.

Если известен граф зависимостей $\mathcal{D}$ для блока, то план S отображает каждую вершину n в неотрицательное целое число, обозначающее такт, в котором ее следует запустить в предположении, что первая операция запускается в такте 1. Таким образом, i-я команда содержит набор операций $\{ n \mid S(n) = i \}$. Допустимый план должен удовлетворять трем ограничениям.

1. $S(n) \geqslant 1$ для любого $n \in N$. Это ограничение запрещает операции, которые запускаются раньше, чем начнется выполнение. План, нарушающий это ограничение, построен неправильно. Для единообразия план также должен содержать хотя бы одну операцию n' с $S(n') = 1$.
2. Если $(n_1, n_2) \in E$, то $S(n_1) + delay(n_1) \leqslant S(n_2)$. Это правило необходимо для корректности. Операция не должна запускаться, пока ее операнды не определены. Некоторые процессоры ожидают, что компилятор будет вставлять `nop`, чтобы удовлетворить этому ограничению. Другие ослабляют его и используют аппаратные блокировки, чтобы отложить операцию до момента, когда ее операнды станут доступны.
3. Каждая команда содержит не больше операций каждого типа t, чем целевая машина может запустить в одном такте. Это ограничение обеспечивает реализуемость. Оно гарантирует, что никакая команда не превысит возможности функциональных блоков.

Если имеется правильно построенный, корректный и реализуемый план, то длина плана равна просто номеру такта, в котором завершается последняя операция, в предположении, что первая команда запускается в такте 1. Длину плана можно вычислить по формуле

$$L(S) = \max_{n \in N} (S(n) + delay(n) - 1).$$

Если *delay* точно отражает задержки операций, то план S должен выполниться за время $L(S)$. (Операции с переменными задержками могут вносить неточности в *delay*.) План S_i оптимален по времени, если $L(S_i) \leqslant L(S_j)$ для всех других планов S_j, содержащих тот же самый набор операций.

Качество плана

Длина плана – классическая мера качества плана. Для операций с фиксированной задержкой длина плана точно отражает время выполнения. В случае же операций с переменной задержкой фактическая стоимость операции зависит от динамических факторов. Если планировщик делает непротиворечивые предположения, то длины различных планов должны быть сравнимы.

Для измерения качества планов необязательно использовать время. Два плана S_i и S_j для одного и того же блока могут порождать разный спрос на регистры. Если процессор требует, чтобы планировщик вставлял `nop` для простаивающих функциональных блоков, то два плана могут различаться количеством выбранных операций. Наконец, S_j может потреблять меньше энергии, чем S_i, при выполнении на целевой системе, потому что никогда не использует один из функциональных блоков, поскольку выбирает меньше команд или так как вызывает меньше изменений состояний битов в логических схемах выборки и декодирования.

Что затрудняет планирование?

Главная задача планировщика – назначить каждой операции в блоке такт, в котором она будет запущена. Для каждого такта он строит набор операций. При этом планировщик должен следить за тем, чтобы никакая операция не запускалась раньше, чем станут доступны ее операнды. Если процессор может использовать аппаратные блокировки, чтобы помешать операции прочитать неопределенное значение, то план может запустить операцию и раньше – после того как запущены все операции, вычисляющие ее операнды. Однако такая операция будет ждать готовности операндов.

При попытке управлять ресурсами процессора планировщик сталкивается с противоречивыми целями. Чтобы достичь высокой производительности, он должен обеспечить занятость функциональных блоков. Чтобы замаскировать задержки многотактных операций, он должен совмещать выполнение независимых операций. Планировщик должен планировать операции с переменной задержкой, например загрузку из памяти, как можно раньше относительно использования их результатов. Все эти цели важны, и все по-

требляют один и тот же конечный ресурс: доступную в коде степень параллелизма на уровне команд. Для удовлетворения этих конфликтующих целей планировщик должен идти на компромиссы.

Измерение производительности

Основная цель планирования команд – улучшить время работы сгенерированного кода. Поскольку в этой главе рассматриваются алгоритмы планирования, нас интересует прежде всего длина плана. Однако при обсуждении производительности используется много других метрик:

- *число команд в секунду*. Эта метрика измеряет количество завершенных операций в секунду. Обычно она применяется для определения пиковой производительности оборудования; ее можно использовать для определения производительности приложений;
- *число тактов на одну команду* (Cycles per Instruction – CPI). Это результат деления затраченного времени в тактах на количество завершенных операций. Для фиксированного потока операций чем меньше CPI, тем лучше производительность;
- *эталоны*. Измеряют время, затраченное на выполнение известного набора задач. Дают информацию об общей производительности системы на конкретной рабочей нагрузке.

Метрики производительности дают представление о качестве кода, сгенерированного компилятором. Но в общем случае они недостаточно детальны, чтобы показать автору компилятора, куда направить усилия. Высокое значение CPI может объясняться структурой кода, неудачной эвристикой планирования или вставкой распределителем регистров слишком большого числа операций выгрузки в память. (Операции выгрузки часто занимают несколько тактов.)

Решение планировщика поместить операцию i в такт c влияет на самое раннее возможное размещение любой операции, зависящей от результата i – любой операции в графе $\mathcal{D}$, которая достижима из i. Если в такте c допустимо выполнить более одной операции, то выбор планировщика может изменить момент самого раннего размещения многих операций – всех, которые прямо или транзитивно зависят от каждого из возможных вариантов выбора.

Задача локального планирования команд является NP-полной для всех архитектур, кроме самых простых. На практике компиляторы аппроксимируют хорошие решения задачи планирования с помощью жадных эвристик. Большинство алгоритмов планирования, применяемых в компиляторах, основаны на одном семействе эвристических методов – *планировании списка*. В следующем разделе планирование списка описывается подробно. А далее будет показано, как обобщить эту идею на более крупные области.

ИТОГИ РАЗДЕЛА

С появлением конвейерного выполнения, неблокирующих операций, операций с переменной задержкой и нескольких функциональных блоков производительность кода стала чувствительной к порядку запуска операций. Планировщик команд в составе компилятора анализирует структуру зависимостей и задержек и изменяет порядок операций с целью уменьшить время выполнения, сохранив исходную семантику кода.

Планирование команд – трудная задача, требующая учета низкоуровневых деталей целевой архитектуры. Она также является вычислительно трудной. Планирование команд в линейном коде является NP-полной задачей практически для всех реалистичных архитектур. Поэтому большинство компиляторов используют для ее решения жадный эвристический метод, например планирование списка.

ВОПРОСЫ

1. В серии процессоров Texas Instruments C6X была введена многотактовая операция `nop`. Она принимает один аргумент: число последовательных тактов, на протяжении которых функциональный блок должен простаивать. Какие преимущества дает многотактовая `nop`? Как планировщик мог бы воспользоваться этой операцией?
2. Предположим, что операции загрузки являются неблокирующими с задержкой k, так что одна загрузка занимает k тактов, но две загрузки, запущенные в соседних тактах, занимают $k + 1$ тактов. Должно ли это наблюдение повлиять на *выбор команд*? Как алгоритмы, описанные в главе 11, могли бы учесть этот эффект?

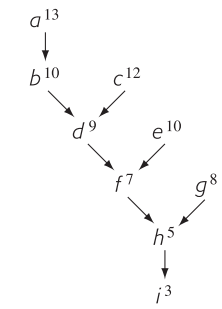
12.3. Локальное планирование

Планирование списка – жадная эвристическая парадигма. Начиная с конца 1970-х годов это основной метод, применяемый компиляторами для планирования команд, главным образом потому, что он находит приемлемые планы и легко адаптируется к новым ISA. Однако планирование списка – это подход, а не конкретный алгоритм. Существует множество реализаций и детальных эвристик. В этом разделе мы рассмотрим базовую схему планирования списка, а также две ее вариации.

Критический путь
путь в $\mathcal{D}$ с самой длинной задержкой.

Планирование списка работает с графом зависимостей блока. Граф $\mathcal{D}$ аннотируется информацией о задержках и приоритетах. Затем аннотированный граф используется для построения, такт за тактом, плана для представленного графом блока.

Граф зависимостей улавливает большинство важных для планирования свойств блока. Во врезке повторен граф на рис. 12.2(b). Для вершины n длина пути от n к корню показана в виде верхнего индекса над n. (Если бы $\mathcal{D}$ имел несколько корней, то планировщик использовал бы длину максимального пути до корня.) Индексы показывают, что путь *abdfhi* является *критическим путем* в $\mathcal{D}$; он и определяет минимальное время выполнения блока.



Граф зависимостей, аннотированный информацией о задержках

Как тогда компилятор должен спланировать это вычисление? Каждую операцию следует разместить в такте, где доступны ее операнды. Поскольку a, c, e и g – листья, они являются начальными кандидатами для планирования. Тот факт, что a лежит на критическом

пути, наводит на мысль запланировать ее в первом такте. После того как *a* запланирована, самым длинным оставшимся путем в $\mathcal{D}$ становится *cdfhi*, поэтому имеет смысл запускать *c* во втором такте. Для уже определенного префикса *ac* операции *b* и *e* являются конкурентами, т. к. их задержки одинаковы. Однако *b* зависит от результата *a*, который не будет доступен раньше такта 4. Следовательно, *eb* – лучший суффикс для *ac*, чем *be*. Продолжая в том же духе, мы придем к плану *acebdgfhi*, показанному на рис. 12.1(b).

Чтобы разрешить этот конфликт, в коде на рис. 12.1(b) имя значения, определенного операцией *e* и используемого в операции *f*, изменено с r_2 на r_3.

Но просто выбрать этот план недостаточно. Операции в обеих вершинах определяют r_2; *d*, в свою очередь, использует значение, порожденное *c*. Планировщик не может переместить *e* раньше *d*, если только не переименует результат *e*, чтобы избежать конфликта с определением r_2 в *c*. Граф зависимостей не улавливает этот конфликт, потому что он проистекает из того факта, что выполнение *e* перезаписывает значение, а не из потока какого-либо значения.

Антизависимость
Говорят, что операция *x* *антизависима* от операции *y*, если *x* предшествует *y* и *y* определяет значение, используемое в *x*. Изменение порядка вычисления *x* и *y* на противоположный могло бы привести к тому, что *x* вычислит другое значение.

Такие ограничения на порядок, как существующее между *c* и *e*, называются *антизависимостями*. Мы обозначаем антизависимость между *e* и *d* как $e \rightarrow_A d$. Планировщик должен учитывать антизависимости в исходном коде. Наш пример содержит четыре антизависимости: $e \rightarrow_A c$, $e \rightarrow_A d$, $g \rightarrow_A e$ и $g \rightarrow_A f$. Все они влекут переопределение r_2. (Существуют также ограничения, основанные на r_1, но каждая антизависимость от r_1 дублирует зависимость, основанную на потоке значений.)

В форме SSA создается пространство имен, избегающее уничтожений, а значит, и многих антизависимостей.

У планировщика есть два способа обработать этот вид антизависимости. Он может либо представлять антизависимости явно и учитывать их так же, как обрабатывает зависимости, проистекающие из потока значений, либо выполнить переименование. Первый подход ограничивает множество планов, которые может найти компилятор. Второй увеличивает размер пространства имен, что может вынудить распределитель регистров вставлять в блок код выгрузки. В описываемом нами алгоритме локального планирования предполагается, что планировщик будет переименовывать регистровые значения в коде. В разделе, посвященном построению графа зависимостей, приведены дополнительные сведения об обнаружении и представлении антизависимостей.

12.3.1. Алгоритм

Классическое планирование списка применяется к одному простому блоку. Чтобы спланировать один блок, планировщик выполняет четыре шага.

1. *Переименование*. Чтобы избежать антизависимостей от регистровых значений, компилятор их переименовывает. Этот шаг не является строго необходимым, но упрощает реализацию планировщика.
2. *Построение графа зависимостей*. Планировщик обходит блок снизу вверх. Для каждой операции он строит вершину, которая представляет саму операцию и определяемое ей значение. Затем он соединяет эту вершину с вершиной, соответствующей каждой операции, которая использует это значение.
3. *Назначение приоритета каждой операции*. Для управления выбором операций планировщик вычисляет один или несколько приоритетов для каждой вершины $\mathcal{D}$. Обычно приоритеты вычисляются в процессе обхода $\mathcal{D}$. Широко распространена схема, в которой приоритетом является максимальное расстояние до корня, взвешенное по задержке.
4. *Планирование списка*. Планировщик начинает с первого такта блока и выбирает столько операций, сколько можно запустить в этом такте. Затем он увеличивает счетчик тактов, обновляет свое представление о том, какие операции могут быть выполнены, и планирует следующий такт. Этот процесс повторяется, пока не будут спланированы все операции.

В следующих подразделах эти шаги обсуждаются более подробно.

12.3.2. Переименование

На первом шаге алгоритм переименовывает все однозначные скалярные значения в блоке, так чтобы каждому имени соответствовало одно определение. Что касается формы кода, мы предполагаем, что значения в регистрах однозначны с момента определения до момента последнего использования. Алгоритм переименования работает с этими именами регистров. Чтобы обозначить различные пространства имен, имя в оригинальном коде называется *исходным*, а имя, созданное в результате его переименования, – *виртуальным*.

> Напомним, что значение неоднозначно, если код может получить к нему доступ по нескольким именам (см. раздел 4.7.1).

Компилятор может создать пространство виртуальных имен, обладающее свойствами, которые упрощают стоящую перед ним задачу. Например, локальная нумерация значений создает пространство виртуальных имен, в котором тождественность имен влечет тождественность значений. Для планирования алгоритм переименования создает пространство имен, исключающее антизависимости между однозначными скалярными значениями. Это пространство имен, в свою очередь, упрощает последующие шаги планирования.

В одном блоке каждое определение создает новое значение. Алгоритм переименования находит все такие значения и назначает им новые виртуальные имена. На участке между определением значения и его последним использованием алгоритм заменяет ссылки на исходное имя его виртуаль-

ным именем. Этот процесс создает взаимно однозначное соответствие между значениями и именами.

Этот алгоритм встречается также при локальном распределении регистров (см. раздел 13.3).

Алгоритм, показанный на рис. 12.3, находит значения, назначает виртуальные имена и переписывает ссылки на одном обратном проходе по блоку. Его центральной структурой является отображение *SToV* исходных имен в виртуальные. В самом начале алгоритм делает все элементы *SToV* недействительными и инициализирует первое виртуальное имя нулем. Затем он обходит все операции в блоке снизу вверх.

```
VName ← 0
for i ← 0 до максимального номера  регистра-источника do
  SToV[i] ← invalid                    // инициализация
for каждой Op в блоке снизу вверх do
  for каждого определения, O, в Op do  // сначала обработать определения
    if SToV[O] = invalid then          // недопустимое определение означает,
      SToV[O] ← VName++                // что значение не используется
    O ← SToV[O]                        // O получает свое новое имя
    SToV[O] ← invalid                  // следующая ссылка – новое имя
  for каждого использования, O, в OP do // затем обработать использования
    if SToV[O] = invalid then          // началось новое значение
      SToV[O] ← VName++
    O ← SToV[O]                        // O получает свое новое имя
```

Рис. 12.3 Переименование для планирования списка

Обратите внимание, что в этом примере все имена в `store` – использования, а не определения.

Для каждой операции алгоритм обрабатывает определения раньше использований. Он заменяет определенные имена соответствующими им виртуальными; если для определенного имени нет виртуального, значит, его значение не используется в блоке. Встретив определение имени, алгоритм делает виртуальное имя недействительным, в результате чего следующее использование этого исходного имени будет иметь свое собственное виртуальное имя. Для каждого использованного имени алгоритм смотрит в *SToV*, имеет ли значение виртуальное имя; если нет, он назначает значению следующее виртуальное имя и переписывает его.

```
loadAI  r_arp,4  ⇒ r7
add     r7,r7    ⇒ r5
loadAI  r_arp,8  ⇒ r6
mult    5,r6     ⇒ r3
loadAI  r_arp,12 ⇒ r4
mult    r3,r4    ⇒ r1
loadAI  r_arp,16 ⇒ r2
mult    r1,r2    ⇒ r0
storeAI r0       ⇒ r_arp,4
```

Пример после переименования

Алгоритм обрабатывает определения раньше использований, чтобы гарантировать корректное поведение, когда одна и та же операция сначала использует имя, а затем переопределяет его. Например, для операции `addI` r_{18}, 12 ⇒ r_{18} алгоритм сначала заменит определение на *SToV*[r_{18}]. Затем он сделает *SToV*[r_{18}] недействительным, создаст новое виртуальное имя и запишет его в *SToV*[r_{18}]. Наконец, он заменит использование новым именем, взятым из *SToV*[r_{18}].

После переименования у каждого живого диапазона имеется уникальное виртуальное имя. Исходное имя, определенное в нескольких операциях, будет заменено несколькими разными виртуальными именами. Версия блока после переименования показана во врезке. Антизависимости, имевшие место во входной программе вследствие порядка операций, теперь разорваны; фактический поток значений остался неизменным. Как обсуждается в следующем разделе, r_{arp} не переименован, потому что остается жив при пересечении границ блоков.

Алгоритм переименования также играет важную роль при локальном распределении регистров (см. раздел 13.3).

Условия в конце блока

Алгоритм на рис. 12.3 предполагает, что у блока нет окружающего контекста, т. е. что блок – это вся программа. В большинстве ситуаций у блока *b* имеются предшественники и преемники в графе потока управления. Чтобы сгенерированный код правильно работал в этом контексте, алгоритм переименования должен сохранять исходные имена, остающиеся живыми при пересечении границ блоков.

Предполагается, что имена представлены целыми числами. Компилятор вправе использовать столько имен, сколько ему нужно.

До переименования компилятор может вычислить информацию Live (см. разделы 8.6.1 и 9.2.2). Затем в алгоритм нужно внести три модификации, имеющие отношение к граничным условиям. Он должен инициализировать *VName* значением, большим максимального номера регистра-источника. Для обработки блоков-преемников он должен инициализировать *SToV* значением *i* для любого $i \in \text{LiveOut}(b)$. Эти два шага гарантируют, что значения в LiveOut имеют имена, ожидаемые блоками-преемниками.

Обработка границ с блоками-предшественниками немного сложнее. Прежде чем алгоритм назначит новое *VName* какому-то исходному имени, он должен установить, определено ли это значение в текущем блоке или в каком-то его предшественнике. Поскольку входной код может определять имена несколько раз, проверки $\text{LiveIn}(b)$ недостаточно. Вместо этого компилятор должен установить, является ли текущее использование экспонированным сверху.

В процессе анализа Live компилятор вычисляет множество экспонированных сверху переменных в *b*, $\text{UEVar}(b)$. Делая это, он может пометить все экспонированные сверху использования. Затем, встретив использование имени *i* такое, что *SToV*[*i*] недействительно, алгоритм переименования может назначить *SToV*[*i*] имя *i*, а не *VName*, если использование экспонировано сверху.

12.3.3. Построение графа зависимостей

На втором шаге процесса локального планирования строится граф зависимостей $\mathcal{D}$ для простого блока. Алгоритм на рис. 12.4 прямолинеен.

Если имена – это небольшие целые числа, то *M* можно реализовать как вектор.

Он создает пустое отображение *M*, которое сопоставляет виртуальным именам в коде вершины $\mathcal{D}$. Затем он сверху вниз обходит операции в блоке. Для каждой операции *O* он создает вершину *n* в $\mathcal{D}$. Он обновляет *M*, так что *M*(*d*) = *n*, для каждого имени *d*, определенного операцией *O*. После этого добавляет ребра, ведущие из *n* в вершины, которые определяют значения, используемые в *O*.

```
создать пустое отображение M                    // определений на вершины
создать вершину undef в D                       // для неопределенного значения
for каждой операции O сверху вниз do            // обойти блок
  создать вершину n для O в D                   // n представляет O
  for каждого имени d, определенного в O do
    положить M(d) равным n
  for каждого имени u, используемого в O do     // настоящая зависимость
    if M(u) не определено then
      положить M(u) равным undef
    добавить ребро (n,M(u)) в D
  if O – операция с памятью then                // антизависимости
    добавить ребра сериализации при необходимости
```

Рис. 12.4 Построение графа зависимостей после переименования

Напомним, что при переименовании были устранены антизависимости между однозначными скалярными именами.

На последнем шаге добавляются ребра, чтобы представить антизависимости между адресами в памяти и гарантировать, что окончательный план принимает их во внимание. Идея проста. Если операция теоретически может изменить содержимое памяти, то планировщик должен гарантировать, что она не повлияет на значения, которые видны другим операциям. Детали несколько сложнее.

```
load  r17 ⇒ r5
load  r18 ⇒ r6
store r7  ⇒ r17
```

Ребро сериализации
Ребро, представляющее антизависимость, называется *ребром сериализации*.

Рассмотрим, к примеру, содержащий три операции фрагмент кода, показанный во врезке. Планировщик может переставить местами две операции `load`; ни одна из них не модифицирует содержимое памяти, поэтому изменение относительного порядка их запуска не окажет влияния на значения, помещенные в r_5 и r_6. Однако операция `store` изменяет содержимое памяти. В отсутствие информации об адресах в операциях загрузки и сохранения планировщик должен гарантировать, что операции загрузки читают из памяти раньше, чем операции сохранения туда записывают. Компилятору, возможно, придется добавить в $\mathcal{D}$ ребра, чтобы *сериализовать* эти операции.

В первой операции `load` используется тот же адрес, что в `store`; компилятор должен спланировать эти операции, так чтобы `load` читала значение раньше, чем `store` его перезапишет. Во второй операции `load` в качестве адреса используется другой регистр. Если компилятор может доказать, что значения

r_{17} и r_{18} всегда различны, то планировщик вправе поменять местами `store` и вторую `load`. Если компилятор не может этого доказать, то должен сохранить относительный порядок второй `load` и `store`.

В таблице на рис. 12.5 показано, какие ситуации могут возникнуть. Планировщик всегда может поменять порядок двух операций `load`. В случае `load` и `store` он должен сохранить их первоначальный порядок – *сериализовать* операции. Эта задержка обозначена как «серийная». В случае `store`, за которой следует `load`, планировщик должен убедиться, что `store` завершается раньше, чем запускается `load`; в общем случае он должен смириться с полной задержкой `store`.

Чтение после чтения (RAR)	`load`	`load`	нет
Запись после чтения (WAR)	`load`	`store`	серийная
Чтение после записи (RAW)	`store`	`load`	сохранение
Запись после записи (WAW)	`store`	`store`	серийная

Рис. 12.5 Задержки сериализации для операций с памятью

Для заданной пары ссылок на память компилятор должен обеспечить подходящую задержку, если не в состоянии доказать, что обе операции ссылаются на разные адреса памяти. В терминах построения графа зависимостей компилятор должен добавить ребро для представления этих ограничений, связав с ним вес, равный подходящей задержке. Компилятор может уменьшить количество антизависимостей, которые должен представить, проанализировав значения адресов памяти.

12.3.4. Вычисление приоритетов

Для принятия правильных решений о планировании компилятор вычисляет один или несколько приоритетов для каждой операции. Сталкиваясь с необходимостью выбора, он отдает предпочтение операции с наибольшим приоритетом. Планировщику необходима стратегия на случай равных приоритетов; популярный подход – использовать второй или третий приоритет. В литературе обсуждается много схем назначения приоритетов; часто используется максимальное расстояние от операции до корня $\mathcal{D}$, взвешенное по задержке.

Такие приоритеты, как взвешенная по задержке глубина, число потомков, номер в порядке обхода в ширину или в глубину, могут быть вычислены при обходе $\mathcal{D}$. Планировщик может сочетать и другие эвристики, например предпочтение операциям загрузки перед операциями сохранения, с численными манипуляциями значениями приоритетов.

Несколько приоритетов можно объединять в виде линейной комбинации. Если p_1 и p_2 – оценки, полученные на основе разных приоритетов, то планировщик может объединить их по формуле $\alpha \cdot p_1 + \beta \cdot p_2$ в одно значение. Если

$\alpha \gg \beta$, то для разрешения ситуации, когда метрики p_1 равны, по сути дела используется p_2. Альтернативно компилятор может наложить ограничение $\alpha + \beta = 1$ и настроить значения α и β для получения наилучшего результата, произведя обучение на наборе программ.

Использование приоритетов второго плана для разрешения неоднозначностей может подтолкнуть планировщик к достижению определенных целей. Например, использование задержки операции в качестве приоритета второго плана способствует размещению операций с большой задержкой ближе к началу блока. Взяв в качестве приоритета второго плана число операндов, являющихся последними случаями использования, можно уменьшить спрос на регистры.

Некоторые схемы вычисления приоритетов трудно закодировать в виде числовых значений. Например, если процессор ограничивает некоторые операции определенными функциональными блоками, то у автора компилятора может возникнуть желание поднять приоритет этих операций на их блоках. Вместо того чтобы манипулировать числовыми приоритетами, автор компилятора может отслеживать операции с ограничениями и без порознь. Планировщик может сначала выбирать из списка ограниченных операций, а при необходимости – из списка неограниченных.

12.3.5. Планирование списка

Последний шаг планирования – построение фактического плана для блока. Самым широко используемым в этой части процесса является алгоритм планирования списка – жадный эвристический алгоритм, который на практике создает хорошие планы. На рис. 12.6 представлена его базовая форма.

```
Cycle ← 1
Ready ← листья 𝒟
Active ← ∅
while (Ready ∪ Active ≠ ∅) do
  for каждого функционального блока f do
    if в Ready существует операция для f then
      положить O равной операции с наибольшим приоритетом   // выбор O по приоритету
       в Ready, которая может быть выполнена на f
      удалить O из Ready                                     // планировать O в такте Cycle
      S(O) ← Cycle
      Active ← Active ∪ { O }
  Cycle ← Cycle + 1                                          // начать новый такт
  for каждой O ∈ Active do
    if S(O) + delay(O) ≤ Cycle then                          // обновить список Ready
      удалить O из Active
      for каждого преемника s вершины O в 𝒟 do
        if s готова
          then добавить s в Ready
```

Рис. 12.6 Алгоритм планирования списка

Алгоритм выполняет абстрактное моделирование выполнения блока. Он игнорирует детали значений и операций, фокусируясь на временных ограничениях, налагаемых ребрами $\mathcal{D}$. Для слежения за временем он поддерживает часы модельного времени в переменной *Cycle*. Она инициализируется значением 1 и увеличивается по мере продвижения по блоку.

Для прослеживания операций алгоритм использует два списка. В списке *Ready* хранятся операции, «готовые» к выполнению в текущем такте; это значит, что их операнды уже вычислены. Первоначально *Ready* содержит все листья $\mathcal{D}$, поскольку они не зависят от других операций в блоке. Список *Active* содержит все операции, которые были запущены в более раннем такте, но еще не завершились.

Алгоритм планирования списка следует простой дисциплине. Каждая итерация планировщика начинается в каком-то конкретном такте. Алгоритм просматривает список *Ready* и пытается выбрать операцию для каждого функционального блока. Если для блока доступно несколько операций, то выбирается операция с наибольшим приоритетом. Перебрав все блоки, планировщик увеличивает *Cycle* и обновляет список *Ready*.

Чтобы обновить *Ready* для нового такта, планировщик сначала находит в *Active* все операции, завершившиеся в предыдущем такте. Для каждой завершившейся операции он проверяет каждого преемника s в $\mathcal{D}$ на предмет его готовности к выполнению. Точнее, он смотрит, все ли операнды, используемые s, доступны, т. е. что операция, породившая операнд, уже завершилась.

> Операция O завершается в конце такта $S(O) - first + delay(O)$.

Процесс заканчивается, когда часы модельного времени достигают первого такта, в котором завершились все операции. Если все операнды листьев $\mathcal{D}$ доступны в первом такте и $delay(op)$ точно отражает время выполнения op, то модельные часы должны давать фактическое время выполнения в тактах. Простой последующий проход может сгенерировать все операции в нужном порядке и при необходимости вставить `nop`.

Алгоритм должен соблюсти еще одно последнее ограничение. Любой условный или безусловный переход, завершающий блок, должен быть спланирован, так чтобы он не завершил выполнение, прежде чем блок закончится. Если i – завершающий условный переход, то его нельзя запланировать раньше такта с номером $L(S) + 1 - delay(i)$. Таким образом, однотактовый условный переход должен быть спланирован в последнем такте блока, а двухтактовый – не раньше чем в предпоследнем такте.

Качество плана, порожденного этим алгоритмом, зависит прежде всего от механизма выбора операции из списка *Ready*. Рассмотрим самый простой сценарий, когда список *Ready* содержит не более одного элемента на каждой итерации. В этом сильно ограниченном случае алгоритм должен сгенерировать оптимальный план. В первом такте может быть выполнена только одна операция. (В $\mathcal{D}$ должен существовать по меньшей мере один лист, а наше ограничение гарантирует, что ровно один.) В каждом последующем такте у алгоритма нет никакого выбора – либо *Ready* содержит операцию,

и алгоритм планирует ее, либо *Ready* пуст, а алгоритм ничего не планирует к запуску в этом такте.

Трудность возникает, когда в некотором такте список *Ready* содержит несколько операций. Тогда планировщик должен выбирать из нескольких готовых операций, и этот выбор критичен. Алгоритм выбирает операции с наибольшим приоритетом. Хорошо спроектированная схема назначения приоритетов включает приоритет второго плана для разрешения неоднозначностей. Предложенная ранее метрика, максимальное расстояние до корня $\mathcal{D}$, взвешенное по задержке, означает, что всегда выбирается операция на критическом пути для текущего такта конструируемого плана. В той мере, в какой влияние приоритета планирования предсказуемо, эта схема должна обеспечить сбалансированное использование самых длинных путей.

Взаимодействие между планированием и распределением

Антизависимости между операциями могут ограничить способность планировщика к изменению порядка операций. Планировщик может избежать антизависимостей с помощью переименования; однако переименование заставляет компилятор выполнять распределение регистров после планирования. Это всего лишь один пример взаимодействий между планированием команд и распределением регистров.

Основная функция планировщика – переупорядочение операций. Поскольку большинство операций одновременно определяют и используют значения, изменение относительного порядка двух операций может изменить время жизни значений. Перемещение операции x ← y + z вперед может увеличить время жизни y и z и уменьшить время жизни x. Наоборот, перемещение операции назад может сократить время жизни y и z и продлить время жизни x. Общее влияние переупорядочения операций на спрос на регистры зависит от детальной информации о временах жизни определений и их использований.

Аналогично распределение регистров может изменить задачу планирования команд. Основные функции распределителя регистров – переименование ссылок и вставка кода выгрузки в память. Обе эти функции влияют на способность планировщика порождать быстрый код. Когда распределитель отображает большое пространство виртуальных имен в меньшее пространство регистров целевой машины, он может привнести антизависимости, которые ограничат возможности планировщика. Точно так же, когда распределитель вставляет код выгрузки, он добавляет в код операции, которые сами должны быть спланированы и преобразованы в команды.

Математически можно показать, что, решая эти задачи совместно, можно получить решения, которые нельзя найти, ни запустив планировщик раньше распределителя, ни запустив распределитель раньше планировщика. Однако обе задачи сами по себе настолько сложны, что реальные компиляторы рассматривают их порознь.

Планирование операций с переменными задержками

Задержка операций с памятью часто зависит от текущего состояния иерархии памяти. Поэтому задержка конкретной операции загрузки может изменяться от двух до сотен тактов. Хуже того, данная операция, т. е. команда

в определенной позиции определенного блока, может иметь разные задержки при двух соседних выполнениях.

В отсутствие детальной информации об адресах, к которым производятся обращения, и о состоянии иерархии памяти планировщик не может точно оценить задержку. Если он будет предполагать худший случай, то рискует, что процессор будет долго простаивать. Предполагая лучший случай, он будет приостанавливать процессор при непопадании в кеш. На практике планировщик должен искать нечто среднее.

Компилятор может получить хорошие результаты, вычислив индивидуальную задержку каждой операции загрузки на основе степени параллелизма на уровне команд, доступного для компенсации этой задержки. При таком подходе, называемом *сбалансированным планированием*, операции загрузки планируются с учетом окружающего их кода, а не оборудования, на котором выполняются. Локально доступный параллелизм распределяется между всеми операциями загрузки в блоке. Эта стратегия сглаживает последствия непопадания в кеш путем планирования для каждой загрузки максимально возможной дополнительной задержки. Он не замедляет выполнение, если непопадания в кеш отсутствуют.

На рис. 12.7 показано вычисление задержек для отдельных операций загрузки в блоке. Алгоритм инициализирует задержку для каждой загрузки значением 1. Затем он рассматривает каждую операцию i в графе зависимостей блока, $\mathcal{D}$. Он находит в $\mathcal{D}$ подмножество $\mathcal{D}_i$ вершин, не зависящих от i. Концептуально это задача достижимости в $\mathcal{D}$. Чтобы найти $\mathcal{D}_i$, мы можем удалить из $\mathcal{D}$ все вершины, являющиеся транзитивными предшественниками i или транзитивными преемниками i, а также все ребра, ассоциированные с этими вершинами.

```
for каждой операции загрузки l в блоке do
    delay(l) ← 1
for каждой операции i в 𝒟 do
  положить 𝒟_i равным множеству вершин и ребер 𝒟, независимых от i
  for каждой компоненты связности C графа 𝒟_i do
    найти максимальное число загрузок, N, на путях в C
    for каждой операции загрузки l в C do
      delay(l) ← delay(l) + delay(i) ÷ N
```

Рис. 12.7 Вычисление задержек для операций загрузки

Затем алгоритм находит компоненты связности $\mathcal{D}_i$. Для каждой компоненты связности C он находит максимальное число N операций загрузки на одном пути в C. N – это количество загрузок в C, между которыми можно разделить задержку операции i, поэтому алгоритм прибавляет $delay(i)/N$ к задержке каждой загрузки в C. Для любой заданной загрузки l эта операция суммирует доли задержки каждой независимой операции i, которые можно использовать для компенсации задержки l. Используя это значение в качестве $delay(l)$, алгоритм создает план, который равномерно распределяет запас времени независимых операций между всеми загрузками в блоке.

12.3.6. Прямое и обратное планирование списка

Алгоритм планирования списка, представленный на рис. 12.6, перемещается по графу зависимостей от листьев к корням и создает план от первого цикла в блоке до последнего. Альтернативный алгоритм работает в противоположном направлении, перемещаясь от корней к листьям и строя план от последнего цикла к первому. Эта версия алгоритма называется *обратным планированием списка*, а оригинальная – *прямым планированием*.

Само по себе планирование списка не особенно дорого. Поэтому некоторые компиляторы прогоняют планировщик несколько раз с различными комбинациями эвристик и сохраняют наилучший план. (Планировщик может повторно пользоваться плодами подготовительной работы – переименования, построения графа зависимостей и вычисления приоритетов.) При такой схеме компилятор должен рассматривать как прямое, так и обратное планирование.

На практике иногда лучше оказывается прямое планирование, а иногда обратное. Различие между прямым и обратным планированиями списка заключается в порядке рассмотрения операций планировщиком. Если план критически зависит от тщательно выбранного порядка некоторого небольшого подмножества операций, то эти два направления могут давать существенно различающиеся результаты. Если критические операции встречаются ближе к листьям, то при прямом планировании они могут рассматриваться совместно, а при обратном, чтобы до них добраться, придется пройти по остальной части блока. Наоборот, если критические операции встречаются ближе к корням, то при обратном планировании они могут рассматриваться совместно, а при прямом алгоритм видит их в порядке, продиктованном решениями, принятыми, начиная с другого конца блока.

Код операции	Задержка
loadI	1
lshift	1
add	2
addI	1
cmp	1
cbr	1
store	4

Чтобы добавить конкретики, рассмотрим пример на рис. 12.8. На нем показан граф зависимостей для простого блока из теста производительности `go`, входящего в комплект SPEC 95. Компилятор добавил зависимости от операций `store` в условный переход, завершающий блок, гарантируя тем самым, что операции с памятью завершаются до начала выполнения следующего блока. Верхние индексы при вершинах графа зависимостей означают задержку от вершины до конца блока; нижние индексы различают похожие операции. В примере предполагаются задержки операций, показанные во врезке.

Мы обратили внимание на этот пример, когда изучали планирование списка, ориентированное на ILOC-машину с двумя функциональными блоками для целочисленной арифметики и одним блоком для выполнения операций с памятью. Пять операций `store` потребляют большую часть времени работы блока. Для минимизации времени выполнения план должен размещать операции `store` как можно раньше.

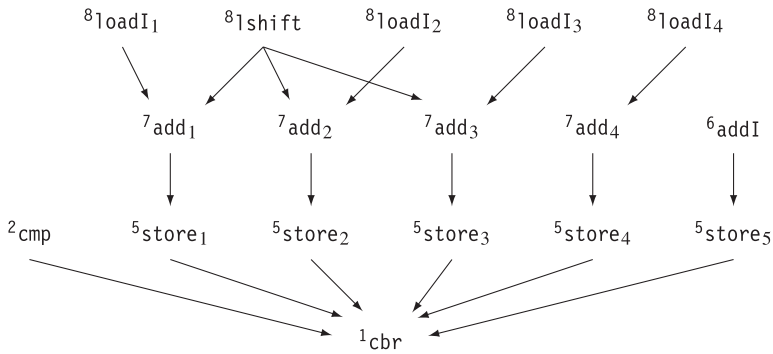


Рис. 12.8 Граф зависимостей для блока из программы go

При прямом планировании списка с использованием задержки до конца блока в качестве приоритета все операции, кроме сравнения, выполняются в порядке приоритета. Сначала планируются пять операций ранга 8, затем четыре операции ранга 7 и одна операция ранга 6. План начинается с операций ранга 5 и сдвигает `cmp` вровень со `store`, потому что `cmp` – лист. Если неоднозначности разрешаются произвольно в порядке слева направо, то получается план, показанный на рис. 12.9(а). Заметим, что операции с памятью начинаются в такте 5, что порождает план, который запускает операцию условного перехода в такте 13.

	Целое	Целое	Память
1	$loadI_1$	lshift	--
2	$loadI_2$	$loadI_3$	--
3	$loadI_4$	add_1	--
4	add_2	add_3	--
5	add_4	addI	$store_1$
6	cmp	--	$store_2$
7	--	--	$store_3$
8	--	--	$store_4$
9	--	--	$store_5$
10	--	--	--
11	--	--	--
12	--	--	--
13	cbr	--	--

(а) Прямое планирование

	Целое	Память	Целое
1	$loadI_4$	--	--
2	addI	lshift	--
3	add_4	$loadI_3$	--
4	add_3	$loadI_2$	$store_5$
5	add_2	$loadI_1$	$store_4$
6	add_1	--	$store_3$
7	--	--	$store_2$
8	--	--	$store_1$
9	--	--	--
10	--	--	--
11	cmp	--	--
12	cbr	--	--

(b) Обратное планирование

Рис. 12.9 Планы для блока из программы go

При обратном планировании списка с теми же приоритетами компилятор сначала помещает условный переход в последний слот блока. Операция `cmp` расположена на один такт раньше, что определяется величиной *delay*(`cmp`). Следующей планируется операция $store_1$ (в силу правила разрешения неоднозначностей слева направо). Ей назначается слот запуска в блоке памяти, расположенный на четыре такта раньше, как того требует величина *delay*(`store`). Планировщик последовательно заполняет предшествующие сло-

ты блока памяти другими операциями store. Заполнять слоты арифметическими операциями он начинает, как только они оказываются готовы. Первой идет add_1, на два такта раньше $store_1$. По завершении алгоритм порождает план, показанный на рис. 12.9(b).

А как насчет внеочередного выполнения?

Динамически планируемые процессоры с поддержкой внеочередного (out-of-order – OOO) выполнения могут улучшить статический план, порожденный компилятором. Так, может быть, внеочередное выполнение вообще устраняет необходимость в планировании компилятором? Чтобы разобраться в этом вопросе, важно понимать, когда именно OOO-процессор может улучшить статический план. Если на этапе выполнения складываются условия лучшие, чем предполагал планировщик, то оборудование внеочередного выполнения может запустить операцию раньше, чем она стоит в статическом плане. Такая ситуация может возникнуть на границе блока, если операнд оказался доступен раньше, чем в худшем случае. Она может возникнуть при наличии операций с переменной задержкой, например загрузки и сохранения. Видя фактические адреса во время выполнения, OOO-процессор может разрешить некоторые зависимости между загрузкой и сохранением, которые не под силу планировщику.

Однако внеочередное выполнение не отменяет необходимости планирования команд компилятором. Поскольку окно опережающего просмотра конечно, плохой план может не поддаваться улучшению. Например, окно опережающего просмотра на 50 команд не позволит процессору выполнить последовательность 100 команд целочисленной арифметики, за которой следует последовательность 100 команд арифметики с плавающей точкой, чередуя пары ⟨целое, с плавающей точкой⟩. Но чередовать более короткие последовательности операций оно должно. Внеочередное выполнение помогает компилятору, улучшая хороший, но не оптимальный план.

Обратный план занимает на один такт меньше, чем прямой. Он размещает addI раньше в блоке, что позволяет запустить $store_5$ в такте 4 – на один такт раньше, чем первая операция с памятью в прямом плане. Рассматривая задачу в другом порядке, используя те же приоритеты и способ разрешения неоднозначностей, обратный алгоритм нашел другое решение.

Почему такое происходит? Прямой планировщик помещает в план все операции ранга 8 раньше операций ранга 7. Несмотря даже на то, что операция addI листовая, прямой планировщик откладывает ее, потому что ее ранг меньше. К тому времени, как у планировщика кончатся операции ранга 8, становятся доступны другие операции ранга 7. С другой стороны, обратный планировщик помещает addI раньше трех операций ранга 8 – решение, которое прямой планировщик рассмотреть не мог.

ИТОГИ РАЗДЕЛА

Планирование списка много лет было преобладающей парадигмой, применяемой компиляторами. Для каждой операции вычисляется такт, в котором ее следует запустить. Алгоритм достаточно эффективен; его сложность прямо связана с графом зависимостей. Этот жадный эвристический подход – как в прямом, так и в обратном варианте – дает отличные результаты для одиночных блоков.

Основная структура данных для планирования команд – граф зависимостей. Он представляет поток данных в блоке. Его легко аннотировать информацией о задержках операций. Граф зависимостей естественно представляет как ограничения, так и критические пути в блоке.

ВОПРОСЫ

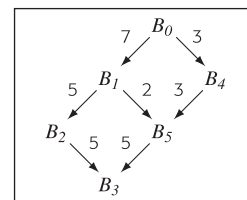
1. Список *Ready* – одна из структур, к которым планировщик списка обращается чаще всего. Сравните стоимости реализации *Ready* в виде упорядоченного списка и приоритетной очереди. Как на этот компромисс влияет ожидаемая длина списка *Ready*?
2. Предложите обоснование для каждого из следующих способов разрешения неоднозначностей.
   a. Отдавать предпочтение операциям, операнды которых находятся в регистрах, перед операциями с непосредственными операндами.
   b. Отдавать предпочтение операциям, операнды которых были определены недавно.
   c. Отдавать предпочтение загрузкам перед операциями, которые вычисляют значение.

12.4. Региональное планирование

Чтобы распространить планирование списка на регионы, авторы компиляторов используют те же стратегии, которые работают с нумерацией значений. Они строят многоблочные регионы, которые можно рассматривать как один блок, а затем применяют к этим регионам алгоритм локального планирования. Как и в случае суперлокальной нумерации значений, компилятор должен особо позаботиться о границах между многоблочным регионом и остальной частью компилируемой процедуры. В этом разделе рассматриваются три схемы, которые авторы компиляторов применяют, чтобы улучшить качество плана: суперлокальное планирование, планирование трасс и клонирование суперблоков. Каждая из них расширяет контекст, в котором компилятор применяет планирование списка.

12.4.1. Суперлокальное планирование

Напомним (см. раздел 8.3), что расширенный простой блок (РПБ) состоит из множества блоков B_1, B_2, ..., B_n, в котором B_1 имеет несколько предшественников, а все остальные блоки B_i – ровно одного предшественника, какой-то B_j, принадлежащий РПБ. Компилятор может идентифицировать РПБ на простом проходе по ГПУ. В ГПУ во врезке имеется один большой РПБ, $\{B_0, B_1, B_2, B_4\}$, и два тривиальных РПБ, $\{B_3\}$ и $\{B_5\}$. В большом РПБ есть два пути, $\langle B_0, B_1, B_2\rangle$ и $\langle B_0, B_4\rangle$. У этих путей имеется общий префикс B_0.



Пример ГПУ с частотами выполнения

Чтобы получить больший контекст, планировщик может рассматривать такие пути в РПБ, как $\langle B_0, B_1, B_2\rangle$, так, будто они являются одиночными

Ребра (B_0, B_4) и (B_1, B_5) – преждевременные выходы относительно $\langle B_0, B_1, B_2\rangle$.

блоками, при условии что учитывает общие префиксы путей и преждевременные выходы из этих путей. Этот подход позволяет компилятору применить планирование списка к более длинным последовательностям операций. В итоге увеличивается доля кода, планируемого совместно, что может уменьшить время выполнения.

Компенсационный код код, вставленный в блок B_i, чтобы уравновесить эффекты перемещения кода через границы блока вдоль пути, не включающего B_i.

Чтобы понять, как общие префиксы и преждевременные выходы осложняют планирование списка, рассмотрим, как планировщик мог бы переместить операции через границы блоков на пути $\langle B_0, B_1, B_2\rangle$ в нашем примере. Такое перемещение кода может потребовать от планировщика вставки *компенсационного кода* для сохранения корректности.

- Компилятор может переместить код вперед, т. е. в более позднее место на пути. Например, он мог бы переместить операцию *i* из B_0 в B_1. Хотя такое решение может ускорить выполнение на пути $\langle B_0, B_1, B_2\rangle$, оно убирает *i* из пути $\langle B_0, B_4\rangle$. Если *i* не является мертвой на $\langle B_0, B_4\rangle$, то планировщик должен скорректировать эту ситуацию.

Поскольку $\langle B_0, B_4\rangle$ – путь в РПБ, мы знаем, что у B_4 только один предшественник.

Для компенсации планировщик может вставить копию *i* в начало B_4. Если было допустимо перенести *i* из B_0, то также допустимо поместить ее в начало B_4. Зависимость, которая могла бы помешать помещению *i* в начало B_4, помешала бы также помещению ее в B_1. Копия *i* в B_4 не удлиняет путь $\langle B_0, B_4\rangle$, но увеличивает общий размер кода.

- Компилятор может переместить операцию назад, т. е. в более раннее место на пути. Например, он мог бы переместить операцию *j* из B_1 в B_0. Хотя это решение могло бы ускорить выполнение вдоль пути $\langle B_0, B_1, B_2\rangle$, оно добавляет *j* в путь $\langle B_0, B_4\rangle$. Это удлиняет путь $\langle B_0, B_4\rangle$ и также может изменить значения, видимые в B_4.

  Если *j* уничтожает какое-то значение, используемое в B_4, то планировщик должен переписать код, так чтобы сделать это значение доступным в начале B_4. Он мог бы вставить в начало B_4 код, который заново вычисляет это значение; переименование могло бы достичь того же эффекта. В любом случае присутствие *j* и компенсационного кода на пути $\langle B_0, B_4\rangle$ удлинит этот путь.

Механизм суперлокального планирования прост. Компилятор выбирает РПБ. При необходимости он производит переименование в регионе. Затем он планирует каждый путь в РПБ в порядке уменьшения частоты выполнения. Он строит граф зависимостей для всего пути и планирует его, игнорируя преждевременные выходы и запоминая все места пересечения границ блоков. На последующем проходе вставляется компенсационный код. Этот процесс повторяется, пока не будет спланирован каждый блок в РПБ. Каждый блок планируется ровно один раз.

Обрабатывая путь с уже запланированным префиксом, планировщик оставляет этот префикс без изменения. Это отдает предпочтение планированию горячих путей перед более холодными. Если оценки частот будущего выполнения точны, то этот подход может улучшить общее время выполнения.

В нашем примере планировщик мог бы обработать путь $\langle B_0, B_1, B_2 \rangle$. Затем он спланировал бы B_4 относительно плана для B_0. И наконец, спланировал бы тривиальные РПБ, B_3 и B_5, как одиночные блоки.

Смягчение последствий вставки компенсационного кода

Планировщик может предпринять меры, чтобы смягчить последствия вставки компенсационного кода. Он может использовать информацию о живучести, чтобы избежать части компенсационного кода вследствие перемещения вперед. Если результат перемещенной операции не является живым при входе в лежащий вне пути блок, то никакого компенсационного кода на этом пути не требуется. Планировщик может избежать вставки компенсационного кода вследствие перемещения назад, если запретит перемещение назад в блок *b* любой операции, определяющей имя $n \in \text{LiveOut}(b)$. Это ограничение может повлиять на способность планировщика улучшать код, но зато позволяет избежать удлинения других путей.

12.4.2. Планирование трасс

Трасса
ациклический путь в ГПУ, выбираемый на основе информации из профиля.

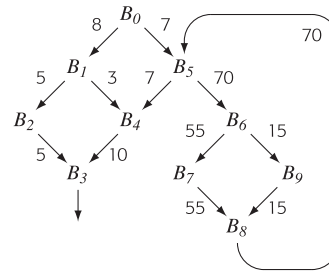
Планирование трасс обобщает подход к суперлокальному планированию, позволяя рассматривать не только пути в одном РПБ. Алгоритм строит в ГПУ ациклические пути максимальной длины, называемые трассами, и применяет к этим трассам алгоритм планирования списка. Как и в случае суперлокального планирования, каждый блок планируется ровно один раз. И точно так же ему, возможно, придется вставить компенсационный код. Алгоритм состоит из двух этапов: построение трассы и планирование.

Алгоритм следует той же базовой схеме, что и суперлокальное планирование. Он находит трассу и планирует ее, запоминая места, где в новый план нужно вставить компенсационный код. Первыми планируются самые горячие пути, так что более короткие и более ограниченные планы выполняются реже.

На рис. 12.10(а) показан алгоритм построения одной трассы. В нем используется жадная эвристика для нахождения самого горячего ациклического пути в ГПУ. Он начинается трассой, которая состоит из самого горячего из оставшихся в графе ребер – с максимальной частотой выполнения – и в цикле добавляет ребро в начало или в конец трассы. На каждом шаге он выбирает самое горячее ребро из доступных – тех, которые примыкают к трассе в начале или в конце. Когда таких ребер не осталось, трасса завершается.

```
trace ← пригодное ребро e с наибольший частотой
x ← самое горячее пригодное ребро, входящее в source(trace)
   или исходящее из sink(trace)
while (x не null) do
  добавить x с подходящей стороны трассы
  x ← самое горячее пригодное ребро, входящее в source(trace)
     или исходящее из sink(trace)
```

(a) Алгоритм построения трассы



(b) Более сложный пример

Рис. 12.10 Алгоритм построения трассы

Построение трассы

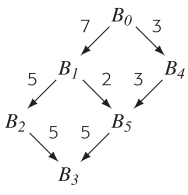
Напомним, что ребро ГПУ (x, y) называется *замыкающим цикл*, если $y \in \text{Dom}(x)$.

Алгоритм исключает некоторые ребра. Он считает ребро *непригодным* для использования в текущей трассе, если оно либо замыкает цикл, либо его начало и конец уже встречались в текущей или более ранней трассе.

Первое ограничение не дает планировщику переместить код за пределы цикла. Мы предполагаем, что оптимизатор уже произвел перемещение инвариантного относительно цикла кода (см. раздел 10.3.1) и что планировщик не должен вставлять компенсационный код, который разбивает замыкающий цикл условный переход.

Чтобы найти соседние трассы, компилятор удаляет принадлежащие трассе ребра из графа и применяет алгоритм построения трассы к оставшемуся графу. Этот процесс повторяется, пока еще имеются пригодные ребра.

Примеры



Пример ГПУ с частотами выполнения

Рассмотрим ГПУ во врезке. Алгоритм начинает с самого горячего ребра, (B_0, B_1). Он продолжает (B_0, B_1) ребрами (B_1, B_2) и (B_2, B_3) именно в таком порядке. И дальше продолжить эту трассу невозможно. Первая трасса – $\langle B_0, B_1, B_2, B_3\rangle$. Эти ребра удаляются из графа.

Построение второй трассы алгоритм начинает с ребра (B_5, B_3). Он продолжает эту трассу ребрами (B_4, B_5) и (B_0, B_4). Дальше ее продолжить нельзя. Вторая трасса – $\langle B_0, B_4, B_5, B_3\rangle$. Эти ребра удаляются из графа. Единственное оставшееся ребро, (B_1, B_5), непригодно.

Заметим, что ребро (B_8, B_5) замыкает цикл и потому непригодно.

На рис. 12.10(b) показан более сложный ГПУ. Его самое горячее ребро (B_5, B_6). Трасса продолжается ребрами (B_6, B_7), (B_7, B_8) и (B_0, B_5) в этом порядке, так что получается $\langle B_0, B_5, B_6, B_7, B_8\rangle$.

В начале построения второй трассы есть два самых горячих ребра, (B_6, B_9) и (B_9, B_8), и алгоритм выбирает одно из них. Другое ребро используется для продолжения трассы, так что получается $\langle B_6, B_9, B_8\rangle$. В этот момент ребер, пригодных для продолжения второй трассы, не осталось.

Третья трасса начинается ребром (B_4, B_5). Алгоритм добавляет ребро (B_5, B_4), так что получается $\langle B_5, B_4, B_3 \rangle$. Больше пригодных для расширения трассы ребер не осталось.

Четвертая трасса начинается ребром (B_0, B_1). Алгоритм продолжает трассу ребрами (B_1, B_2) и (B_2, B_3). Больше пригодных ребер нет. Четвертая трасса – $\langle B_0, B_1, B_2, B_3 \rangle$. В графе осталось только ребро (B_1, B_4), которое соединяет две трассы и потому непригодно.

Планирование

Зная трассу, планировщик применяет к ней алгоритм планирования списка точно так же, как при суперлокальном планировании. В случае трассы есть одна дополнительная возможность для вставки компенсационного кода; трасса может иметь промежуточные точки входа – блоки в середине трассы, имеющие нескольких предшественников, как, например, B_4 на рис. 12.10(b).

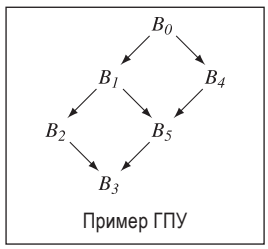
- Прямое перемещение операции *i* вперед по трассе через промежуточную точку входа может добавить *i* на путь вне трассы. Если *i* переопределяет значение, которое остается живым при переходе через промежуточную точку входа, то может понадобиться какая-то комбинация переименования или пересчета. Альтернатива – либо запретить перемещение вперед через промежуточную точку входа, либо использовать клонирование, чтобы избежать такой ситуации (см. раздел 12.4.3).
- Обратное перемещение операции *i* вперед через промежуточную точку входа может вынудить компилятор добавить *i* на путь вне трассы. Изменение простое, потому что *i* уже встречалось на пути вне трассы. Планировщик уже должен скорректировать конфликт по именам, возникший при обратном перемещении по трассе; в компенсационном коде вне трассы можно просто определить то же самое имя.

Если промежуточная точка входа появилась из ранее спланированного блока, то планировщику, возможно, придется расщепить входящее ядро, если конкретную ситуацию не удается разрешить с помощью переименования. Альтернатива – запретить перемещение вперед через промежуточную точку входа, если это вынуждает вставить компенсационный код на уже спланированную трассу.

Чтобы спланировать процедуру целиком, планировщик сначала производит переименование во всей процедуре. Затем он в цикле находит новую трассу и планирует ее, пока не окажется, что построить новую трассу невозможно. Чтобы спланировать отдельную трассу, планировщик строит граф зависимостей для трассы, вычисляет приоритеты и выполняет на ней планирование списка. Процесс продолжается, пока не будут спланированы все блоки.

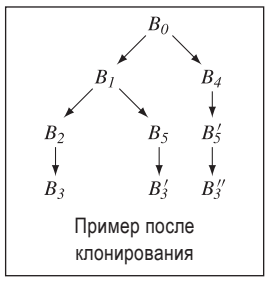
Суперлокальное планирование можно рассматривать как вырожденный случай планирования трассы, в котором промежуточные точки входа в трассу запрещены.

12.4.3. Клонирование с учетом контекста



Пример ГПУ

Точки соединения в графе потока управления могут ограничить возможности для суперлокального планирования или для планирования трасс. Чтобы улучшить результаты, компилятор может клонировать блоки с целью создать более длинные пути, свободные от соединений. Клонирование суперблока дает в точности такой эффект (см. раздел 10.6.1). Для суперлокального планирования оно увеличивает размер РПБ и длины некоторых путей в РПБ. Для планирования трасс оно помогает избежать осложнений, вызванных промежуточными точками входа в трассу. В любом случае клонирование также устраняет некоторые условные и безусловные переходы в РПБ.

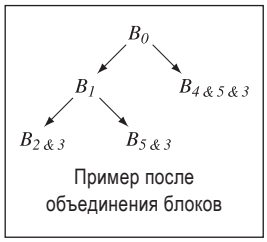


Пример после клонирования

Для примера ГПУ во врезке клонирование породило бы ГПУ на рисунке «ГПУ после клонирования». Блок B_5 был клонирован, чтобы создать отдельные экземпляры для пути из B_1 и для пути из B_4. Аналогично B_3 был клонирован дважды, чтобы создать уникальные экземпляры для всех входящих в него путей. В совокупности эти действия устранили все точки соединения в ГПУ.

В этот момент компилятор может объединить некоторые блоки, чтобы создать окончательный ГПУ, показанный на рисунке «Пример после объединения блоков». На этом шаге устраняются некоторые управляющие конструкции, например безусловный переход из B_4 в B_5'.

Теперь весь граф является одним РПБ. Компилятор может выбрать самый горячий путь, скажем $\langle B_0, B_1, B_{2\&3}\rangle$, и спланировать его. Затем он может использовать $\langle B_0, B_1\rangle$ как префикс, чтобы спланировать $B_{5\&3}$. Наконец, он может использовать B_0 как префикс, чтобы спланировать $B_{4\&5\&3}$. Клонирование ГПУ устраняет большую часть интерференции между этими путями.



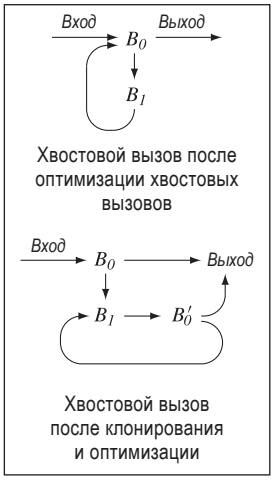
Пример после объединения блоков

Сравните этот результат с результатом суперлокального планирования для исходного ГПУ. Суперлокальный алгоритм спланировал B_4 относительно B_0, как в клонированном графе. Однако суперлокальный алгоритм спланировал B_3 и B_5 без предшествующего контекста. В клонированном графе каждый из этих блоков продублирован и спланирован со множеством своих префиксов. Цена такой специализации – увеличенный размер кода.

Процедуры с хвостовой рекурсией также могут выиграть от клонирования. Напомним (см. разделы 7.5.2 и 10.4.1), что процедура обладает хвостовой рекурсией, если ее последним действием является хвостовой вызов, т. е. рекурсивный вызов самой себя. Обнаружив хвостовой вызов, компилятор может преобразовать его в безусловный переход на точку входа в процедуру.

Как показано во врезке, клонирование блоков в процедуре с хвостовой рекурсией может создать форму кода, при которой планировщик получает более длинный блок и более ограниченный контекст для работы.

На первой диаграмме во врезке показан абстрагированный ГПУ для процедуры с хвостовой рекурсией, после того как хвостовой вызов был оптимизирован. В блок B_0 можно войти по двум путям: через точку входа в процедуру и из B_1. Планировщик должен сделать худшее предположение о том, что предшествует B_0. Клонировав B_0, как показано на рисунке ниже, компилятор может сделать так, что управление будет попадать в B'_0 только по одному ребру, что может улучшить результаты регионального планирования. Чтобы еще упростить ситуацию, компилятор может объединить B'_0 с концом B_1, создав одноблочное тело цикла. Результирующий цикл можно планировать либо локальным планировщиком, либо планировщиком циклов – как удобнее.



ИТОГИ РАЗДЕЛА

Методы регионального планирования строят более длинные линейные участки кода, к которым можно применить планирование списка. Инфраструктура регионального планирования просто предоставляет больше контекста и больше операций планировщику списка в попытке дать ему больше свободы и возможностей. Качество кода, порождаемого региональным планировщиком, до некоторой степени определяется качеством планировщика, на котором он основан.

Суперлокальное планирование, планирование трасс и клонирование суперблока перед суперлокальным планированием могут потребовать включения компенсационного кода, который потребляет дополнительную память. На некоторых путях, возможно, придется ввести дополнительные операции. Однако опыт показывает, что выигрыш от регионального планирования часто перевешивает затраты.

ВОПРОСЫ

1. При региональном планировании перемещение кода через границы блоков может повлечь за собой необходимость вставки компенсационного кода. Как компилятор может распознать ситуации, когда перемещение операции через границу блока не требует компенсационного кода?
2. И планирование трассы, и клонирование стремятся улучшить результаты суперлокального планирования. Сравните эти подходы и их ожидаемые результаты.

12.5. Дополнительные темы

Оптимизация сгенерированного компилятором кода еще со времен первого компилятора FORTRAN уделяла особое внимание улучшению кода в циклах. Причина проста: код внутри цикла выполняется чаще, чем код вне цикла.

Это наблюдение привело к разработке специализированных методов, старающихся уменьшить полное время выполнения цикла. Эти планировщики циклов могут создавать лучшие планы, чем региональные планировщики, по одной простой причине: они учитывают поток значений во всем цикле, включая замыкающий его условный переход.

Планирование цикла вступает в игру, когда планировщик по умолчанию не может создать компактный и эффективный код цикла. Если в спланированном цикле нет неиспользуемых слотов запуска, то планирование цикла вряд ли поможет. С другой стороны, если количество пустых слотов запуска велико, то планирование цикла может повысить загрузку функциональных блоков и сократить общее время выполнения цикла.

Программный конвейер создает цикл, повторяя поведение аппаратного конвейера. Интервалы выполнения соседних итераций цикла перекрываются, так что операции из двух или более итераций выполняются одновременно. Конвейерный цикл уменьшает промежуток времени между моментами начала i-й и $(i+1)$-й итераций – *интервал инициации* цикла. Чем меньше интервал инициации, тем быстрее выполняется цикл в целом.

Интервал инициации
Интервалом инициации цикла называется измеряемый в тактах промежуток времени между началами i-й и $(i+1)$-й итераций.

12.5.1. Стратегия организации программного конвейера

Почему программный конвейер дает выигрыш, понятно. Для обычного цикла интервал инициации равен числу тактов, c, необходимых для выполнения одной итерации цикла. Если в цикле выполняется n итераций, то общее время выполнения равно $n \cdot c$.

В конвейерном цикле $(i+1)$-я итерация начинается раньше, чем закончилась i-я итерация. При такой стратегии интервал инициации, ii, уменьшается, что приводит к сокращению времени выполнения цикла.

Конвейерный цикл начинает новую итерацию каждые ii тактов. Каждая из первых $n-1$ итераций обходится всего в ii тактов. Последняя итерация обходится в полную стоимость, c, потому что ей не с чем перекрываться. Таким образом, полная стоимость конвейерного цикла равна $(n-1) \cdot ii + c$ тактов. По построению, мы можем гарантировать, что $ii \leqslant c$.

Ядро цикла
Центральная часть цикла с программным конвейером, *ядро*, выполняет основную часть итераций цикла в режиме частичного перекрытия.

С точки зрения формы кода, преобразованный цикл состоит из конвейерного *ядра*, которое выполняет вычисления в стационарном режиме, а также пролога и эпилога, где происходят инициализация и финализация цикла. Общий эффект аналогичен аппаратному конвейеру, который одновременно выполняет несколько разных операций.

Когда конвейерный цикл выполняется, пролог заполняет конвейер. Если ядро выполняет операции из трех итераций исходного цикла, то каждая ядерная итерация обрабатывает приблизительно одну треть каждой актив-

ной итерации исходного цикла. Чтобы начать выполнение, пролог должен проделать достаточно работы, дабы подготовиться к последней трети итерации 1, второй трети итерации 2 и первой трети итерации 3.

После того как ядро цикла завершится, эпилог завершает последние итерации, которые опустошают конвейер. В нашем гипотетическом примере эпилог должен выполнить последнюю треть предпоследней итерации и последние две трети конечной итерации. Пролог и эпилог увеличивают общий размер кода цикла. Чтобы добавить обсуждению конкретики, рассмотрим следующий цикл на C:

```
for (i=0; i < 200; i++)
    z[i] = x[i] * y[i];
```

Предположим, что нижние границы x, y и z равны нулю. На рис. 12.11 показан ILOC-код, который компилятор мог бы сгенерировать для цикла, после оптимизации и локального планирования. Были применены снижение стоимости операций и замены теста линейной функции (см. раздел 10.7.2), так что адресные выражения для x, y и z модифицированы с помощью операций addI, а проверка в конце цикла переписана в терминах смещения от начала x, что устраняет необходимость в i.

Такт	Функциональный блок 0			Комментарии
–4	loadI	@x	⇒ $r_{@x}$	Инициализировать цикл
–3	loadI	@y	⇒ $r_{@y}$	начальными загрузками
–2	loadI	@z	⇒ $r_{@z}$	
–1	addI	$r_{@x}$, 796	⇒ r_{ub}	
1	L_1: loadAO	r_{arp}, $r_{@x}$	⇒ r_x	Получить x[i] & y[i]
2	loadAO	r_{arp}, $r_{@y}$	⇒ r_y	
3	addI	$r_{@x}$, 4	⇒ $r_{@x}$	Увеличить указатели
4	addI	$r_{@y}$, 4	⇒ $r_{@y}$	в тени загрузок
5	mult	r_x, r_y	⇒ r_z	Собственно работа
6	cmp_LE	$r_{@x}$, r_{ub}	⇒ r_{cc}	Тень mult
7	storeAO	r_z	⇒ r_{arp}, $r_{@z}$	Сохранить результат
8	addI	$r_{@z}$, 4	⇒ $r_{@z}$	Увеличить указатель z
9	cbr	r_{cc}	→ L_1, L_2	Переход, замыкающий цикл
	L_2: ...			

Рис. 12.11 Пример цикла, локальный планировщик, один функциональный блок

Код на рис. 12.11 спланирован для машины с одним функциональным блоком. Задержки операции показаны во врезке. В первом столбце таблицы показан такт относительно начала цикла (метка L_1), в котором запускается каждая операция.

Код операции	Задержка
loadAO	3
storeAO	3
loadI	1
addI	1
mult	2
cmp_LE	1
cbr	1

Задержки в примере

В коде перед циклом инициализируются указатели для массивов ($r_{@x}$, $r_{@y}$, $r_{@z}$). Верхняя грани-

ца диапазона $r_{@x}$ помещается в r_{ub}; этот регистр используется для проверки в конце цикла. В теле цикла загружаются x и y, выполняется умножение и результат сохраняется в z. В плане используются все слоты запуска в тени операций с длительной задержкой. Пока выполняется длительная загрузка, обновляются $r_{@x}$ и $r_{@y}$. Сравнение выполняется одновременно с умножением. Слоты после сохранения заполнены обновлением $r_{@z}$ и условным переходом. В итоге машина с одним функциональным блоком полностью занята.

Рассмотрим, что происходит с тем же кодом на суперскалярном процессоре с двумя функциональными блоками и теми же задержками. Предположим, что операции загрузки и сохранения должны выполняться на блоке 0, функциональные блоки приостанавливают работу, когда операция запускается раньше, чем готовы ее операнды, и что процессор не может запускать операции на приостановленном блоке. На рис. 12.12 показана трасса выполнения первой итерации цикла. Операция `mult` в такте 3 приостанавливается, потому что ни r_x, ни r_y не готовы. Она приостановлена в такте 4 в ожидании r_y, снова начинает выполнение в такте 5 и порождает r_z в такте 6. Операция `storeAO` должна приостановиться до начала такта 7. В предположении, что оборудование может сообщить, что $r_{@z}$ содержит адрес, отличный от находящихся в $r_{@x}$ и $r_{@y}$, процессор может запустить первую операцию `loadAO` для второй итерации в такте 7. Если это не так, то процессор приостановится, пока сохранение не завершится.

Такт	Функциональный блок 0				Функциональный блок 1			
–2		loadI	@x	⇒ $r_{@x}$	loadI	@y	⇒ $r_{@y}$	
–1		loadI	@z	⇒ $r_{@z}$	addI	$r_{@x}$, 796	⇒ r_{ub}	
1	L_1:	loadAO	r_{arp},$r_{@x}$	⇒ r_x	nop			
2		loadAO	r_{arp},$r_{@y}$	⇒ r_y	addI	$r_{@x}$,4	⇒ $r_{@x}$	
3		addI	$r_{@y}$,4	⇒ $r_{@y}$	mult	r_x,r_y	⇒ r_z	
4		cmp_LE	$r_{@x}$,r_{ub}	⇒ r_{cc}	nop			
5		storeAO	r_z	⇒ r_{arp},$r_{@z}$	addI	$r_{@z}$,4	⇒ $r_{@z}$	
6		*ожидание готовности* r_z			cbr	r_{cc}	→ L_1, L_2	
7	*... начало следующей итерации ...*							

Рис. 12.12 Пример цикла, локальный планировщик, два функциональных блока

При наличии двух функциональных блоков код выполняется быстрее. Предцикл сокращается с четырех до двух тактов. Интервал инициации сокращается с девяти до шести тактов. Критический путь выполняется так быстро, как можно ожидать; умножение можно запустить до готовности r_y, и выполняется оно максимально быстро. Сохранение начинается, как только будет готов операнд r_z. Некоторые слоты запуска пусты (на блоке 0 в такте 6 и на блоке 1 в тактах 1 и 4).

Переупорядочение линейного кода может изменить план выполнения. Например, перемещение обновления $r_{@x}$ перед загрузкой из $r_{@y}$ позволяет процессору запустить обновления $r_{@x}$ и $r_{@y}$ в тех же тактах, что загрузки из

этих регистров. Это дает возможность запускать некоторые операции раньше в плане, но ничего не делает для ускорения критического пути. Итоговый результат тот же – шеститактовый цикл.

Конвейерное выполнение кода может еще сократить интервал инициации, как показано на рис. 12.13. Верхняя часть, над штриховой линией, содержит пролог цикла. Средняя часть содержит пятитактовое ядро. Нижняя часть, под второй штриховой линией, содержит эпилог цикла. В следующем подразделе представлен алгоритм, который генерирует ядро этого конвейерного цикла.

	Такт		Функциональный блок 0			Функциональный блок 1		
Пролог	–6		loadI	@x	⇒ $r_{@x}$	loadI	@y	⇒ $r_{@y}$
	–5		loadAO	r_{arp},$r_{@x}$	⇒ r_x	addI	$r_{@x}$, 796	⇒ r_{ub}
	–4		loadAO	r_{arp},$r_{@y}$	⇒ r_y	addI	$r_{@x}$,4	⇒ r_x
	–3		loadI	@z	⇒ $r_{@z}$	addI	$r_{@y}$,4	⇒ r_y
	–2		nop			nop		
	–1		mult	r_x,r_y	⇒ r_z	nop		
Ядро	1	L_1:	loadAO	r_{arp},$r_{@x}$	⇒ r_x	addI	$r_{@x}$,4	⇒ $r_{@x}$
	2		loadAO	r_{arp},$r_{@y}$	⇒ r_y	addI	$r_{@y}$,4	⇒ $r_{@y}$
	3		cmp_LE	$r_{@x}$,r_{ub}	⇒ r_{cc}	nop		
	4		storeAO	r_z	⇒ r_{arp},$r_{@z}$	addI	$r_{@z}$,4	⇒ $r_{@z}$
	5		cbr	r_{cc}	→ L_1, L_2	mult	r_x,r_y	⇒ r_z
Эпилог	+1	L_1:	nop			nop		
	+2		storeAO	r_z	⇒ r_{arp},$r_{@z}$	nop		
	+3		nop			...		

Рис. 12.13 Пример цикла, программный конвейер, два функциональных блока

12.5.2. Алгоритм программного конвейера

Чтобы организовать конвейерное выполнение цикла, планировщик сначала строит оценку минимального интервала инициации, необходимого циклу. Затем он пытается спланировать цикл в ядре этой длины; если не получается, он увеличивает размер ядра на единицу и пробует снова. Этот процесс рано или поздно остановится; в худшем случае планировщик достигнет интервала инициации оригинального цикла без конвейера. Получив ядро, планировщик генерирует соответствующие ему пролог и эпилог.

Оценка размера ядра

Чтобы оценить размер ядра, планировщик цикла может вычислить нижние границы числа тактов, которые должно содержать ядро.

- Первая нижняя граница вытекает из простого наблюдения: план должен запустить каждую операцию в теле цикла. Число тактов, необходимых для запуска всех операций, ограничено снизу величиной

  $$RC = max_u(\lceil O_u/N_u \rceil),$$

где u пробегает все типы функциональных блоков, O_u – число операций типа u в цикле, а N_u – число функциональных блоков типа u. *RC* представляет ресурсное ограничение.

Рекурренция
циклическое вычисление, которое создает цикл в графе зависимостей.
Рекурренция должна охватывать несколько итераций.

- Вторая нижняя граница вытекает из другого простого наблюдения: интервал инициации должен быть достаточно велик для того, чтобы завершилась каждая рекурренция в цикле. Планировщик может вычислить эту нижнюю границу по длинам рекурренций:

$$DC = max_r(\lceil d_r/k_r \rceil),$$

где r пробегает все рекурренции в теле цикла, d_r – накопительная задержка на рекурренции r, а k_r – число итераций, охватываемых r. *DC* – ограничение зависимости.

Эти две оценки, *RC* и *DC*, являются нижними границами, потому что планировщик может и не найти план, умещающийся в такое число тактов.

Планировщик может использовать величину $ii = max(RC, DC)$ в качестве первой оценки интервала инициации. Он пытается спланировать цикл размера не больше ii. Если эта попытка терпит крах, то он увеличивает ii и пробует снова. Мы знаем, что эта тактика приведет к успеху для некоторого ii в диапазоне $max(RC,DC) \leqslant ii \leqslant c$, где c – длина плана, найденного локальным планировщиком.

Чтобы вычислить *RC*, планировщик должен учитывать две разные оценки. Цикл состоит из девяти операций, планируемых на двух функциональных блоках, что дает значение *RC*, равное $\lceil 9/2 \rceil = 5$. В нем имеется три операции с памятью, которые могут выполняться только на блоке 0; эти операции дают значение *RC*, равное $\lceil 3/1 \rceil = 3$. Таким образом, ограничение *RC* для цикла равно $max(5, 3) = 5$.

Для вычисления *DC* нужен граф зависимостей. Он содержит три реккуренции, по одной для `r@x`, `r@y` и `r@я`. Для всех трех накопительная задержка равна 1 и каждая охватывает одну итерацию, так что $DC = \lceil 1/1 \rceil = 1$.

При $RC = 5$ и $DC = 1$ имеет $ii = max(RC, DC) = 5$.

Планирование ядра

Планирование по модулю
Планирование списка с циклическим тактовым генератором иногда называют *планированием по модулю*.

Циклически порожденная зависимость
зависимость, которая представляет значение, переносимое по ребру ГПУ, соответствующему условному переходу, замыкающему цикл.

Для планирования ядра компилятор использует планировщик списка с планом фиксированной длины на ii слотов. Обновление тактового генератора, *Cycle* на рис. 12.6, производится по модулю ii. Такты в графе зависимостей привносят два осложнения в работу планировщика.

Во-первых, планировщик должен понимать, что циклически порожденные

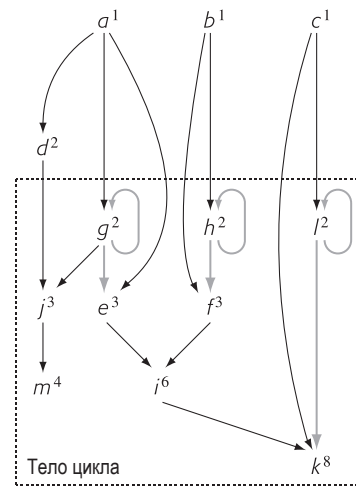
зависимости, как, например, (*g*, *e*), (*h*, *f*) и (*l*, *k*), не ограничивают первую итерацию цикла. На первой итерации используются значения, вычисленные до начала цикла, которые связаны с ребрами, идущими из вершин *a*, *b* и *c* графа зависимостей на рис. 12.14(b).

```
a:      loadI   @x          ⇒ r@x
b:      loadI   @y          ⇒ r@y
c:      loadI   @z          ⇒ r@z
d:      addI    r@x, 796    ⇒ rub

e:  L1: loadAO  rarp, r@x   ⇒ rx
f:      loadAO  rarp, r@y   ⇒ ry
g:      addI    r@x, 4      ⇒ r@x
h:      addI    r@y, 4      ⇒ r@y
i:      mult    rx, ry      ⇒ rz
j:      cmp_LE  r@x, rub    ⇒ rcc
k:      storeAO rz          ⇒ rarp, r@z
l:      addI    r@z, 4      ⇒ r@z
m:      cbr     rcc         → L1, L2

n:  L2: ...
```

(a) Первоначальный код примера цикла



(b) Граф зависимостей

Рис. 12.14 Граф зависимостей для примера цикла

Во-вторых, циклически порожденные зависимости раскрывают критические антизависимости. В нашем примере *g* читает и записывает $r_{@x}$. Каким бы ни было значение в $r_{@x}$, *g* не может обновить $r_{@x}$, прежде чем *e* использует хранящееся в нем значение. Следовательно, *g* не может быть запланировано раньше *e*. (Они могут быть запущены в одном такте, потому что *e* читает свои аргументы в начале такта, а *g* записывает результат в конце такта.) В силу такой же логики *h* не может предшествовать *f* и *l* не может предшествовать *k*.

На рис. 12.15 показаны шаги, которые планировщик выполняет для кода в нашем примере. Предполагается, что имеется процессор с двумя функциональными блоками, причем операции с памятью могут выполняться только на блоке 0, и интервал инициации равен 5.

Такт 1. Список готовых начинается с (*e*, *f*). Антизависимости исключают *g*, *h* и *l*. Планировщик выбирает *e* и размещает *e* на блоке 0. Такое размещение удовлетворяет антизависимость от *g*, поэтому *g* перемещается в список готовых. *g* расположена дальше от корня, чем *f*, поэтому планировщик размещает *g* на блоке 1, как показано на первом рисунке.

Такт 2. *f* и *j* готовы. Ориентируясь на расстояние до корня, планировщик размещает *f* на блоке 0, что добавляет *h* в список готовых. Планировщик выбирает *h* на основе расстояния до корня и размещает *h* на блоке 1.

Такт 3. Готова только *j*. Планировщик размещает ее на блоке 0.

Такт 1

Такт	F0	F1
1	*e*	*g*

Такт 2

Такт	F0	F1
1	*e*	*g*
2	*f*	*h*

Такт 3

Такт	F0	F1
1	*e*	*g*
2	*f*	*h*
3	*j*	—

Такт 4

Такт	F0	F1
1	*e*	*g*
2	*f*	*h*
3	*j*	—
4	—	—

Такт 5

Такт	F0	F1
1	*e*	*g*
2	*f*	*h*
3	*j*	—
4	—	—
5	*m*	*i*

Такт 6

Такт	F0	F1
1	*e*	*g*
2	*f*	*h*
3	*j*	—
4	—	—
5	*m*	*i*

Такт 7

Такт	F0	F1
1	*e*	*g*
2	*f*	*h*
3	*j*	—
4	*k*	*l*
5	*m*	*i*

Рис. 12.15 Шаги планирования по модулю в нашем примере

Такт 4. В этой точке *m* готова. Однако планировщик ограничен необходимостью разместить условный переход, так чтобы управление передавалось после такта 5. Поэтому такт 4 остается пустым.

Такт 5. *m* и *i* готовы. Планировщик размещает их в такте 5.

Такт 1. Следующий такт по модулю равен 1. Список готовых пуст, поэтому планировать нечего. *k* все еще находится в списке активных, поэтому планировщик увеличивает счетчик и продолжает работу.

Такт 2. Операция *k* готова. Поскольку это `store`, она должна выполняться на блоке 0. В такте 2 блок 0 занят, поэтому планировщик увеличивает счетчик.

Такт 3. Блок 0 все еще занят, поэтому планировщик увеличивает счетчик.

Такт 4. В такте 4 блок 0 свободен. Планировщик помещает *k* на блок 0 для выполнения в такте 4. Это размещение удовлетворяет антизависимость и делает *l* готовой. Планировщик размещает *l* на блоке 1 в такте 4.

Планировщик в цикле увеличивает счетчик (по модулю 5), пока списки активных и готовых не опустеют. Поскольку ни операция *k*, ни операция *l* не имеют потомков в графе зависимостей, оба списка становятся пустыми, и алгоритм завершается.

Планирование по модулю терпит неудачу, когда не может найти слот для запуска какой-то операции. Если такое происходит, то алгоритм увеличивает интервал инициации на 1 и пробует снова. Этот процесс обязательно останавливается: когда *ii* достигает длины оригинального плана, планирование производится в рамках одной итерации.

Изменение способа вычисления адреса может уменьшить ядро, как показано в упражнении 9.

Аналогично применение автоинкремента к $r_{@x}$, $r_{@y}$ и $r_{@z}$ удалило бы операции из цикла и изменило бы план.

Генерирование кода пролога и эпилога

В принципе, сгенерировать код пролога и эпилога просто. В обоих случаях ключевое наблюдение заключается в том, что компилятор может использовать граф зависимостей для направления процесса в нужную сторону.

Чтобы сгенерировать код пролога, компилятор начинает с каждого экспонированного сверху использования в цикле и следует вдоль графа зависимостей на этапе обратного планирования. Для каждого экспонированного сверху использования он должен сгенерировать цепочку операций, которая вычисляет необходимое значение, правильно спланировав их с учетом задержек. Чтобы сгенерировать код эпилога, компилятор начинает с каждого экспонируемого вниз использования в цикле и следует вдоль графа зависимостей на этапе прямого планирования.

Для ядра нашего конвейерного цикла на рис. 12.16 сверху экспонированы использования $r_{@x}$, $r_{@y}$, $r_{@x}$, r_{ub} и r_z. Первые четыре использования были в оригинальном цикле. Экспонированная ссылка на r_z появилась, потому что планировщик переместил операцию `storeAO` с i-й итерации ядра на $(i + 1)$-ю.

Такт	Функциональный блок 0	Функциональный блок 1
1	L_1: loadAO r_{arp},$r_{@x}$ ⇒ r_x	addI $r_{@x}$,4 ⇒ $r_{@x}$
2	loadAO r_{arp},$r_{@y}$ ⇒ r_y	addI $r_{@y}$,4 ⇒ $r_{@y}$
3	cmp_LE $r_{@x}$,r_{ub} ⇒ r_{cc}	nop
4	storeAO r_z ⇒ r_{arp},$r_{@z}$	addI $r_{@z}$,4 ⇒ $r_{@z}$
5	cbr r_{cc} → L_1, L_2	mult r_x,r_y ⇒ r_z

Рис. 12.16 Окончательный план ядра для конвейерного цикла

Таким образом, пролог по-прежнему должен содержать операции, определяющие $r_{@x}$, $r_{@y}$, $r_{@x}$ и r_{ub}. Кроме того, он должен вычислять первое значение r_z и увеличивать адреса в регистрах $r_{@x}$ и $r_{@y}$, так чтобы на первой итерации ядра читались вторые значения `x` и `y`. С такими ограничениями получается пролог, показанный на рис. 12.13.

В этом случае эпилог простой. Ядро выполняет почти всю последнюю итерацию, кроме `storeAO`. Таким образом, эпилог содержит эту операцию, размещенную так, чтобы умножение завершилось до запуска сохранения.

12.5.3. Последний пример

```
c = 0;
for (i=0;i<n;i++) {
    c = c + a[i] * b[i];
}
```

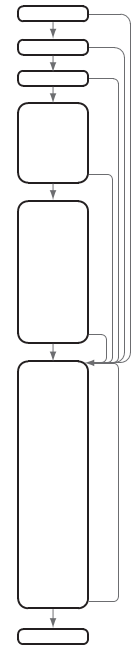
Пример цикла на C

В компиляторах для высокопроизводительных архитектур ЦПОС (цифровой процессор обработки сигналов, *англ.* DSP) программный конвейер был нужен, чтобы воспользоваться всеми преимуществами их широких команд. Например, рассмотрим простой цикл, показанный во врезке. В 2002 году наш коллега из компании Texas Instruments прислал нам код, который их компилятор сгенерировал для выполнения данного цикла на процессорах TMS320C6x.

На рис. 12.17(a) показан окончательный код, оптимизированный так, что содержит ядро с одной командой и 8 операциями.

```
          B     .S1   LOOP         ; branch to loop
          B     .S1   LOOP         ; branch to loop
          B     .S1   LOOP         ; branch to loop
          B     .S1   LOOP         ; branch to loop
||        ZERO  .L1   A2           ; zero A side product
||        ZERO  .L2   B2           ; zero B side product
          B     .S1   LOOP         ; branch to loop
||        ZERO  .L1   A3           ; zero A side accumulator
||        ZERO  .L2   B3           ; zero B side accumulator
||        ZERO  .D1   A1           ; zero A side load value
||        ZERO  .D2   B1           ; zero B side load value
-------------------------------------------------------------
LOOP:     LDW   .D1   *A4++, A1    ; load a[i] & a[i+1]
||        LDW   .D2   *B4++, B1    ; load b[i] & b[i+1]
||        MPY   .M1X  A1, B1, A2   ; a[i] * b[i]
||        MPYH  .M2X  A1, B1, B2   ; a[i+1] * b[i+1]
||        ADD   .L1   A2, A3, A3   ; ca += a[i] * b[i]
||        ADD   .L2   B2, B3, B3   ; cb += a[i+1] * b[i+1]
|| [B0]   SUB   .S2   B0, 1, B0    ; decrement loop counter
|| [B0]   B     .S1   LOOP         ; branch to loop
-------------------------------------------------------------
          ADD   .L1X  A3, B3, A3   ; c = ca + cb
```

(a) Ассемблерный код TMS320C6x для нашего примера



(b) ГПУ для цикла

Рис. 12.17 Программный конвейер в компиляторе VLIW DSP

Процессоры C6x могут упаковать в один такт до восьми операций. Знак || в начале строки означает, что операция входит в тот же командный пакет, что и операция на предыдущей строке. У процессора имеется два банка регистров, A и B. Коды, начинающиеся точкой, например .S1, обозначают, в каком месте VLIW-команды начинается эта операция. Наконец, операция загрузки двойного слова (LDW) читает из памяти два числа, операции «умножить младшие» (MPY) и «умножить старшие» (MPYH) перемножают младшие и старшие слова соответственно.

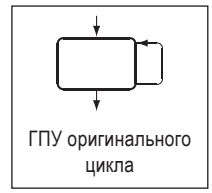
Некоторые детали в этом фрагменте кода опущены. Мы предполагаем, что регистры B0, A4 и B4 уже были инициализированы ранее.

Компилятор конвейеризовал цикл и уместил его в одну VLIW-команду – операции между штриховыми линиями на рисунке (a). Задержка условного перехода составляет 5 тактов; операции могут быть выполнены в слотах его отсрочки. Таким образом, код пролога цикла состоит из пяти команд, каждая из которых содержит операцию условного перехода. Последние две команды пролога также обнуляют различные регистры, используемые в цикле, чтобы обеспечить правильные начальные условия.

С точки зрения программного конвейера, компилятор нашел минимальный интервал инициации – использовать меньше одного такта не получится.

Но поток управления, показанный на рисунке (b), противоречит интуиции. Пролог заполняет конвейер ожидающими операциями перехода, так что каждая итерация ядра выполняется в слоте отсрочки перехода, запущенного несколькими тактами ранее. Мы не думаем, что типичный программист придумал бы такую форму кода.

Этот пример появился в ходе исследований на тему того, как вывести точные представления, включая ГПУ, из оптимизированного ассемблерного кода. Без слов понятно, что ГПУ на рис. 12.17(b) запутывает структуру оригинального цикла. Алгоритмически вывести этот ГПУ сложно. Вывести ГПУ оригинального цикла из оптимизированного ассемблерного кода кажется гораздо сложнее. Оригинальный ГПУ показан во врезке.



12.6. Итоги и перспективы

Чтобы добиться разумной производительности на современном процессоре, компилятор должен выполнять планирование команд. В большинстве компиляторов используется планирование списка. Алгоритм легко адаптируется и параметризуется меняющимися схемами назначения приоритетов, правилами разрешения неоднозначностей и даже направлением планирования. Он устойчив в том смысле, что дает хорошие результаты для самых разных программ. На практике он часто находит оптимальное по времени решение.

Вариации планирования списка, работающие в более крупных регионах, сталкиваются с проблемами, возникающими отчасти из-за всё возрастающей сложности процессоров. Региональное планирование и планирование циклов – это ответы на увеличение количества конвейеров и их индивидуальных задержек. По мере того как машины становятся сложнее, планировщикам необходимо больше контекста, позволяющего выявить достаточно параллелизма на уровне команд, чтобы загрузить машину работой. Планирование трасс было разработано для VLIW-архитектур, которые предоставляют много функциональных блоков. Программный конвейер предлагает способ увеличить количество операций, запускаемых в одном такте, и таким образом уменьшить полное время выполнения цикла.

Замечания к этой главе

Задачи планирования возникают во многих областях, начиная со строительства, промышленного производства, доставки посылок и кончая планированием полезной нагрузки космического корабля. Планированию, в том числе его многим специализированным вариантам, посвящена обширная литература. Планирование команд как отдельная задача изучалось с 1960-х годов.

Для простых ситуаций существуют алгоритмы, гарантирующие построение оптимальных планов. Например, для машины с одним функциональным блоком и одинаковыми задержками операций алгоритм пометки Сети–Ульмана создает оптимальный план для дерева выражений [321]. Его можно видоизменить с целью порождения хорошего кода для ОАГ выражения. Основываясь на алгоритме Сети–Ульмана, Фишер и Прёбстинг разработали алгоритм, который порождает оптимальные или почти оптимальные результаты для операций с памятью, имеющих небольшие задержки [299]. К сожалению, в нем возникают проблемы с ростом величины задержки или количества функциональных блоков.

В значительной части литературы по планированию рассматриваются варианты алгоритма планирования списка. Статью Ландскова и др. часто называют основополагающей работой по планированию списка [248], но алгоритм восходит по меньшей мере к работе Хеллера 1961 года [198]. Из других статей, основанных на планировании списка, упомянем работы Бернштейна и Родеха [42], Гиббонса и Мучника [169] и Хеннесси и Гросса[199]. Кришнамурти и др. предлагают высокоуровневый обзор литературы по конвейерным процессорам [243, 332]. Кернс, Ло и Эггерс разработали сбалансированное планирование как способ адаптировать планирование списка к неопределенным задержкам операций с памятью [232, 257]. В алгоритме RBF Шильке изучалось использование рандомизации и повторения как альтернативы схемам многоуровневых приоритетов [318]. Мотвани и др. использовали α-β-настройку для смешения двух эвристик: одна уменьшает спрос на регистры, другая увеличивает степень параллелизма на уровне команд [278].

Многие авторы описывали алгоритмы регионального планирования. Первым автоматизированным региональным методом, кажется, является алгоритм планирования трасс Фишера [158, 159]. Он использовался в нескольких коммерческих [148, 259] и многочисленных исследовательских системах [330]. Ху и др. предложили в качестве альтернативы планирование суперблока [211]; внутри цикла алгоритм клонирует блоки, чтобы избежать точек соединения, как показано в разделе 12.4.3.

Клик предложил алгоритм глобального планирования, основанный на использовании глобального графа ценности [90]; он использовался в оригинальном компиляторе HotSpot Server Compiler для JAVA [288]. Бала и Рубин разработали технику, способную эффективно выполнять агрессивные методы планирования, такие как перемещение кода через границы блоков и обновления ранее спланированных блоков [33]. Свое идеи они применили в глобальном планировщике для компилятора компании Kendall Square Research.

Несколько авторов предлагали методы, в которых используются специфические особенности оборудования [311, 330]. Другие подходы, в которых для улучшения планирования используется репликация, см. в работах Эбчиоглы и Накатани [147] и Гупта и Соффа [183]. Суини и Бити предложили выбирать пути на основе информации о доминировании [339]; другие изучали различные аспекты этого подхода [115, 210, 338].

Программному конвейеру посвящено много исследований. Рау и Глезер предложили эту идею в 1981 году [304]. Лэм разработал представленную здесь схему программного конвейера [245]; в этой работе описана иерархическая схема обработки потока управления внутри цикла. Айкен и Николау разработали похожий подход, получивший название идеального конвейера [11], одновременно с работой Лэма. Пример со сравнением обратного и прямого планирований на рис. 12.8 был предложен нашему вниманию Филипом Шильке [318]. Он взял его из программы тестирования производительности `go`, входящей в состав комплекта SPEC 95. Пример точно улавливает эффект, который побудил многих авторов компиляторов включать прямые и обратные планировщики в фазу синтеза.

Упражнения

К разделу 12.3

1. Некоторые операции, например копирование регистра в регистр, могут выполняться почти на любом функциональном блоке, правда, с разными кодами операций. Может ли планировщик с выгодой воспользоваться этими альтернативами? Предложите модификации базового алгоритма планирования списка, которые позволят ему использовать «синонимы» для такой простой операции, как копирование.
2. В большинстве современных процессоров имеются слоты отсрочки в некоторых или всех операциях условного перехода. При наличии одного слота отсрочки операция, следующая сразу за условным переходом, выполняется, пока обрабатывается условный переход; таким образом, идеальное место для планирования условного перехода – предпоследний такт простого блока. (У большинства процессоров имеется версия условного перехода, которая не выполняет слот отсрочки, так что компилятор может избежать генерирования команды `nop` в незаполненном слоте отсрочки.)
   a. Как бы вы изменили алгоритм планирования списка, чтобы улучшить его способность «заполнять» слоты отсрочки?
   b. Предложите набросок прохода после планирования, на котором заполнялись бы слоты отсрочки.
   c. Предложите остроумное использование слотов отсрочки условных переходов, которые не могут быть заполнены полезными операциями.
3. Обслуживание списков *Ready* и *Active* – основная часть алгоритма планирования списка (см. рис. 12.6). На каждом такте планировщик должен определять, какие операции в *Active* завершились, а затем проверять каждого преемника только что завершившейся операции на предмет готовности его операндов.
   a. Проверка для каждого преемника *s*, все ли используемые им операнды доступны, – самая дорогая часть планирования списка. Предложите схему уменьшения асимптотической стоимости этих проверок.

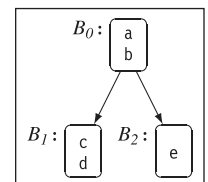
b. Чтобы избежать просмотра всего списка *Active*, автор компилятора мог бы реализовать *Active* с помощью отдельного списка для каждого такта. Сколько понадобилось бы списков? Как такая схема отразилась бы на стоимости планирования?

c. Хорошая реализация схемы с несколькими списками из части «b» могла бы использовать список повторно. Предложите набросок кода, необходимого для повторного использования списков.

К разделу 12.4

4. При суперлокальном планировании какие структуры данных должен был бы сохранять планировщик в конце блока с несколькими преемниками? Предложите эффективный способ их сохранения.

5. Рассмотрим фрагмент ГПУ, показанный справа.



Фрагмент ГПУ для упражнения 5

Предположим, что планировщик решил переместить операцию c из B_1 в B_0, поставив ее между a и b. (Если это перемещение безопасно, то c не изменяет значений, используемых в b.) Планировщик должен гарантировать, что c также не влияет на e.

1. Если c переопределяет значение, используемое в e, то как планировщик мог бы воспользоваться переименованием, чтобы избежать влияния на e? Какие определения и использования надлежит переименовать?
2. Если значение снова используется в каком-то блоке, следующем за B_1, то как планировщик мог бы гарантировать, что эти использования получают правильное значение? Сформулируйте свой ответ в терминах множеств LiveOut для B_1 и B_2.

6. Снова обращаясь к ГПУ во врезке, предположим, что планировщик решил переместить a в блок B_2. При каких условиях он может избежать вставки копии a в B_1?

7. Используя ГПУ на рис. 12.18:

a. Перечислите расширенные простые блоки в ГПУ, а также все пути через каждый РПБ. В каком порядке компилятор должен планировать эти пути?

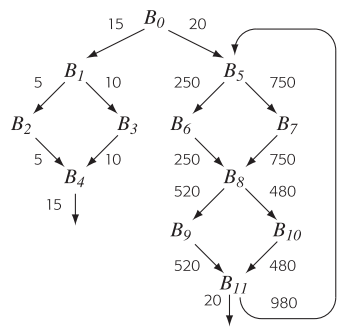


Рис. 12.18 ГПУ к упражнению 7

b. Перечислите трассы, которые алгоритм построения трасс найдет в ГПУ, в том порядке, в каком они должны быть спланированы.

c. Покажите ГПУ, который получился бы в результате применения клонирования суперблока к ГПУ на рис. 12.18. Пометьте вершины оценками времени выполнения в тактах.

К разделу 12.5

8. Программный конвейер перекрывает выполнение итераций цикла, чтобы создать эффект, напоминающий аппаратный конвейер.

a. Обсудите, какое влияние этот программный конвейер окажет на спрос на регистры в конвейеризованном цикле (по сравнению с оригинальным).

b. Как планировщик может воспользоваться предикатным выполнением, чтобы уменьшить плату за программный конвейер в терминах размера кода? (Предикатное выполнение обсуждается в разделе 7.4.2 на стр. 371.)

9. Пример кода на рис. 12.11 генерирует пятитактовое ядро программного конвейера, потому что он содержит девять операций. Если бы компилятор генерировал адреса `x`, `y` и `z` по-другому, как показано на рис. 12.19, то он мог бы еще уменьшить число операций в теле цикла.

Такт	Функциональный блок 0		Комментарии
–5	addI	r_{arp},@x ⇒ $r_{@x}$	Инициализировать цикл
–4	addI	r_{arp},@y ⇒ $r_{@y}$	начальными загрузками
–3	addI	r_{arp},@z ⇒ $r_{@z}$	
–2	loadI	0 ⇒ r_{ctr}	
–1	loadI	792 ⇒ r_{ub}	
1	L_1: loadAO	r_{ctr},$r_{@x}$ ⇒ r_x	Получить x[i] & y[i]
2	loadAO	r_{ctr},$r_{@y}$ ⇒ r_y	
3	mult	r_x,r_y ⇒ r_z	Приостановить до готовности r_x и r_y
6	cmp_LE	r_{ctr},r_{ub} ⇒ r_{cc}	Тень mult
7	storeAO	r_z ⇒ r_{ctr},$r_{@z}$	Сохранить результат
8	addI	r_{ctr},4 ⇒ r_{ctr}	Увеличить указатель смещения
9	cbr	r_{cc} → L_1, L_2	Переход, замыкающий цикл
	L_2: ...		

Рис. 12.19 Код к упражнению 9

В этой схеме используется на один регистр больше, чем в оригинальном коде.

a. Вычислите *RC* и *DC* для этой версии цикла.

b. Сгенерируйте программно конвейеризованное тело цикла.

c. Сгенерируйте код пролога и эпилога для вашего конвейеризованного тела цикла.

Глава 13

Распределение регистров

АННОТАЦИЯ

Код, сгенерированный компилятором, должен эффективно использовать ограниченные ресурсы целевого процессора. Одним из самых ограниченных ресурсов являются аппаратные регистры. Регистры играют огромную роль для производительности откомпилированного кода. В то же время распределение регистров – процесс определения того, какие значения хранить в регистрах, – комбинаторно трудная задача.

Большинство компиляторов отделяют решение о распределении регистров от других решений, связанных с оптимизацией, поэтому включают отдельный проход для этой цели. Мы начнем эту главу с локального распределения регистров, чтобы познакомиться с проблемой и терминологией. В основной же части главы речь пойдет о глобальном распределении регистров и их назначении с помощью раскрашивания графа. В разделе «Дополнительные темы» обсуждаются некоторые из многочисленных вариантов этого метода, описанные в научных статьях и применяемые на практике.

КЛЮЧЕВЫЕ СЛОВА

Распределение регистров, выгрузка регистров, консолидация операций копирования, распределители на основе раскрашивания графов

13.1. Введение

Регистры – важная часть архитектуры большинства процессоров. Поскольку процесс обращается к регистрам быстрее, чем к памяти, они играют существенную роль в производительности откомпилированного кода. Распределение регистров – настолько трудная задача, что большинство компиляторов реализуют ее в виде отдельного прохода, выполняемого до или после планирования команд.

Распределитель регистров определяет, какие значения будут находиться в регистрах в каждой точке кода, а какие будут храниться в памяти. После того как это решение принято, распределитель должен переписать код, что-

бы привести его в действие; обычно это сводится к добавлению операций загрузки и сохранения, которые перемещают значения между памятью и регистрами. Распределитель может переместить значение в память, потому что количество живых значений в коде больше, чем регистров в целевой машине. Или же распределитель может хранить значение в памяти между использованиями, потому что не может доказать, что оставить его в регистре безопасно.

Концептуальная дорожная карта

Распределитель регистров принимает на входе программу, в которой используется некое произвольное число регистров. Он преобразует код в эквивалентную программу с учетом конечности набора регистров целевой машины. В каждой точке кода он решает, какие значения будут находиться в регистрах, а какие в памяти. Вообще говоря, доступ к данным в регистрах быстрее, чем обращение к памяти.

Для преобразования кода, так чтобы он не выходил за пределы набора регистров целевой машины, распределитель вставляет операции, которые перемещают значения между регистрами и памятью. Добавленные операции, или «код выгрузки», включают загрузку, сохранение и вычисления адресов. Распределитель стремится минимизировать затраты на выгрузку и сохранение на этапе выполнения.

Выгрузка
Когда распределитель перемещает значение из регистра в память, говорят, что он *выгружает* (spill) значение.

Восстановление
Когда распределитель помещает в регистр ранее выгруженное значение, говорят, что он *восстанавливает* (restore) значение.

Трудность заключается в том, что распределение регистров – комбинаторно трудная задача. Проблемы, с которыми приходится сталкиваться при распределении и назначении, в своей самой общей форме являются NP-полными. Следовательно, распределитель не может гарантировать нахождение оптимального решения за разумное время. Хороший распределитель регистров работает быстро – за время от $\mathbf{O}(n)$ до $\mathbf{O}(n^2)$, где n – размер входной программы. Стало быть, хороший распределитель находит эффективное приближенное решение трудной задачи и делает это быстро.

Несколько слов о времени

Распределитель регистров работает на этапе компиляции и преобразует почти готовую программу из пространства имен ПП в фактические регистры и память целевой ISA. За распределением регистров может следовать проход планирования или финальная оптимизация, например проход оконной оптимизации.

Распределитель порождает код, работающий на этапе выполнения. Поэтому, когда распределитель рассуждает о стоимости различных решений, он на этапе компиляции делает оценку ожидаемого изменения времени выполнения окончательного кода. По необходимости такие оценки являются приближенными.

Несколько систем компиляции включали переносимые распределители регистров, управляемые описаниями. Чтобы переконфигурировать такую систему, автор компилятора создает описание целевой машины на этапе проектирования, а затем на этапе сборки инструменты строят рабочий распределитель.

Краткий обзор

Виртуальный регистр
символическое имя регистра, которое компилятор использует до распределения регистров.

Мы обозначаем виртуальные регистры VR_i, $i \geqslant 0$.

Физический регистр
фактический регистр целевого процессора.

Мы обозначаем физические регистры PR_i, $i \geqslant 0$.

Чтобы упростить ранние этапы трансляции, многие компиляторы используют ПП, в котором пространство имен не привязано ни к адресному пространству целевого процессора, ни к его набору регистров. Для трансляции ПП-кода в ассемблерный код целевой машины эти имена следует отобразить на аппаратные ресурсы целевой машины. Значения, хранящиеся в памяти ПП программы, должны быть преобразованы в машинные адреса с помощью методов, описанных в разделе 7.3. Значения, хранящиеся в виртуальных регистрах (VR), должны быть преобразованы в физические регистры (PR) процессора.

Модель памяти ПП программы в значительной степени определяет роль распределителя регистров (см. раздел 4.7.1).

- В случае модели памяти регистр-регистр ПП может использовать столько виртуальных регистров, сколько необходимо, не учитывая размер набора физических регистров. Позже распределитель регистров должен отобразить множество VR на множество PR и вставить код для перемещения значений между PR и памятью.
- В случае модели память-память ПП хранит все значения в памяти, за исключением ближайшей окрестности операции, в которой значение определяется или используется. Распределитель регистров может назначить значению регистр на протяжении части времени его жизни, чтобы повысить производительность.

Таким образом, в модели регистр-регистр входной код может быть представлен не в той форме, в какой он будет выполняться на целевом компьютере. Распределитель регистров преобразует этот код в пространство имен и форму, позволяющие выполнить его на целевой машине. При этом распределитель стремится минимизировать стоимость нового вставленного им кода. С другой стороны, в модели память-память любые перемещения между регистрами и памятью явные; код мог бы исполняться на целевой машине. В этой ситуации роль распределителя сводится к оптимизации с целью удержать некоторые значения в регистрах на время большее, чем во входном коде.

Эта глава посвящена в основном глобальному распределению регистров в компиляторе с моделью памяти регистр-регистр. В разделе 13.3 обсужда-

ются проблемы, возникающие в одноблочном распределителе; такой локальный распределитель поможет нам обосновать потребность в глобальном распределении. В разделе 13.4 изучается глобальное распределение регистров с помощью раскрашивания графов. Наконец, в разделе 13.5 мы рассмотрим варианты схемы глобального раскрашивания, которые обсуждались в литературе и были испытаны в практических компиляторах.

Раскрашивание графа
такое назначение цветов вершинам графа, при котором вершины n_1 и n_2 раскрашены в разные цвета, если в графе существует ребро (n_1, n_2).

13.2. Предварительные сведения

Проектирование и реализация распределителя регистров – сложная задача. Решения, принимаемые на этапе проектирования распределителя, могут зависеть от принятых ранее проектных решений. В то же время проектные решения о распределителе могут повлиять на решения, принимаемые на более ранних проходах компилятора. В этом разделе кратко описываются некоторые проблемы, возникающие при проектировании распределителя.

13.2.1. Пространство имен для распределения: интервалы жизни переменных

По существу, распределитель регистров создает новое пространство имен в коде. Он получает код, написанный в терминах VR и адресов памяти, а выводит код, в котором VR отображены на физические регистры и некоторые дополнительные адреса памяти.

Чтобы повысить эффективность сгенерированного кода, распределитель должен минимизировать лишние перемещения данных как между регистрами и памятью, так и между регистрами. Если распределитель решает хранить некоторое значение *x* в физическом регистре, то должен сделать так, чтобы все определения *x* ссылались на один и тот же PR, а при любом использовании *x* читался именно этот PR – если такое вообще возможно. Если эту цель удастся достичь, то ненужные операции копирования регистра в регистр будут устранены; возможно, также будут устранены некоторые операции сохранения и загрузки.

Большинство распределителей регистров строят новое пространство имен: пространство имен *интервалов жизни* (live range). Каждому интервалу жизни (LR) соответствует одно значение в пределах области, исследуемой распределителем. Распределитель анализирует поток значений в коде, строит его множество LR

Интервалы жизни
замкнутое множество взаимосвязанных определений и использований.

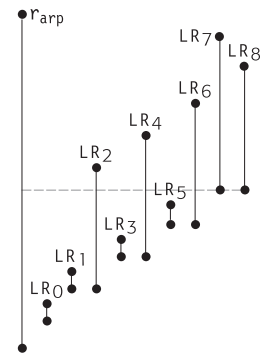
В большинстве распределителей интервалы жизни используются как значения, для которых они решают вопрос о размещении – в физическом регистре или в памяти.

и производит переименования с целью использовать имена LR. Детали того, что именно составляет LR, зависят от области и алгоритма распределения.

В одном блоке LR легко визуализировать и понять. Каждый LR соответствует одному определению и простирается от этого определения до последнего использования значения. На рис. 13.1(a) показан фрагмент ILOC, впервые встретившийся в главе 1, а на рис. 13.1(b) – код, переименованный с использованием интервалов жизни. Сами интервалы жизни показаны в виде графика на рис. 13.1(c). График можно описать как множество интервалов; например, LR_2 – это [6, 9], LR_8 – [3, 6]. На рисунке (c) предполагается отсутствие совмещения при выполнении операций.

1	loadAI	r_{arp}, @a ⇒ r_a	loadAI	r_{arp},@a	⇒ LR_7
2	loadI	2 ⇒ r_2	loadI	2	⇒ LR_8
3	loadAI	r_{arp}, @b ⇒ r_b	loadAI	r_{arp},@b	⇒ LR_6
4	loadAI	r_{arp}, @c ⇒ r_c	loadAI	r_{arp},@c	⇒ LR_4
5	loadAI	r_{arp}, @d ⇒ r_d	loadAI	r_{arp},@d	⇒ LR_2
6	mult	r_a, r_2 ⇒ r_a	mult	LR_7,LR_8	⇒ LR_5
7	mult	r_a, r_b ⇒ r_a	mult	LR_5,LR_6	⇒ LR_3
8	mult	r_a, r_c ⇒ r_a	mult	LR_3,LR_4	⇒ LR_1
9	mult	r_a, r_d ⇒ r_a	mult	LR_1,LR_2	⇒ LR_0
10	storeAI	r_a ⇒ r_{arp}, @a	storeAI	LR_0	⇒ r_{arp},@a

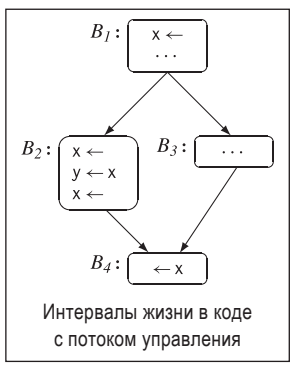
(a) Пример из раздела 1.3.3 (b) После переименования в LR



(c) Протяженности интервалов жизни

Рис. 13.1 Интервалы жизни в простом блоке

Мы обозначаем LR_8 как начинающийся в операции 3, потому что эта операция читает свои аргументы в начале выполнения и записывает результат в конце выполнения. Таким образом, LR_8 фактически определен в конце второй операции. По соглашению, мы обозначаем интервал жизни как начинающийся первой операцией *после* его определения. При таком подходе становится ясно, что два экземпляра r_1 в addI r_1, 10 ⇒ r_1 – это не один и тот же интервал жизни, если только другой контекст не делает их таковыми.



Интервалы жизни в коде с потоком управления

LR заканчивается по одной из двух причин. Операция может быть последним использованием имени на текущем пути в коде. Или же код может переопределить имя перед следующим использованием, чтобы начать новый LR.

В ГПУ с потоком управления ситуация сложнее, как показано во врезке. Рассмотрим переменную *x*. Три ее определения образуют два отдельных и непересекающихся интервала жизни.

1. Использование в B_4 относится к двум определениям: в B_1 в конце B_2. Эти три события образуют первый LR, который мы обозначим LR_1, он охватывает B_1, B_3, B_4 и последнее предложение в B_2.

2. Использование x в B_2 относится только к определению, предшествующему ему в B_2. Эта пара образует второй LR, обозначенный LR_2. LR_1 и LR_2 не зависят друг от друга.

При более сложном потоке управления интервалы жизни могут иметь более запутанные формы. В глобальном распределителе из раздела 13.4 LR состоит из всех определений, достигающих некоторого использования, плюс всех использований, достигаемых этими определениями. Эта совокупность определений и использований образует замкнутое множество. Нотация интервалов, хорошо показавшая себя в случае одного блока, не улавливает сложности этой ситуации.

Вариации на тему интервалов жизни

Интервалы жизни настолько протяженные, насколько возможно при заданной точности анализа. При наличии более точной информации о неоднозначности некоторые интервалы жизни могут удлиниться.

В различных алгоритмах распределения *интервалы жизни* определяются по-разному. Локальный распределитель, описанный в разделе 13.3, рассматривает все время жизни значения в простом блоке как один интервал жизни; он использует интервал жизни максимальной длины в блоке. Глобальный распределитель, описанный в разделе 13.4, также использует интервал жизни максимальной длины, но во всей процедуре.

В других распределителях используются другие определения интервала жизни. Распределители с линейным сканированием используют аппроксимацию интервалов жизни, которая завышает их длину, но порождает интервальное представление, которое дает более быстрый распределитель. Распределители на основе SSA трактуют каждое SSA-имя как отдельный интервал жизни; после распределения они должны выполнить трансляцию из формы SSA. Несколько распределителей ограничивали длину интервалов жизни в соответствии с особенностями графа потока управления, например циклами, чтобы лучше управлять размещением кода выгрузки. В разделе 13.5.3 описываются некоторые из таких альтернативных формулировок.

Форма кода и интервалы жизни

В этом обсуждении переменная является *скалярной*, если это одно значение, которое может поместиться в регистр.

Распределитель регистров должен понимать, когда переменную исходного кода допустимо размещать в регистре. Если переменная неоднозначная, то она может размещаться в регистре только между моментом создания и следующей операцией сохранения в коде (см. раздел 4.7.2). Если она однозначная и скалярная, то распределитель может рассмотреть возможность ее хранения в регистре в течение более длительного срока.

У компилятора есть два основных способа определить, когда значение однозначно. Он может выполнить статический анализ и установить, какие

значения безопасно хранить в регистре; анализ может быть либо недорогим и неточным, либо дорогим и точным. Или же знания о неоднозначности можно закодировать в форме кода.

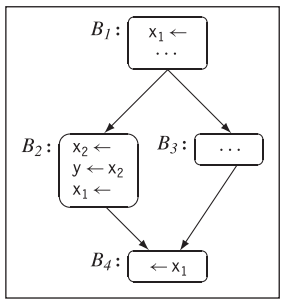
Если компилятор пользуется моделью памяти регистр-регистр, то может выделить VR каждому однозначному значению. Если VR жив после возврата из процедуры, в которой определен, как в случае статического значения или параметра, переданного по ссылке, то ему понадобится также место постоянного хранения в памяти. Компилятор может сохранять VR в нужных местах кода.

Если в ПП используется модель память-память, то распределитель все равно может извлечь выгоду из знаний о неоднозначности. Компилятор должен сохранять эту информацию в записях таблицы символов для каждого значения.

13.2.2. Интерференция

Интерференция
Два интервала жизни LR_1 и LR_2 *интерферируют*, если существует операция, где они оба живы, и компилятор не может доказать, что они имеют одно и то же значение.

Самая главная функция распределителя регистров – определить для двух интервалов жизни, могут ли они занимать один и тот же регистр. Два LR могут разделять один регистр, если используют один и тот же класс регистров и не являются живыми одновременно. Если два LR используют один и тот же класс регистров, но существует операция, где оба они живы, то эти LR не могут занимать один и тот же регистр, если только компилятор не может доказать, что их значения одинаковы. О таких интервалах жизни говорят, что они *интерферируют*.



Два LR, которые используют физически различные классы регистров, не могут интерферировать, потому что не конкурируют за один и тот же ресурс. Так, например, LR с плавающей точкой не может интерферировать с целочисленным LR на процессоре, который пользуется различными регистрами для этих двух видов значений.

В примере ГПУ во врезке оба LR для x не интерферируют; x_2 жива только внутри B_2 – участка кода, на котором x_1 мертва. Следовательно, решения о выделении регистров для x_1 и x_2 независимы. Они могли бы разделять один PR, но нет никакой причины, заставляющей распределитель сделать такой выбор.

Граф интерференции
граф $G = (N, E)$, имеющий вершину n для каждого LR, в котором ребро (LR_i, LR_j) существует тогда и только тогда, когда LR_i и LR_j интерферируют.

Принцип работы глобальных распределителей заключается в нахождении интерференций и использовании их для направления процесса распределения. Распределитель, описанный в разделе 13.4, строит конкретное представление этих

конфликтов, *граф интерференции*, и затем раскрашивает этот граф, чтобы отобразить интервалы жизни на PR. Эта парадигма лежит в основе многих глобальных распределителей; меняются только точность графа и конкретный алгоритм раскрашивания.

Нахождение интерференций

Для нахождения интерференций компилятор сначала вычисляет информацию о живучести для кода. Затем он посещает каждую операцию в коде и добавляет интерференции. Если операция определяет LR_i, то распределитель добавляет интерференцию к каждому LR_j, который жив на этой операции.

Единственное исключение из этого правила – операция копирования $LR_i \leftarrow LR_j$, которая устанавливает значение LR_i равным значению LR_j. Поскольку LR-источник и LR-приемник имеют одно и то же значение, операция не создает интерференцию между ними. Если LR_i и LR_j не интерферируют никаким другим способом, то они могли бы разделять общий PR.

Интерференция и спрос на регистры

Граф интерференции дает быстрый способ оценить *спрос на регистры*. Для вершины графа LR_i *степенью* LR_i, обозначаемой LR_i°, называется число соседей LR_i в графе. Если все соседи LR_i живы на одной и той же операции, то для размещения всех этих значений в регистрах понадобилось бы $LR_i^\circ + 1$ регистров. Если эти значения не являются живыми на одной и той же операции, то спрос на регистры может быть меньше степени. Максимальная по всем вершинам степень в графе интерференции дает грубую верхнюю границу количества регистров, потребного для выполнения программы без выгрузки.

Представление физических регистров

Часто распределитель включает в граф интерференции вершины для представления PR. Эти вершины позволяют компилятору определять как связи, так и интерференции с PR. Например, если код передает LR_i процедуре в качестве второго параметра, то компилятор мог бы запомнить этот факт с помощью копирования из LR_i в PR, который будет содержать второй параметр.

Некоторые компиляторы используют PR для управления назначением LR. Чтобы принудительно сопоставить интервалу LR_i регистр PR_j, компилятор может добавить псевдоинтерференцию из LR_i в каждый PR, кроме PR_j. Аналогично, чтобы помешать LR_i использовать PR_j, компилятор может добавить интерференцию между LR_i и PR_j. Этот механизм работает, но может стать громоздким. Механизм обработки перекрывающихся классов регистров, представленный в разделе 13.4.7, дает более общий и элегантный способ управления размещением в конкретном PR.

Псевдоинтерференция
Добавляя ребро между LR_i и PR_j, компилятор должен понимать, что это ребро не вносит фактического вклада в спрос на регистры.

13.2.3. Код выгрузки

Решив, что не может дальше хранить какой-то LR в регистре, распределитель должен выгрузить этот LR в память, перед тем перераспределить PR под другое значение. Он также должен восстановить выгруженный LR в PR перед последующим использованием. Эти добавленные операции выгрузки и восстановления увеличивают время выполнения, поэтому распределитель должен вставлять их как можно реже. Самая прямая мера качества распределения – время, потраченное на выгрузку регистров.

Распределители различаются гранулярностью выгрузки значений. Глобальный распределитель, описанный в разделе 13.4, выгружает весь интервал жизни. Решив выгрузить LR_i, он вставляет операцию выгрузки после каждого определения в LR_i и операцию восстановления сразу перед каждым использованием LR_i. По сути дела, это разбивает LR_i на множество крохотных LR, по одному в каждом определении и в каждом использовании.

С другой стороны, локальный распределитель, описанный в разделе 13.3, выгружает интервал жизни только между точкой, в которой его PR перераспределен, и его следующим использованием. Поскольку этот распределитель работает в одном блоке с линейным потоком управления, реализовать такую политику ему легко; для LR имеется всего одно следующее использование, и точка выгрузки всегда предшествует этому использованию.

Между двумя крайностями – «выгружать везде» и «выгружать однократно» – лежит много вариантов. Исследователи экспериментировали с выгрузкой частичных интервалов жизни. Проблема выбора оптимальной гранулярности выгрузки, несомненно, так же трудна, как нахождение оптимального распределения; правильная гранулярность, вероятно, различна для разных интервалов жизни. В разделе 13.5 описываются некоторые схемы, опробованные авторами компиляторов и исследователями.

Неравномерная стоимость выгрузки

LR мог бы быть чистым вследствие предшествующей выгрузки на текущем пути или потому, что его значение существует также в памяти.

Выгрузку осложняет еще и то, что распределитель должен учитывать тот факт, что свойства LR могут изменять стоимость его выгрузки и восстановления.

Грязное значение. В общем случае LR содержит значение, которое уже было вычислено, но еще не было сохранено в памяти; говорят, что LR *грязный*. Грязный LR требует сохранения в точках выгрузки и загрузки в точках восстановления.

Чистое значение. Если значение LR уже существует в памяти, то выгрузка не требует сохранения; говорят, что LR *чистый*. Выгрузка чистого LR не стоит ничего; стоимость его восстановления такая же, как для грязного значения.

Пересчитываемое значение. Некоторые LR содержат значения, которые дешевле вычислить заново, чем выгружать и восстанавливать. Если значе-

ния, используемые для вычисления значения LR, доступны в каждой точке использования LR, этот распределитель может просто пересчитать значение LR по запросу. Классический пример дает LR, определяемый непосредственной загрузкой. Выгрузка такого LR не стоит ничего, а чтобы восстановить его, компилятор вставляет код повторного вычисления. Обычно операция непосредственной загрузки дешевле, чем загрузка из памяти.

Распределитель должен по возможности учитывать неравномерную природу стоимости выгрузки. Конечно, это усложняет распределитель. К тому же NP-полнота задачи распределения означает, что не существует простой эвристики, позволяющей сделать наилучший выбор в каждой ситуации.

Места выгрузки

Место выгрузки
адрес памяти, ассоциированный с LR, в котором хранится значение последнего, когда LR не назначен никакой PR.

Выгружая грязное значение, распределитель должен разместить его где-то в памяти. Если LR точно соответствует переменной, хранящейся в памяти, то распределитель мог бы выгрузить значение в это место. В противном случае ему придется зарезервировать в памяти место для хранения значения на время, пока оно живо, но не находится в PR.

Отметим, что любое значение в месте выгрузки однозначно, – это важно для планирования, следующего за распределением.

Большинство распределителей помещают места выгрузки в конце области локальных данных процедуры. Эта стратегия позволяет коду выгрузки и восстановления обращаться к значению с фиксированным смещением от УЗА, используя операцию с непосредственной адресацией памяти, если в ISA такая имеется. Распределитель просто увеличивает размер области локальных данных на этапе компиляции, так что распределение не влечет прямых затрат на этапе выполнения.

Поскольку LR представляет интерес только во время выполнения той части кода, где он жив, у распределителя имеется возможность уменьшить объем памяти, используемой под выгрузку. Если LR_i и LR_j не интерферируют, то могут разделить общее место выгрузки. Таким образом, распределитель может использовать граф интерференции также для раскрашивания мест выгрузки и уменьшения расхода памяти под выгрузку.

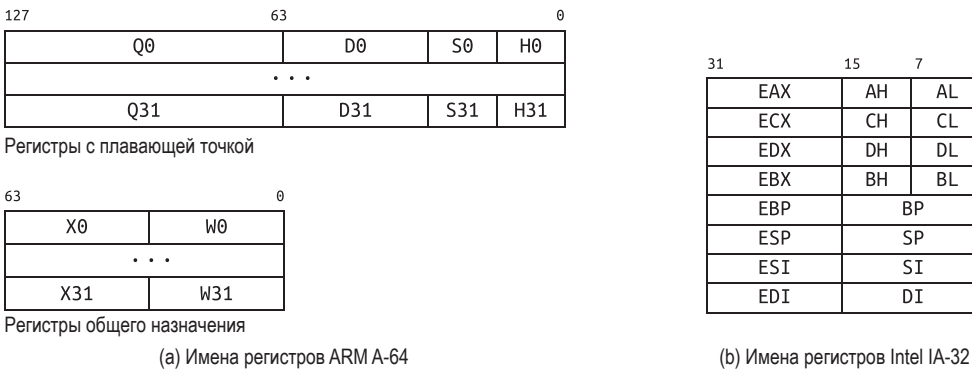
13.2.4. Классы регистров

Класс регистров
группа именованных регистров с общими свойствами, например длиной и набором поддерживаемых операций.

Многие процессоры поддерживают несколько классов регистров. Например, в большинстве ISA имеет один набор регистров общего назначения (РОН) для использования в операциях с целыми числами и адресами и другой набор регистров с плавающей точкой (РПТ). В случае РОН и РПТ

оба класса регистров почти всегда реализуются в виде физически и логически раздельных наборов.

Часто ISA совмещает несколько классов регистров в одном наборе физических регистров. Как показано на рис. 13.2(a), ARM A-64 поддерживает четыре размера значений с плавающей точкой, в одном наборе РПТ с четверной точностью (128 разрядов). 128-разрядные РПТ называются `Q0`, `Q1`, ..., `Q31`. Каждый `Qi` совмещается с 64-разрядным регистром `Di`, 32-разрядным регистром `Si` и 16-разрядным регистром `Hi`. Более короткие регистры занимают младшие разряды более длинных.



(a) Имена регистров ARM A-64

(b) Имена регистров Intel IA-32

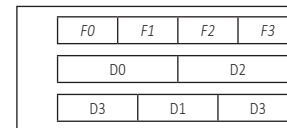
Рис. 13.2 Совмещенные имена регистров

РОН в ARM A-64 устроены так же. 64-разрядные РОН имеют как 64-разрядные имена `Xi`, так и 32-разрядные имена `Wi`. И на этот раз 32-разрядный регистр занимает младшие разряды 64-разрядного.

> Обсуждение ограничено подмножеством набора регистров IA-32. Сегментные регистры и большинство регистров, добавленных в IA-64, игнорируются.

Intel IA-32 располагает небольшим набором регистров, изображенным на рис. 13.2(b). В нем восемь 32-разрядных регистров. В системе команд CISC-машины разные регистры используются для конкретных целей, поэтому у регистров уникальные имена. Ради обратной совместимости с 16-разрядными процессорами набор PR поддерживает схему именования подполей 32-разрядных регистров.

- В четырех регистрах программист может поименовать 32-разрядный регистр, его младшие 16 разрядов и два 8-разрядных регистра. Это аккумулятор (`EAX`), регистр счетчика (`ECX`), регистр данных (`EDX`) и регистр базы (`EBX`).
- В остальных четырех регистрах программист может поименовать 32-разрядный регистр и его младшие 16-разрядов. Это база стека (`EBP`), указатель стека (`ESP`), начальный индекс строки (`ESI`) и конечный индекс строки (`EDI`).



Пары регистров

Показано, что D3 циклически оборачивается.

На рисунке не показаны регистры счетчика команд (`EIP` и `IP`) и регистр флагов (`EFLAGS` и `FLAGS`), которые тоже имеют 32- и 16-разрядные имена. В архитектуре IA-64 имеются дополнительные 32-разрядные РОН, но сохранены имена и функциональность регистров IA-32.

Во многих более ранних ISA использовались схемы спаривания в наборе РОН. Во врезке показано, как мог бы работать набор четырех регистров. Он состоял бы из четырех 32-разрядных PR: `F0`, `F1`, `F2` и `F3`. 64-разрядные значения занимали бы пару соседних регистров. Если пара регистров может начинаться любым регистром, то возможны четыре пары: `D0`, `D1`, `D2` и `D3`.

Некоторые ISA вводят ограничение, требуя, чтобы пара начиналась нечетным регистром, – такая пара называется выровненной. В этом случае были бы доступны только регистры `D0` и `D2`. В случае выровненных пар использование `D0` делает невозможным применение `F0` и `F1`. В случае невыровненных пар использование `D0` все равно препятствует применению `F0` и `F1`, а также `D1` и `D3`.

В общем случае распределитель регистров должен эффективно использовать все доступные регистры. Таким образом, он должен понимать процессорные классы регистров и включать механизмы их справедливого и эффективного использования. Для физически непересекающихся классов, таких как регистры с плавающей точкой и регистры общего назначения, распределитель может просто выделять их независимо. Если для выгрузки регистров с плавающей точкой РОН нужны вычисления адресов, то компилятор должен сначала распределить РПТ.

Дизайн пространства имен для набора регистров влияет на трудность управления классами регистров в распределителе. Например, схема именования в ARM A-64 позволяет распределителю рассматривать все поля одного PR как единый ресурс; он может использовать либо `X0`, либо `W0`. Напротив, IA-32 позволяет одновременно использовать `AH` и `AL`. Таким образом, для работы с набором регистров IA-32 распределителю нужны более сложные механизмы. В разделе 13.4.7 рассматривается, как встроить такие механизмы в распределитель регистров на основе раскрашивания графа.

ИТОГИ РАЗДЕЛА

Распределитель регистров должен в каждой точке кода принять решение о том, какие значения хранить в регистрах. С этой целью он вычисляет пространство имен для значений в коде – интервалы жизни. Распределитель должен понять, какие интервалы жизни не могут разделять общий регистр, т. е. какие интервалы интерферируют между собой. Наконец, он должен назначить каким-то интервалам жизни регистры, а какие-то оставить в памяти. Чтобы поддержать свои решения, распределитель должен вставлять в нужные места операции загрузки и сохранения для перемещения значений между регистрами и памятью.

ВОПРОСЫ

1. Рассмотрите блок линейного кода, в котором наибольший спрос на регистры равен *j*. Предположим, что распределителю разрешено использовать *k* регистров. Если $j = k$, то может ли распределитель отобразить интервалы жизни на PR без выгрузки?
2. Рассмотрите процедуру, представленную *n* ILOC-операциями. Можете ли вы ограничить сверху количество вершин и ребер графа интерференции?

13.3. Локальное распределение регистров

Напомним, что *простым блоком* называется линейная последовательность кода максимальной длины.

Простейшая постановка задачи распределения регистров – локальное распределение: рассматривается один простой блок и один класс, состоящий из *k* физических регистров. Эта проблема дает представление о многих сложностях распределения и послужит нам полезным введением в концепции и терминологию, которые понадобятся при обсуждении глобального распределения. Чтобы упростить обсуждение, мы будем предполагать, что вся программа состоит из одного блока.

Во входном коде используются *исходные регистры*, записываемые как sr_i.

В выходном коде используются *физические регистры*, записываемые как pr_i или просто r_i.

В общем случае физическим регистрам соответствуют именованные регистры целевой ISA.

Входной блок содержит последовательность трехадресных операций, каждая из которых имеет вид $op_i\ sr_i\ ,\ sr_j \Rightarrow sr_m$. На верхнем уровне можно сказать, что локальный распределитель регистров переписывает блок, заменяя каждую ссылку на исходный регистр (SR) ссылкой на конкретный физический регистр (PR). Распределитель должен сохранить оригинальную семантику входного блока, адаптируя вычисление к *k* PR, предоставляемым целевой машиной.

Если в какой-то точке блока в вычислении оказывается больше *k* живых значений, т. е. значений, которые могут быть использованы в будущем, то некоторые из них придется на протяжении некоторого отрезка их жизни разместить в памяти (*k* регистров могут содержать не более *k* значений). Следовательно, распределитель должен вставить в блок код, который перемещает значения между памятью и регистрами и гарантирует, что все значения окажутся в PR, когда в них возникнет необходимость, и что ни в какой момент не требуется более *k* PR.

В этом разделе описывается вариант алгоритма Бэста, который восходит еще к оригинальному компилятору FORTRAN. Это один из самых сильных известных алгоритмов локального распределения. Он совершает два прохода по коду. На первом проходе собираются подробные сведения об определениях и использованиях значений; по сути дела, он вычисляет

множество Live в блоке. Затем на втором проходе выполняется собственно распределение.

Выгрузка
Когда распределитель перемещает живое значение из PR в память, говорят, что он *выгружает* значение.

Восстановление
Когда распределитель извлекает ранее выгруженное значение из памяти, говорят, что он *восстанавливает* значение.

Алгоритм Бэста руководствуется одним принципом: когда распределителю нужен PR, а все они заняты, он должен выгрузить PR, содержащий значение, следующее использование которого отстоит дальше других в будущем. Интуитивно это понятно; алгоритм выбирает PR, который уменьшит спрос на регистры на самом протяженном интервале. Если стоимость выгрузки и восстановления всех значений одинакова, то такой выбор оптимален. На практике эти допущения оправдываются редко, но тем не менее алгоритм Бэста работает отлично.

Объяснить алгоритм проще на примере конкретной структуры данных. Предположим, что имеется трехадресный, ILOC-подобный код, представленный в виде списка операций. Каждая операция, например `mult` sr_1, $sr_2 \Rightarrow sr_3$, представлена структурой:

	ОПЕРАНД 1				ОПЕРАНД 2				ОПЕРАНД 3			
Opcode	SR	VR	PR	NU	SR	VR	PR	NU	SR	VR	PR	NU
mult	r_1	-	-	∞	r_2	-	-	∞	r_3	-	-	∞

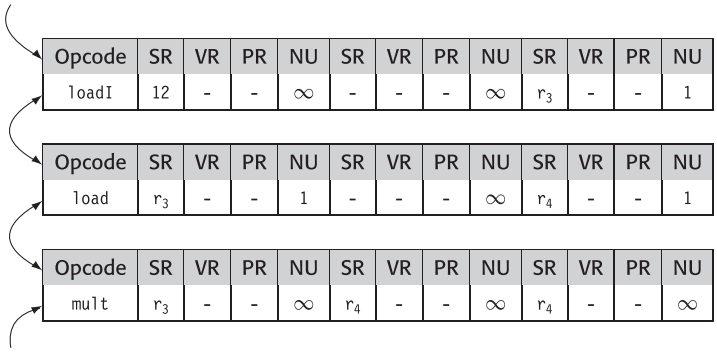
У операции имеется код (opcode), два входа (операнды 1 и 2) и результат (операнд 3). У каждого операнда имеется имя исходного регистра (SR), имя виртуального регистра (VR), имя физического регистра (PR) и индекс его следующего использования (NU).

Распределение регистров по сути своей является процессом построения нового пространства имен и модификации кода с целью использования этого пространства. Разделение имен SR, VR и PR упрощает написание и отладку распределителя.

Множество операций можно было бы представить в виде дважды связанного списка, как показано на рис. 13.3. Локальный распределитель должен будет обходить этот список в обоих направлениях. Список можно было бы создать в массиве структур или из индивидуально или блочно выделенных структур.

Поскольку семантика ясна, мы храним константу для операции `loadI` в поле `SR` ее первого операнда.

У первой операции, `loadI`, первым аргументом является непосредственное значение, хранящееся в поле SR. Второго аргумента у нее нет. Следующая операция, `load`, также имеет всего один аргумент. Последняя операция, `mult`, имеет два аргумента. Поскольку фрагмент кода не содержит следующего использования никакого из регистров, упомянутых в операции mult, ее поле NU равно ∞.



Opcode	SR	VR	PR	NU	SR	VR	PR	NU	SR	VR	PR	NU
loadI	12	-	-	∞	-	-	-	∞	r_3	-	-	1
load	r_3	-	-	1	-	-	-	∞	r_4	-	-	1
mult	r_3	-	-	∞	r_4	-	-	∞	r_4	-	-	∞

Рис. 13.3 Представление списка операций

13.3.1. Переименование в локальном распределителе

Чтобы упростить реализацию локального распределителя, компилятор может сначала переименовать исходные регистры, так чтобы они соответствовали интервалам жизни. В одном блоке LR состоит из одного определения и одного или нескольких использований. *Протяженность* LR – это интервал между его определением и последним использованием в блоке.

Алгоритм переименования находит интервал жизни каждого значения в блоке. Он назначает каждому LR новое имя – имя его VR. Наконец, он переписывает код в терминах VR. Переименование создает взаимно однозначное соответствие между LR и VR, что, в свою очередь, упрощает многие структуры данных в локальном распределителе. Затем распределитель рассуждает о VR, а не о произвольных именах SR.

Компилятор может найти интервалы жизни и переименовать их в VR на одном обратном проходе по блоку. По ходу дела он может также собрать и запомнить информацию о следующем использовании для каждого определения и использования в блоке. В алгоритме на рис. 13.4 предполагается представление, описанное в предыдущем разделе.

Алгоритм переименования строит два отображения: *SRToVR* отображает имя SR в имя VR, а *PrevUse* отображает имя SR в порядковый номер его самого недавнего использования. Вначале алгоритм инициализирует все элементы *SRToVR* значением *invalid*, а все элементы *PrevUse* – значением ∞.

Алгоритм обходит блок от последней операции к первой. Для каждой операции он сначала посещает определения, а затем использования. Для каждого операнда он обновляет отображения и определяет поля VR и NU.

Посещая определение или использование, алгоритм сначала проверяет, назначен ли VR регистру SR операнда, *O.SR*. Если нет, то он назначает SR следующий свободный VR и запоминает этот факт в *SRToVR[O.SR]*. Затем он запоминает имя VR и информацию о следующем использовании в записи операнда. Если операнд используется, то алгоритм записывает

Если SR для определения не имеет VR, то это значение никогда не используется. Но алгоритм все равно назначает VR регистру SR.

в *PrevUse[O.SR]* индекс текущей операции. Для определения он сбрасывает *PrevUse[O.SR]* обратно в ∞.

```
VRName ← 0
for i ← 0 to максимального номера исходного регистра do
  SRToVR[i] ← invalid
  PrevUse[i] ← ∞
index ← длина блока
for каждой операции Op в блоке снизу вверх do
  for каждого операнда, O, который определяет Op do     // сначала определения
    if SRToVR[O.SR] = invalid then                       // у определения нет использований
      SRToVR[O.SR] ← VRName++                            // все равно начать новый VR
    O.VR ← SRToVR[O.SR]                                  // установить VR и NU для O
    O.NU ← PrevUse[O.SR]
    PrevUse[O.SR] ← ∞
    SRToVR[O.SR] ← invalid          // следующее использование SR начинает новый VR

  for для каждого операнда, O, который использует Op do
                                    // использования после определений
    if SRToVR[O.SR] = invalid then  // начать новый VR
      SRToVR[O.SR] ← VRName++
    O.VR ← SRToVR[O.SR]             // установить VR и NU для O
    O.NU ← PrevUse[O.SR]
  for каждого операнда, O, который использует Op do
    PrevUse[O.SR] ← index           // сохранить, чтобы установить следующий NU

  index ← index - 1
```

Рис. 13.4 Переименование исходных регистров в интервалы жизни

Заметим, что все операнды `store` являются использованиями. `store` определяет место в памяти, а не регистр.

Алгоритм посещает определения раньше использований, чтобы отображения корректно обновлялись в случаях, когда имя SR встречается и в определении, и в использовании. Например, в операции `add` r_{17}, $r_{18} \Rightarrow r_{18}$ алгоритм заменит определение на *SRToVR[*r_{18}*]*, обновит *SRToVR[*r_{18}*]* новым именем VR для использования, а затем положит *PrevUse[*r_{18}*]* равным ∞.

После переименования словосочетания *интервал жизни* и *виртуальный регистр* используются как синонимы.

После переименования у каждого интервала жизни имеется уникальное имя VR. Имя SR, определенное в нескольких местах, переписано как несколько разных VR. Кроме того, у каждого операнда в блоке имеется поле NU, в которое записан либо порядковый номер следующей операции в блоке, где используется его значение, либо ∞, если значение больше нигде не используется. Распределитель использует эту информацию, когда выбирает, какие VR выгружать.

Снова рассмотрим код на рис. 13.1. На рисунке (a) показан оригинальный код, на рисунке (b) – он же после переименования. На рисунке (c) показана протяженность каждого диапазона интервального графика. Распределитель

не переименовывает r_{arp}, потому что это выделенный PR, в котором хранится указатель на запись активации, и, следовательно, он не находится под контролем распределителя.

MAXLIVE
Максимальное число одновременно живых VR в блоке.

Максимальный спрос на регистры, `MAXLIVE`, имеет место в начале первой операции `mult`, помеченной на рисунке (c) штриховой линией. В этой точке живы шесть VR. В начале операции живы VR_7 и VR_8. Эта операция является последним использованием VR_7 и VR_8, а также определением VR_5.

13.3.2. Распределение и назначение

Алгоритм локального распределителя показан на рис. 13.5. Он выполняет распределение и назначение на одном прямом проходе по блоку. Вначале предполагается, что никакие значения не находятся в PR. Он обходит операции по порядку и инкрементно выделяет PR каждому VR. Для каждой операции распределитель выполняет три шага.

1. Чтобы у каждого использования гарантированно был PR, распределитель сначала ищет PR в отображении *VRToPR*. Если в записи для VR находится *invalid*, то алгоритм вызывает *GetAPR*, чтобы найти PR. Распределитель использует простую схему пометки, чтобы избежать выделения одного и того же PR конфликтующим использованиям в одной операции.

Если одна команда содержит несколько операций, то распределитель должен обработать все использования раньше любого определения.

2. На втором шаге распределитель проверяет, являются ли какие-нибудь использования последними использованиями VR. Если да, то он может освободить PR, сделав его доступным для нового назначения – либо результату текущей операции, либо какому-то VR в будущей операции.
3. На третьем шаге распределитель гарантирует, что каждому VR, определенному операцией, выделен PR для хранения его значения. И снова для поиска подходящего регистра распределитель вызывает *GetAPR*.

Все шаги простые, кроме выбора значения для выгрузки. Сложность локального распределения в основном кроется в решении этой задачи.

Механизм работы GetAPR

Обрабатывая операцию, распределитель должен найти PR для любого VR *v*, которому регистр еще не назначен. Это и есть сущность задачи распределения регистров. Возможны два случая.

1. *Какой-то PR p свободен*. Распределитель может назначить *p* регистру *v*. Ради эффективности алгоритм ведет стек свободных PR.

2. *Свободных PR нет*. Распределитель должен выбрать, у какого VR отнять его PR *p*, сохранить значение *p* в его место выгрузки и назначить *p* для хранения *v*.

Если ссылка на *v* является использованием, то распределитель должен затем восстановить значение *v* из памяти в *p*.

```
for vr ← 0 to максимальный номер VR do
  VRToPR[vr] ← invalid
for pr ← 0 to максимальный номер PR do
  PRToVR[pr] ← invalid
  PRNU[pr] ← ∞
  push(pr)      // pop() выполняется в GetAPR()

// обойти блок
for каждой операции OP в блоке в линейном порядке do
  очистить пометку в каждом PR  // сбросить пометки
  for каждого использования U в OP do
                // распределить использования
    pr ← VRToPR[U.VR]
    if (pr = invalid) then
      U.PR ← GetAPR(U.VR,U.NU)
      Restore(U.VR,U.PR)
    else
      U.PR ← pr
    установить пометку в U.PR

  for каждого использования U в OP do
                // последнее использование?
    if (U.NU = ∞ and PRToVR[U.PR] ≠ invalid) then
      FreeAPR(U.PR)

  очистить пометку в каждом PR  // сбросить пометки

  for каждого определения D в OP do
                // распределить определения
    D.PR ← GetAPR(D.VR, D.NU)
    установить пометку D.PR
```

```
GetAPR(vr, nu)
  if стек не пуст then
    x ← pop()
  else
    выбрать непомеченный x
      для выгрузки
    Spill(x)
  VRToPR[vr] ← x
  PRToVR[x] ← vr
  PRNU[x] ← nu
  return x

FreeAPR(pr)
  VRToPR[PRToVR[pr]] ← invalid
  PRtoVR[pr] ← invalid
  PRNU[pr] ← ∞
  push(pr)
```

Рис. 13.5 Локальный распределитель

Эвристика Бэста говорит, что распределитель должен выгружать PR, для текущего VR которого следующее использование максимально далеко. Алгоритм поддерживает структуру *PRNU*, чтобы это решение можно было воплотить в жизнь. Он просто выбирает PR с максимальным *PRNU*. Если распределитель находит два PR с одинаковыми *PRNU*, то должен выбрать какой-то один.

Реализация *PRNU* – компромисс между эффективностью обновления и поиска. Алгоритм обновляет *PRNU* при каждой ссылке на регистр. Он производит поиск в *PRNU* при каждой выгрузке. Как показано на рисунке, *PRNU* – простой массив; если обновления производятся гораздо чаще, чем выгрузка, то это имеет смысл. Если выгрузки достаточно часты, то реализация *PRNU* с помощью приоритетной очереди могла бы улучшить время распределения.

Код выгрузки и восстановления

В точке, где распределитель вставляет код выгрузки, все физические регистры (PR) уже использованы. Автор компилятора должен гарантировать, что распределитель все же может выгрузить значение.

Возможны два сценария. Вероятнее всего, ISA целевой машины поддерживает режим адресации, который позволяет осуществить выгрузку без необходимости в дополнительном PR. Например, если для УЗА имеется выделенный регистр, скажем r_{arp}, и ISA включает операцию сохранения с непосредственной адресацией, как, например, `storeAI` в ILOC, то места выгрузки в области локальных данных можно адресовать без дополнительного PR.

Если целевая машина поддерживает только простые загрузку и сохранение или реализацию, в которой места выгрузки не могут находиться в записи активации, то компилятор должен зарезервировать какой-то PR для вычисления адреса, уменьшив тем самым пул доступных PR. Конечно, зарезервированный регистр понадобится, только если MAXLIVE > k. (Если MAXLIVE ⩽ k, то никакая выгрузка не нужна, как не нужен и зарезервированный регистр.)

Отслеживание физических и виртуальных регистров

Чтобы знать о связи между VR и PR, распределитель поддерживает два отображения. *VRToPR* для каждого VR содержит либо имя назначенного ему в текущий момент PR, либо значение *invalid*. *PRToVR* для каждого PR содержит либо имя VR, которому он в текущий момент назначен, либо значение *invalid*.

Во время прохода по блоку распределитель обновляет оба этих отображения, так чтобы всегда выполнялся следующий инвариант:

if *VRToPR*[**vr**] ≠ *invalid* then *PRToVR*[*VRToPR*[**vr**]] = **vr**.

Код *GetAPR* и *FreeAPR* следит за соблюдением инварианта при работе с этими отображениями. Кроме того, эти процедуры поддерживают массив *PRNU*, который сопоставляет PR порядковый номер операции, в которой он используется в следующий раз; эта величина интерпретируется как расстояние до следующего использования.

Выгрузка и восстановление

Концептуально реализация процедур *Spill* и *Restore* на рис. 13.5 может быть очень простой.

- Чтобы выгрузить PR p, распределитель может использовать *PRToVR* для нахождения VR v, который в настоящий момент находится в p. Если у v еще нет места выгрузки, то распределитель назначает ему таковое. Затем он генерирует операцию для сохранения p в месте выгрузки. Наконец, он обновляет отображения *VRToPR*, *PRToVR* и *PRNU*.

Места выгрузки обычно размещаются в области локальных данных в конце записи активации.

- Чтобы восстановить VR *v* в PR *p*, распределитель просто генерирует операцию загрузки из места выгрузки *v* в *p*. На последнем шаге он обновляет отображения *VRToPR*, *PRToVR* и *PRNU*.

Если стоимость всех выгрузок одинакова и стоимость всех восстановлений одинакова, то алгоритм Бэста порождает оптимальное распределение для блока.

Затруднения, связанные со стоимостью выгрузки

В реальных программах стоимости выгрузки и восстановления неравномерны. В коде встречаются чистые и грязные значения; стоимость выгрузки грязного значения выше, чем чистого. Чтобы убедиться в этом, рассмотрим выполнение локального распределителя в блоке, содержащем ссылки на три имени, x_1, x_2 и x_3, при наличии всего двух PR ($k = 2$).

Предположим, что распределитель регистров находится в точке, где x_1 и x_2 находятся в регистрах, и x_1 чистое, а x_2 грязное. На рис. 13.6 показано, как разные решения о выгрузке влияют на код в этих двух сценариях.

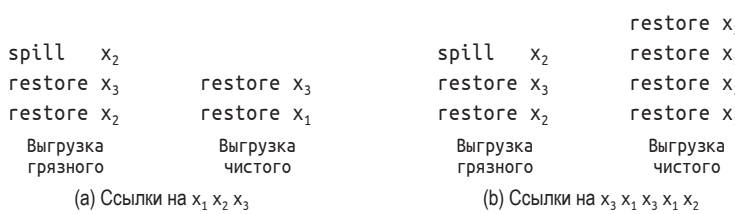


Рис. 13.6 Выгрузка чистых и грязных значений

Строка ссылок
Строка ссылок – это просто список ссылок на регистры или адреса. В этом контексте каждая ссылка является использованием, а не определением.

На рисунке (a) рассмотрен случай, когда строка ссылок для оставшейся части блока – x_3 x_1 x_2. Если распределитель всегда выгружает чистые значения раньше грязных, то для этой строки ссылок размер кода выгрузки будет меньше.

На рисунке (b) рассмотрен случай, когда строка ссылок для оставшейся части блока – x_3 x_1 x_3 x_1 x_2. Здесь, если распределитель всегда выгружает чистые значения раньше грязных, то размер кода выгрузки будет больше.

Присутствие чистых и грязных значений фундаментально изменяет задачу локального распределения. Стоит распределителю столкнуться с двумя видами значений с разной стоимостью выгрузки, как задача становится NP-трудной. Появление пересчитываемых значений, из-за которых стоимость восстановления также становится неравномерной, делает задачу еще сложнее. Поэтому быстрый детерминированный распределитель не может всегда принимать оптимальные решения о выгрузке. Однако такие локальные распределители могут порождать хорошие решения, делая выбор из LR с разной стоимостью выгрузки с помощью относительно простой эвристики.

Напомним, однако, что задача NP-трудная. Для нее не существует эффективного детерминированного алгоритма, который всегда дает оптимальные результаты.

На практике распределитель может найти лучшие распределения, если будет различать грязные, чистые и пересчитываемые значения (см. раздел 13.2.3). Если расстояния от двух PR до их следующих использований одинаковы, а стоимости выгрузки различны, то распределитель должен выгружать PR с меньшей стоимостью.

Однако проблема выбора между PR усложняется, если стоимости их выгрузки различны, а расстояния до следующего использования близки, но не одинаковы. Например, если дано грязное значение с расстоянием до следующего использования n и пересчитываемое значение с расстоянием $n - 1$, то второе значение иногда оказывается лучшим выбором.

ИТОГИ РАЗДЕЛА

Ограниченность контекста в задаче локального распределения регистров в достаточной мере упрощает задачу, чтобы быстрый, интуитивно понятный алгоритм мог найти высококачественные распределения. Принцип работы локального распределителя, описанного в этом разделе, прост: *когда нужен PR, выгрузить PR, для которого следующее использование отстоит максимально далеко в будущем.*

В блоке, где для всех значений стоимость выгрузки одинакова, локальный распределитель достиг бы оптимальных результатов. Если же распределитель должен иметь дело с грязными и чистыми значениями, то задача становится комбинаторно сложной. Локальный распределитель может порождать хорошие результаты, но не может гарантировать оптимальных.

ВОПРОСЫ

1. Модифицируйте алгоритм переименования на рис. 13.4, так чтобы он вычислял также MAXLIVE – максимальное число одновременно живых значений в любой команде блока.
2. Пересчитывать известную константу – естественное решение, потому что выгрузка не требует никакого кода, а для восстановления нужно выполнить одну операцию непосредственной загрузки. При каких условиях распределителю было бы выгодно пересчитать такую операцию, как add r_a, $r_b \Rightarrow r_x$?

13.4. Глобальное распределение с помощью раскрашивания

Большинство компиляторов используют глобальный распределитель регистров, который рассматривает всю процедуру как контекст. Глобальный распределитель должен учитывать более сложный поток управления, чем локальный. В интервалах жизни может быть много определений и использований; они образуют сложные переплетения поперек графа управления потоком. Различные блоки исполняются разное число раз, что усложняет оценку стоимости выгрузки. И хотя интуитивные соображения, относящиеся к локальному

распределению, в какой-то мере помогают, алгоритмы глобального распределения гораздо запутаннее.

Выгрузка LR разбивает его на небольшие части, которые можно хранить в разных PR.

Глобальный распределитель сталкивается еще с одной сложностью: он должен координировать использование регистров в нескольких блоках. В локальном алгоритме отображение зарегистрированных LR в PR, по сути дела, произвольно. Напротив, глобальный распределитель должен либо хранить LR в одном и том же регистре во всех блоках, где он жив, либо вставлять код для перемещения его между регистрами.

Большинство схем глобального распределения основаны на одной парадигме. Они представляют конфликты между использованиями регистров графом интерференции, а затем раскрашивают этот граф, чтобы найти распределение. При такой модели автор компилятора должен принять множество решений. Интервалы жизни могут быть короткими или длинными. Граф может быть точным или приближенным. Когда распределитель должен что-то выгружать, он может выгружать LR везде или только в регионах, где спрос на регистры высок.

Из-за разнообразия вариантов существуют разные алгоритмы. В этом разделе мы рассмотрим один конкретный набор проектных решений: интервалы жизни максимальной длины, точный граф интерференции и дисциплина выгрузки везде. Эти решения определяют глобальный распределитель на основе раскрашивания. В разделе 13.5 исследуются варианты и улучшения этого распределителя.

На рис. 13.7 показана структура глобального распределителя на основе раскрашивания.

Найти интервалы жизни. Распределитель находит интервалы жизни и переписывает код, подставляя уникальные имена для каждого LR. Новое пространство имен гарантирует, что на различные значения не налагаются ограничения просто потому, что они имеют одинаковое имя во входном коде.

Построить граф. Распределитель строит граф интерференции. Он создает вершину для каждого LR и добавляет ребро из LR_i в любую LR_j, которая жива в операции, определяющей LR_i, если только эта операция не является копированием. Построение графа занимает львиную долю времени распределения.

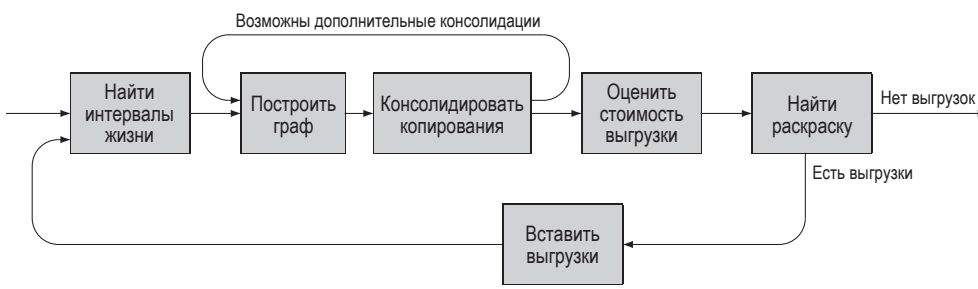


Рис. 13.7 Структура глобального распределителя на основе раскрашивания

Консолидация копий. Распределитель смотрит на каждую операцию копирования $LR_i \Rightarrow LR_j$. Если LR_i и LR_j не интерферируют, то он объединяет LR, удаляет копирование и обновляет граф. Консолидация уменьшает количество LR и степени других вершин графа.

К сожалению, обновление графа консервативное, а не точное (см. раздел 13.4.3). Таким образом, если какие-то LR объединяются, то распределитель повторяет процесс *Построить–Консолидировать*, пока не окажется, что доступных для консолидации LR не осталось – обычно для этого хватает 2–4 итераций.

В вычислении стоимости выгрузки есть много специальных случаев (см. разделы 13.2.3 и 13.4.4).

Оценить стоимость выгрузки. Распределитель вычисляет для каждого LR оценку стоимости выгрузки всего LR на этапе выполнения. Он складывает стоимость выгрузок и восстановлений, умножая каждую на оценку частоты выполнения блока, в который нужно было бы вставить код.

Найти раскраску. Распределитель пытается построить *k*-раскраску графа интерференции. Он применяет двухэтапный процесс: упрощение графа с целью построить порядок раскрашивания с последующей реконструкцией графа, в ходе которой назначаются цвета по мере обратной вставки каждой вершины в граф.

Если распределитель находит *k*-раскраску, то он переписывает код и выходит. Если какие-то вершины остались нераскрашенными, то распределитель вызывает процедуру *Вставить выгрузки*, которая выгружает нераскрашенные LR. Затем распределитель заново запускается на модифицированном коде.

Вторая и последующие попытки раскрашивания занимают меньше времени, чем первая, потому что в результате консолидации на первом проходе уменьшился как размер задачи, так и размер графа интерференции.

Вставить выгрузки. Для каждого нераскрашенного LR распределитель вставляет код выгрузки после каждого определения и код восстановления перед каждым использованием. В результате нераскрашенный LR преобразуется в набор крохотных LR, по одному в каждой ссылке. Эту модифицированную программу проще раскрасить, чем оригинальный код.

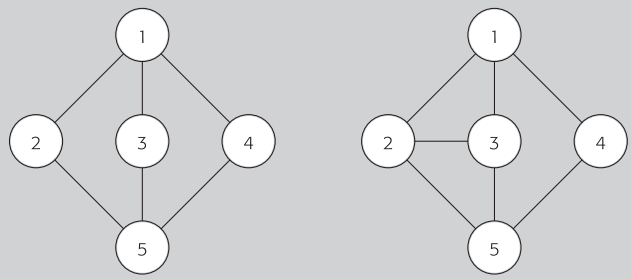
В следующих разделах эти шаги описываются более подробно.

13.4.1. Нахождение глобальных интервалов жизни

На первом шаге глобальный распределитель строит глобальные интервалы жизни максимальной длины (см. раздел 13.2.1). Глобальный LR – это множество определений и использований, содержащее все использования, которых может достичь входящее в набор определение, а также все определения, которые могут достичь этих использований. Таким образом, LR образует сложное переплетение определений и использований, которое в идеале должно размещаться в одном PR.

Раскрашивание графа

Раскрашивание графа – распространенная парадигма глобального распределения регистров. Для произвольного графа G раскраска сопоставляет цвет каждой вершине G таким образом, что никакие две соседние вершины не окрашены одним цветом. Раскраска, в которой используется k цветов, называется k-раскраской, а наименьшее k для заданного графа называется *минимальным хроматическим числом* графа. Рассмотрим следующие графы:



Граф слева является 2-раскрашиваемым. Например, мы можем назначить цвет *синий* вершинам 1 и 5 и *красный* вершинам 2, 3 и 4. Добавление ребра (2, 3), как показано на рисунке справа, делает граф 3-раскрашиваемым, но не 2-раскрашиваемым. (Назначаем *синий* вершинам 1 и 5, *красный* вершинам 2 и 4 и *желтый* вершине 3.)

Для заданного графа нахождение минимального хроматического числа является NP-трудной задачей. Аналогично NP-трудная задача – определение того, является ли граф k-раскрашиваемым для заданного k. Распределители на основе раскрашивания графа применяют приближенные методы для нахождения раскрасок, совместимых с доступными ресурсами.

Максимальная степень вершин графа дает верхнюю границу его хроматического числа. Граф, для которого максимальная степень вершины равна x, всегда можно раскрасить $x + 1$ цветами. Оба показанных выше графа демонстрируют, что эта граница неточная. Максимальная степень для обоих графов равна 3. Но оба графа можно раскрасить менее чем четырьмя цветами. В обоих случаях у вершин с большой степенью есть соседи, которые можно окрасить одним цветом.

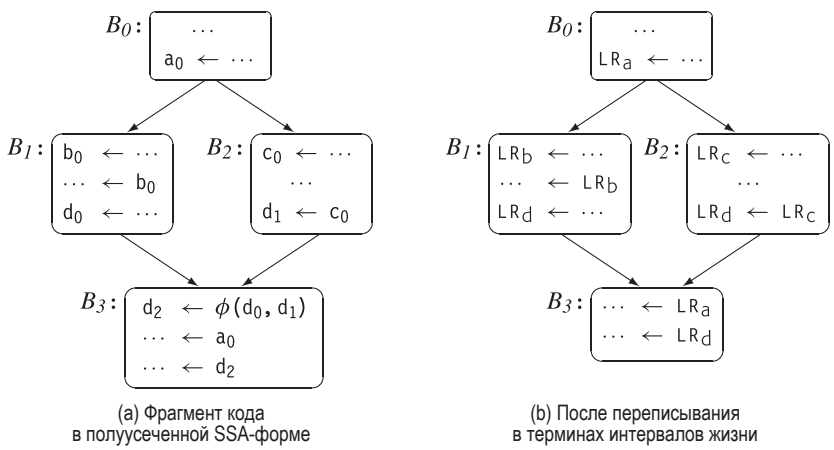
Алгоритм построения интервалов жизни прост, если распределитель может работать с SSA-формой кода. Таким образом, первый шаг нахождения интервалов жизни – преобразовать входной код в форму SSA, если это необходимо. Затем распределитель может построить интервалы жизни максимального размера, применяя простой подход: для каждой φ-функции объединить все имена, определений и использований, в один LR. Применив это правило к каждой φ-функции, распределитель создаст набор максимальных глобальных LR.

Чтобы сделать этот процесс эффективным, компилятор может использовать алгоритм построения объединения непересекающихся множеств. Вначале он назначает уникальное имя множества каждому SSA-имени. Затем он посещает каждую φ-функцию в коде и объединяет множества, ассоциированные с каждым параметром φ-функции, и множество для результата φ-функции. После того как все φ-функции будут обработаны, каждое остав-

шееся уникальное множество становится LR. Распределитель может либо переписать код с использованием имен LR, либо создать и поддерживать отображение между SSA-именами и именами LR. На практике первый подход кажется проще.

Поскольку в процессе нахождения LR никакие определения или использования не перемещаются, трансляция из формы SSA тривиальна. Пространство имен LR улавливает эффекты, которые потребовали бы создания новых копий в ходе трансляции из формы SSA. Таким образом, компилятор может просто опустить φ-функции и SSA-имена в процессе переименования.

На рис. 13.8(a) показан фрагмент кода в полуусеченной SSA-форме, включающий исходные переменные a, b, c и d. Для нахождения интервалов жизни распределитель назначает каждому SSA-имени содержащее его множество. Он объединяет множества, ассоциированные с именами, используемыми в φ-функции, $\{d_0\} \cup \{d_1\} \cup \{d_2\}$. Это дает окончательный набор из четырех LR: $LR_a = \{a_0\}$, $LR_b = \{b_0\}$, $LR_c = \{c_0\}$ и $LR_d = \{d_0, d_1, d_2\}$. На рис. 13.8(b) показан код, переписанный с использованием LR.

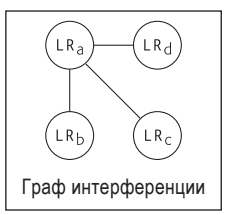


(a) Фрагмент кода в полуусеченной SSA-форме

(b) После переписывания в терминах интервалов жизни

Рис. 13.8 Обнаружение интервалов жизни

13.4.2. Построение графа интерференции



Граф интерференции

Для моделирования интерференций глобальный распределитель строит граф интерференции, $I = (N, E)$. Вершины, элементы множества N, представляют отдельные интервалы жизни, а ребра, элементы множества E, – интерференции между ними. Таким образом, неориентированное ребро $(n_i, n_j) \in E$ существует тогда и только тогда, когда соответствующие интервалы жизни LR_i и LR_j интерферируют. Граф интерференции для кода на рис. 13.8(b) показан во врезке. LR_a интерферирует с каждым из LR_b, LR_c и LR_d. Ни один из LR_b, LR_c и LR_d не интерферирует ни с каким другим; они могли бы разделять один PR.

```
for каждого LR_i do
  создать вершину n_i ∈ N

for каждого простого блока b do
  LiveNow ← LiveOut(b)
  for каждой операции i в b_i снизу вверх
     в предположении, что i имеет форму op_i LR_a, LR_b ⇒ LR_c do
    удалить LR_c из LiveNow
    for каждого LR_i ∈ LiveNow† do
      добавить ⟨LR_i, LR_c⟩ в E
    добавить LR_i и LR_b в LiveNow

† Если операция является
копированием, LR_i ⇒ LR_j,
не добавлять ребро ⟨LR_i, LR_j⟩.
```

Рис. 13.9 Построение графа интерференции

Если компилятор сможет раскрасить I в k или меньше цветов, то сможет отобразить цвета непосредственно в PR и получить допустимое распределение. В примере выше LR_a интерферирует с каждым из LR_b, LR_c и LR_d. При раскрашивании LR_a должен получить свой собственный цвет, а при распределении он не может делить PR с LR_b, LR_c или LR_d. Остальные интервалы жизни не интерферируют друг с другом. Следовательно, LR_b, LR_c и LR_d могли бы быть окрашены одним цветом и, стало быть, делить один PR. Этот граф интерференции является 2-раскрашиваемым, и код можно переписать так, что будет использоваться всего два регистра.

Теперь посмотрим, что произошло бы, если бы на каком-то другом этапе компилятор изменил порядок двух последних определений в B_1, как показано во врезке. При таком изменении LR_b оказывается жив в точке определения LR_d. Это добавляет ребро (LR_b, LR_d) в граф интерференции, что делает его 3-, а не 2-раскрашиваемым. (Граф настолько мал, что доказать это можно прямым перебором.) Для нового графа у распределителя есть два варианта: использовать три регистра или, если доступно всего два регистра, выгрузить LR_b или LR_a, перед тем как определять LR_d в B_1. Альтернативно распределитель мог бы изменить порядок обеих операций и исключить интерференцию между LR_b и LR_d. Обычно распределители регистров не переупорядочивают операции, а предполагают, что порядок операций фиксирован, и оставляют вопрос о его изменении на усмотрение планировщика команд (см. главу 12).



B_1 после изменения порядка определений

Имея код, переименованный с использованием LR, и множества LiveOut для каждого блока в переименованном коде, распределитель может построить граф интерференции на одном проходе по блоку, как показано на рис. 13.9. Алгоритм обходит блок снизу вверх. В каждой операции он вычисляет LiveNow – множество значений, живых в текущей операции. В конце блока LiveOut и LiveNow должны совпадать. По мере прохода вверх по блоку алгоритм добавляет в граф ребра интерференции и обновляет множество LiveNow с целью отразить воздействие каждой операции.

В разделе 11.3.2 тоже используются множества LiveNow.

Алгоритм реализует данное ранее определение интерференции: LR_i и LR_j интерферируют, если LR_i жив в точке определения LR_j, или наоборот. Распределитель добавляет в каждой операции интерференцию между определенным LR и каждым LR, живым после этой операции.

Операции копирования нуждаются в специальной обработке. Копирование $LR_i \Rightarrow LR_j$ не создает интерференцию между LR_i и LR_j, потому что оба интервала жизни имеют одно и то же значение после копирования и, следовательно, могли бы занимать один и тот же регистр. Если последующий контекст создает интерференцию между этими интервалами жизни, то эта операция создаст ребро. Такая обработка копирований порождает граф интерференции, который точно улавливает, когда LR_i и LR_j могут занимать один регистр. Встречая операции копирования, распределитель должен создавать список всех операций копирования для последующего использования при консолидации.

При вставке в списки следует справляться с битовой матрицей, чтобы избежать дубликатов.

Для повышения эффективности распределитель должен строить нижнетреугольную битовую матрицу и множество списков смежности для представления E. Битовая матрица позволяет проверять интерференцию за постоянное время, а списки смежности обеспечивают эффективный обход соседей вершины. Такая стратегия с двумя представлениями потребляет больше памяти, чем одно представление, но это окупается уменьшением времени распределения. Как было предложено в разделе 13.2.4, распределитель может строить отдельные графы для непересекающихся классов регистров, что уменьшает максимальный размер графа.

13.4.3. Консолидация операций копирования

Распределитель может использовать граф интерференции для реализации сильной формы консолидации копирований. Если код содержит операцию копирования, $LR_i \Rightarrow LR_j$, и распределитель может установить, что LR_i и LR_j не интерферируют в какой-то другой операции, то он может объединить оба LR и устранить одну операцию копирования. Мы говорим, что копирования были «консолидированы».

В своей диссертации Бриггс приводит примеры, когда консолидация устраняет до трети первоначальных интервалов жизни.

У консолидации копирований есть несколько положительных последствий. Она устраняет фактическую операцию копирования, что делает код меньше и потенциально быстрее. Он уменьшает степень любого LR, который раньше интерферировал с LR_i и LR_j. Она удаляет вершину из графа. Благодаря всему этому проход раскрашивания оказывается быстрее и эффективнее.

На рис. 13.10 показан простой пример с одним блоком. Оригинальный код приведен на рисунке (а). Интервалы справа обозначают протяженности интервалов жизни, участвующих в операции копирования. Несмотря на то

что LR_a перекрывается с LR_b и LR_c, он не интерферирует ни с тем, ни с другим, потому что перекрытия приходятся на операции копирования. Поскольку LR_b жив в точке определения LR_c, интервалы LR_b и LR_c интерферируют. Обе операции копирования являются кандидатами на консолидацию.

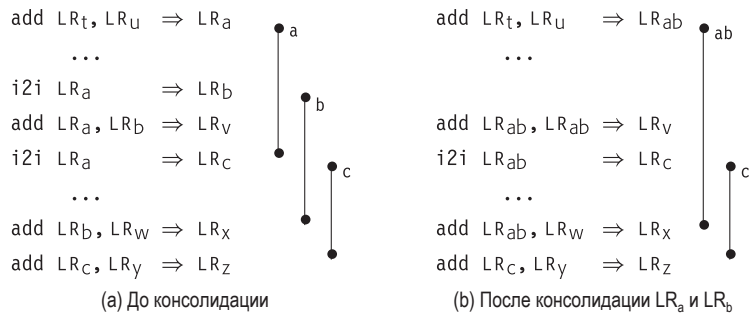


Рис. 13.10 Объединение интервалов жизни путем консолидации операций копирования

На рис. 13.10(b) показан результат консолидации LR_a и LR_b с образованием LR_{ab}. LR_{ab} и LR_c по-прежнему не интерферируют, потому что LR_c создан операцией копирования из LR_{ab}. Объединение LR_a и LR_b уменьшает LR_v° на единицу. До консолидации LR_a и LR_b интерферировали с LR_v. После консолидации эти значения занимают один LR вместо двух. В общем случае консолидация двух интервалов жизни либо уменьшает степени их соседей, либо оставляет их неизменными, но увеличить степени соседей она не может.

Чтобы выполнить консолидацию, распределитель обходит список операций копирования, построенный *Build Graph*, и проверяет каждую операцию, $LR_i \Rightarrow LR_j$. Если $(LR_i, LR_j) \notin E$, то LR_i и LR_j не интерферируют, и распределитель объединяет их, удаляет копирование и обновляет I с целью отразить новый объединенный интервал LR_{ij}. Распределитель может обновлять I консервативно, перемещая каждое ребро из LR_i и LR_j в LR_{ij} и устраняя дубликаты. Это обновление неточное, но позволяет распределителю продолжить консолидацию.

> При проверке принадлежности следует использовать битовую матрицу для повышения эффективности.

На практике распределители консолидируют каждый интервал жизни, который могут, зная интерференции в I. Затем они переписывают код, чтобы отразить пересмотренные LR и устранить консолидированные копирования. Далее они перестраивают I и снова пытаются консолидировать копирования. Обычно этот процесс останавливается после двух раундов консолидации.

Пример иллюстрирует неточную природу этого консервативного обновления графа. Обновление перемещает ребро (LR_b, LR_c) из LR_b в LR_{ab}, хотя на самом деле LR_{ab} и LR_c не интерферируют. Перестройка графа на основе преобразованного кода дает точный граф интерференции, без ребра (LR_{ab}, LR_c). В этот момент распределитель может консолидировать LR_{ab} и LR_c.

Если распределитель может консолидировать LR_i либо с LR_j, либо с LR_k, то решение сформировать LR_{ij} может предотвратить последующую консо-

лидацию с LR_k, и наоборот. Следовательно, порядок консолидации важен. В принципе, компилятор должен сначала консолидировать самые часто выполняемые операции копирования. Таким образом, распределитель мог бы консолидировать операции копирования в порядке убывания оценки частоты выполнения блока, содержащего операцию. Чтобы реализовать эту идею, распределитель может рассматривать простые блоки в порядке убывания глубины вложенности.

В этой стратегии учтен урок, преподанный построением полуусеченной SSA-формы: включать только имена, которые важны для дела.

На практике стоимость построения графа интерференции в первом раунде консолидации составляет преобладающую часть всей стоимости распределителя на основе раскрашивания графа. Последующие итерации цикла построения-консолидации обрабатывают меньший граф и потому работают быстрее. Чтобы уменьшить стоимость консолидации, компилятор может построить подмножество графа интерференции, включающее только интервалы жизни, участвующие в операции копирования.

13.4.4. Оценка глобальной стоимости выгрузки

Обнаружив, что не может разместить все интервалы жизни в регистрах, распределитель должен выбрать, какой LR выгрузить. Обычно распределитель применяет тщательно спроектированную метрику для ранжирования вариантов и выбирает тот, который согласно этой метрике является наилучшим кандидатом. Локальный распределитель использовал для этой цели расстояние до следующего использования LR, и в одноблочном контексте такая метрика хорошо работает. В глобальном распределителе метрика включает оценку затрат на выгрузку и восстановление конкретного LR во время выполнения.

Чтобы вычислить оценку затрат на выгрузку LR, распределитель должен исследовать каждое определение и использование в LR. Для каждого определения он оценивает стоимость выгрузки после него и умножает эту величину на оценку частоты выполнения блока, содержащего определение. Для каждого использования он оценивает стоимость восстановления перед ним и умножает эту величину на оценку частоты выполнения блока, содержащего использование. Он складывает полученные оценки стоимостей и считает сумму стоимостью выгрузки для данного LR.

Разумеется, фактическое вычисление сложнее, чем описано выше. Для данного определения или использования в LR значение может быть грязным, чистым или пересчитываемым (см. раздел 13.2.3). У разных определений и использований может быть разная классификация, поэтому распределитель должен выполнить достаточно подробный анализ, чтобы классифицировать все ссылки в LR. Эта классификация определяет стоимость выгрузки или восстановления данной ссылки.

Определить точное количество выполнений блока трудно. К счастью, для принятия решений о выгрузке достаточно относительных частот выполне-

ния; распределителю нужно знать, что одна ссылка, скорее всего, выполняется гораздо чаще другой. Таким образом, распределитель выводит для каждого блока число, обозначающее относительную частоту его выполнения. Эти частоты применяются к каждой ссылке внутри блока.

Распределитель мог бы вычислять стоимость выгрузки по запросу – когда возникает необходимость принять решение о выгрузке. Если он находит k-раскраску без выгрузок, то вычисление стоимости по запросу могло бы уменьшить общее время распределения. Если распределитель должен выгружать регистры часто, то пакетное вычисление стоимости, вероятно, обошлось бы дешевле вычислений по запросу.

Рисунок 13.7 наводит на мысль, что распределителю стоило бы выполнить вычисление стоимости до того, как попытается раскрасить граф. Распределитель может отложить вычисление до момента, когда придется выполнить первую выгрузку. Если выгружать вообще ничего не придется, то он сможет избежать накладных расходов на вычисление стоимости выгрузки; если же без выгрузки все же не обойтись, то он будет вычислять стоимости для меньшего множества LR.

Учет частот выполнения

Чтобы вычислить стоимость выгрузки, распределителю нужно оценить частоту выполнения каждого простого блока. Компилятор может вывести эти оценки из данных профилирования или эвристически. Многие компиляторы предполагают, что каждый цикл выполняется 10 раз, вследствие чего блок, входящий в состав цикла с глубиной вложенности d, получает вес 10^d. Таким образом, блоку, входящему в состав одного цикла, назначается вес 10, входящему в два вложенных цикла – вес 100 и т. д. Непредсказуемое предложение `if-then-else` уменьшает оценку частоты вдвое. На практике смещение этих оценок настолько велико, что предпочтительнее осуществлять выгрузку LR во внешних циклах, чем во внутренних.

> Использование оценки 10^d может привести к проблеме из-за целочисленного переполнения при вычислении стоимости выгрузки.
>
> Многие авторы компиляторов испытали эту проблему на собственной шкуре. Для глубоко вложенных циклов при вычислении стоимости выгрузки лучше использовать операции с плавающей точкой.

Отрицательная стоимость выгрузки

Интервал жизни, который содержит операции загрузки, сохранения и больше никаких использований, должен получить отрицательную стоимость выгрузки, если загрузка и сохранение ссылаются на один и тот же адрес. (Такие интервалы жизни могут быть созданы в результате оптимизации; например, если использование было устранено, а сохранение было результатом вызова процедуры, а не определения нового значения.) Иногда выгрузка интервала жизни может устранять операции копирования с более высокой стоимостью,

чем у операций выгрузки; такой интервал жизни тоже имеет отрицательную стоимость. Любой интервал жизни с отрицательной стоимостью выгрузки должен быть выгружен, потому что это уменьшает спрос на регистры *и* удаляет операции из кода.

Бесконечная стоимость выгрузки

```
LRi ← ...
Mem[LRj] ← LRi
```
Стоимость выгрузки LR_i равна ∞

Некоторые интервалы жизни настолько короткие, что их выгрузка не помогает. Рассмотрим короткий LR, показанный во врезке. Если распределитель попробует выгрузить LR_i, то вставит операцию сохранения после определения и операцию загрузки перед использованием, создав два новых LR. Ни один из этих новых LR не использует меньше регистров, чем исходный, поэтому выгрузка не дает никакого преимущества. Распределитель должен назначить исходному LR бесконечную стоимость выгрузки, гарантируя тем самым, что не будет пытаться его выгрузить. В общем случае LR должен иметь бесконечную стоимость выгрузки, если никакой другой LR не заканчивается между его определениями и его использованиями. Это условие означает, что между определениями и использованиями доступность регистров не изменяется.

```
инициализировать стек и список выгрузки
пустыми множествами
while (N ≠ ∅) do
  if ∃n ∈ N с n° < k then
    node ← n
  else node ← n, выбранная с учетом
    метрики выгрузки
  удалить вершину и ее ребра из I
  поместить вершину в стек
```
(a) Алгоритм Simplify

```
while (stack ≠ ∅) do
  node ← pop(stack)
  вставить вершину и ее ребра в I
  назначить вершине цвет, не используемый
    ее соседями
```
(b) Алгоритм Select

Рис. 13.11 *Simplify* и *Select*: базовый алгоритм раскрашивания

Интервалы бесконечной стоимости представляют для компилятора проблему в части формы кода. Если код содержит более $k - 1$ вложенных LR бесконечной стоимости и ни один LR не заканчивается в этом регионе, то LR бесконечной стоимости образуют нераскрашенную клику в графе интерференции. Хотя такая ситуация необычна, она встречалась нам на практике. Распределитель регистров не может тут ничего исправить; автор компилятора просто должен гарантировать, что распределитель не получит такой код.

13.4.5. Раскрашивание графа

Глобальный распределитель раскрашивает граф в два этапа. На первом шаге, называемом *Simplify*, вычисляется, в каком порядке пытаться выполнить раскрашивание. На втором шаге, *Select*, все вершины рассматриваются по по-

рядку, и распределитель пытается назначить вершине цвет из своей палитры k цветов.

Для раскрашивания графа распределитель полагается на простое наблюдение:

> *Если вершина имеет меньше k соседей, то она должна получить цвет независимо от того, какие цвета назначены ее соседям.*

Таким образом, любую вершину n степени, меньшей k (обозначается $n^\circ < k$), раскрасить тривиально.

Simplify

Чтобы вычислить порядок раскрашивания, распределитель находит тривиально раскрашенные вершины и удаляет их из графа. Он запоминает порядок удаления, помещая удаленные вершины в стек. Удаление вершины и ассоциированных с ней ребер из графа уменьшает степени всех ее соседей. Алгоритм показан на рис. 13.11(a).

При удалении вершин из графа распределитель должен сохранить вершину и ее ребра для последующей обратной вставки в *Select*. Распределитель может либо построить структуру для их запоминания, либо включить в состав данных о каждом ребре и каждой вершине отметку, показывающую, активны они или нет.

В *Simplify* используется два разных механизма для выбора вершины, которая подлежит удалению следующей. Если существует вершина n с $n^\circ < k$, то распределитель выбирает эту вершину. Поскольку такие вершины окрашены тривиально, порядок, в котором они удаляются, не имеет значения. Если все оставшиеся вершины ограничены и имеют степень $\geqslant k$, то распределитель выбирает вершину для удаления, исходя из ее *метрики выгрузки*. Любая вершина n, удаляемая в ходе этого процесса, имеет $n^\circ \geqslant k$; следовательно, она не может получить цвет на этапе назначения. Цикл останавливается, когда граф оказывается пустым. В этот момент стек содержит все вершины в порядке их удаления.

Метрика выгрузки
эвристика, используемая для выбора LR, подлежащего выгрузке

Select

Чтобы раскрасить граф, распределитель перестраивает граф интерференции в порядке, обратном порядку удаления. Он в цикле извлекает вершину n из стека, вставляет n и ее ребра обратно в I и выбирает для n цвет, отличный от цветов ее соседей. Алгоритм показан на рис. 13.11(b).

Чтобы выбрать цвет вершины n, распределитель подсчитывает цвета соседей n в текущем графе и назначает n неиспользуемый цвет. Он может просматривать множество цветов в одном

Наш опыт показывает, что порядок, в котором распределитель рассматривает цвета, не имеет существенного практического значения.

и том же порядке или назначать цвета циклически. Если для n не хватает цвета, она остается нераскрашенной.

Когда стек оказывается пуст, граф I перестроен. Если у каждой вершины имеется цвет, то распределитель переписывает код, заменяя имена LR именами PR, и возвращается. Если какие-то вершины остались нераскрашенными, распределитель выгружает соответствующие LR. Распределитель передает список нераскрашенных LR процедуре *Вставить выгрузки*, которая добавляет операции выгрузки и восстановления в код. Затем *Вставить выгрузки* перезапускает распределитель для модифицированного кода. Процесс повторяется, пока каждая вершина I не получит цвет. Обычно распределитель находит раскраску и останавливается после двух проходов по большому циклу на рис. 13.7.

Почему это работает?

Глобальный распределитель вставляет каждую вершину обратно в граф, из которого она была удалена. Если алгоритм упрощения удаляет вершину для LR_n из I, потому что $n^\circ < k$, то он заново вставляет LR_n в граф, в котором $n^\circ < k$ и вершина n тривиально раскрашена.

Единственный случай, когда вершина n может не получить цвет, – если n была удалена из I вследствие использования метрики выгрузки. *Select* заново вставляет такую вершину в граф, в котором $n^\circ \geqslant k$. Заметим, однако, что это условие является утверждением о степени в графе, а не о доступности цветов.

Обновление графа интерференции

И консолидация, и выгрузка изменяют множество вершин и ребер в графе интерференции. В обоих случаях граф необходимо обновить, прежде чем продолжать распределение.

В глобальном распределителе на основе раскрашивания производится консервативное обновление после каждой консолидации; это обновление также запускает еще одну итерацию цикла *Построение–Консолидация* на рис. 13.7. Точность графа достигается путем перестройки его с нуля.

Распределитель откладывает вставку кода выгрузки до конца процесса *Simplify–Select*, а затем вставляет весь этот код и запускает новую итерацию цикла *Построение–Консолидация–Вычисление стоимости выгрузки–Раскрашивание*. И на этот раз точность достигается путем перестройки графа с нуля.

Если бы распределитель умел точно обновлять граф, то можно было бы устранить оба цикла на рис. 13.7. Консолидацию можно было бы завершить за один проход. Вставлять код выгрузки можно было бы инкрементно при обнаружении нераскрашенной вершины; обновленный граф правильно отражал бы интерференции для выбора цвета.

Улучшенные инкрементные обновления могут уменьшить время распределения. Точное обновление дало бы то же распределение, что оригинальный распределитель с точностью до изменения порядка принятия решений. Неточное, но консервативное обновление могло бы ускорить распределение, хотя и ценой возможного ухудшения качества кода в силу неточности графа.

Если соседи n использовали все k цветов, то распределитель не найдет цвета для n. Но если они использовали меньше k цветов, то цвет для n будет найден. На практике вершина n часто имеет нескольких соседей, окрашенных в один и тот же цвет. Таким образом, *Select* нередко находит цвета для некоторых из таких ограниченных вершин.

Simplify определяет, в каком порядке *Select* пытается раскрашивать вершины. Этот порядок играет важнейшую роль в определении того, какие вершины будут раскрашены. Для вершины n, удаленной из графа, потому что $n^\circ < k$, порядок относительно вершин, остающихся в графе, неважен. Порядок может быть важен относительно вершин, уже находящихся в стеке; ведь n, возможно, имела ограничения до момента, когда некоторые из этих предыдущих вершин были удалены. Для вершин, удаленных из графа вследствие метрики выгрузки, порядок критически важен. Ветвь else на рис. 13.11(а) выполняется, только когда все оставшиеся вершины имеют степень $\geqslant k$. Таким образом, вершины, оставшиеся в графе в этот момент, принадлежат более сильно связным подграфам I.

Оригинальный распределитель на основе раскрашивания появился в компиляторе PL.8 компании IBM.

Порядок ограниченных вершин определяется *метрикой выгрузки*. Оригинальный распределитель на основе раскрашивания выбирал вершину, которая минимизировала отношение *cost÷degree*, где *cost* – оценка стоимости выгрузки, а *degree* – степень вершины в текущем графе. Эта метрика балансирует стоимость выгрузки с количеством вершин, чья степень уменьшится.

Были испробованы и другие метрики выгрузки. Некоторые из них являются вариациями на тему *cost÷degree*, в т. ч. $cost \div degree^2$, *cost÷area* и $cost \div area^2$, где *area* для LR определяется как сумма MAXLIVE, вычисленная по всем командам, оказавшимся внутри LR. Эти метрики пытаются балансировать стоимость выгрузки конкретного LR с тем, насколько эта выгрузка облегчит раскрашивание оставшегося графа. Экспериментировали также с самой *cost*; здесь упор делается на скорости выполнения. В приложениях, чувствительных к потреблению памяти, метрика «полное число операций выгрузки» может способствовать уменьшению роста потребления памяти в результате выгрузки.

На практике ни одна эвристика не является лучшей во всех случаях. Поскольку раскрашивание занимает мало времени по сравнению с построением I, распределитель может раскрасить I, применив несколько метрик, а затем оставить наилучший результат.

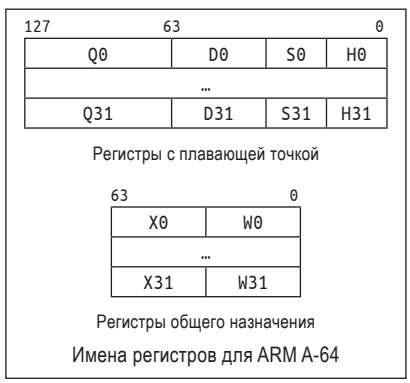
13.4.6. Вставка кода выгрузки и восстановления

Код выгрузки, созданный глобальным распределителем регистров, не сложнее кода, вставленного локальным распределителем. Процедура *Вставить выгрузки* получает список нераскрашенных LR. Для каждого LR она вставляет нужный код после каждого определения и перед каждым использованием.

В глобальном случае возникают те же сложности, что в локальном распределителе. Распределитель должен понимать разницу между грязными, чистыми и пересчитываемыми значениями. На практике в глобальном контексте труднее понять, к какому виду относится значение.

Глобальный распределитель применяет дисциплину *выгружать везде*. Нераскрашенный интервал жизни выгружается в каждой точке определения и восстанавливается в каждой точке использования. На практике выгруженный LR часто содержит определения и использования, которые встречаются в регионах, где доступны физические регистры. Несколько авторов изучали способы ослабить дисциплину повсеместной выгрузки, с тем чтобы оставлять LR в PR в тех регионах, где спрос на регистры мал (см. обсуждение в разделе 13.5).

13.4.7. Обработка совмещенных классов регистров



На практике распределитель регистров должен быть готов к странным свойствам набора регистров целевой машины и действующим для нее соглашениям о вызове. Эта реальность налагает ограничения и на распределение, и на назначение.

Например, на ARM A-64 все четыре регистра с плавающей точкой, `Q1`, `D1`, `S1` и `H1`, делят место в одном PR, как показано во врезке. Таким образом, если компилятор выделяет `D1` под LR_i, то `Q1`, `S1` и `H1` оказываются недоступны, пока LR_i жив. Аналогичные ограничения возникают из-за совмещенных регистров общего назначения, таких как пара `X3` и `W3`.

Чтобы понять, как совмещение классов регистров влияет на структуру распределителя регистров, рассмотрим, как можно было бы модифицировать локальный распределитель с целью обработки регистров общего назначения в ARM A-64.

- В представленном выше алгоритме используется один атрибут для описания LR – его номер виртуального регистра. В случае совмещенных классов, например `Xi` и `Wi`, каждому LR необходим еще атрибут для описания его класса.
- В стеке свободных регистров следует использовать имена, взятые из одного из двух наборов, `Xi` или `Wi`. Состояние – выделен или свободен – совмещенной пары, например `X0` и `W0`, одинаково.
- Отображения *VRToPR*, *PRToVR* и *PRNU* могут остаться однозначными. Если для VR_i выделен регистр, то *VRToPR* будет отображать поле VR *num* на номер регистра, а его поле *class* будет указывать, какое имя – `X` или `W` – следует использовать.

Поскольку в локальном алгоритме имеется простой способ смоделировать состояние PR, расширения не вызывают трудностей.

Для обработки более сложной ситуации, например регистра EAX в IA-32, в локальный распределитель придется внести более обширные изменения. Использование EAX требует всего регистра и запрещает использовать одновременно с ним AH или AL. Аналогично применение AH или AL запрещает одновременно использовать EAX. Однако распределитель вправе одновременно использовать AH и AL. Подобные особые правила применяются и к другим совмещенным именам, показанным во врезке и на рис. 13.2(b).



Имена регистров для Intel IA-32

Глобальные распределители на основе раскрашивания графа имеют более сложные модели интерференции и доступности регистров, чем локальный распределитель. Чтобы адаптировать их для справедливого использования совмещенных классов регистров, требуется более хитроумный подход.

Описание классов регистров

Прежде чем описывать систематический способ обработки распределения и назначения для нескольких классов регистров, нам необходим систематический способ описания самих этих классов. Компилятор может представить члены каждого класса множеством. На рис. 13.2(a) мы видим, что в ARM A-64 имеется шесть классов:

```
Q: {Q0, Q1, ... , Q31}    D: {D0, D1, ... , D31}    S: {S0, S1, ... , S31}
H: {H0, H1, ... , H31}    X: {X0, X1, ... , X31}    W: {W0, W1, ... , W31}
```

Таким образом, *class*(D3) = {D0, D1, ..., D31} – множество 64-разрядных регистров с плавающей точкой.

Проще всего описать совмещение классов регистров можно с помощью функции *alias*(*r*). Для имени регистра *r* *alias*(*r*) отображает *r* на множество имен регистров, занимающих физическое пространство, пересекающееся с пространством *r*. В ARM A-64 *alias*(W1) = {X1}, а *alias*(S2) = {Q2, D2, H2}. Аналогично в IA-32 *alias*(AH) = {EAX}, *alias*(AL) = {EAX}, а *alias*(EAX) = {AL, AH}. Поскольку AL и AH занимают непересекающиеся пространства, они не являются псевдонимами друг друга. Компилятор может вычислить информацию, которая ему понадобится для распределения и назначения по связям, определяемым функциями *class* и *alias*.

Раскрашивание с совмещенными классами

Наличие совмещенных классов регистров усложняет раскрашивание, назначение и консолидацию.

Раскрашивание

Распределитель на основе раскрашивания графа аппроксимирует раскрашиваемость путем сравнения степени вершины с количеством доступных цветов. Если $n^\circ < k$, числа доступных регистров, то *Simplify* классифицирует n как тривиально раскрашиваемую, удаляет n из графа и помещает в упорядочивающий стек. Если граф не содержит тривиально раскрашиваемых вершин, то *Simplify* выбирает следующую вершину для удаления с использованием своей метрики выгрузки.

Наличие нескольких классов регистров означает, что k может зависеть от класса. Для вершины n, которая представляет LR_n, $k = |class(LR_n)|$.

Наличие совмещенных классов регистров еще больше усложняет аппроксимацию раскрашиваемости. Если у класса LR нет псевдонимов, то применима простая арифметика: один сосед уменьшает запас возможных регистров на единицу. Если у класса LR есть псевдонимы, т. е. классы регистров совмещены, то может случиться, что один сосед уменьшает запас возможных регистров больше чем на единицу.

- В IA-32 `EAX` убирает `AH` и `AL`; пул 8-разрядных регистров сокращается на два. Эта связь не симметрична: использование `AL` не препятствует использованию `AH`. Пара невыровненных регистров с плавающей точкой – пример более общего варианта этой проблемы.
- С другой стороны, ARM A-64 гарантирует, что каждый сосед считается как один регистр. Например, `W2` занимает младшие 32 разряда регистра `X2`; для старших разрядов `X2` никакого имени не существует. Регистры с плавающей точкой обладают таким же свойством: с любым 128-разрядным регистром (`Qi`) ассоциировано по одному имени для каждой точности.

Если мы хотим обобщить *Simplify* таким образом, чтобы она справедливо и корректно работала с совмещенными классами регистров, то должны заменить условие $n^\circ < k$ консервативной и правильной оценкой раскрашиваемости в более сложном случае совмещенных классов регистров. Вместо того чтобы подсчитывать соседей n, мы должны подсчитать тех соседей, с которыми n конкурирует за регистры. Смит, Рамсей и Холлоуэй описывают эффективный способ получить справедливую и корректную оценку раскрашиваемости. Их распределитель заранее вычисляет вспомогательные данные на основе информации о классах и псевдонимах, а затем оценивает раскрашиваемость n, зная класс n и регистры, назначенные ее релевантным соседям.

Назначение

Традиционно при обсуждении распределителей на основе раскрашивания графа предполагается, что назначение конкретных регистров конкретным интервалам жизни не оказывает существенного влияния на раскрашиваемость. В литературе игнорируется различие между выбором цветом «по кругу» и по принципу «первым пришел, первым ушел», если только речь не идет о таких необычных случаях, как смещенное раскрашивание (см. раздел 13.5.1).

В случае совмещенных классов регистров некоторые варианты выбора могут ограничить использование более одного регистра. В IA-32 использование `EAX` уменьшает запас 8-разрядных регистров на два, `AH` и `AL`, а не на один. И на этот раз пара невыровненных регистров с плавающей точкой порождает более общий вариант проблемы. Как одно назначение регистра может конфликтовать с несколькими другими, так одно назначение может изменять набор доступных инкрементных вариантов.

Рассмотрим поиск одного 8-разрядного регистра в IA-32. Если бы доступны были `AL` и `CL`, но `AH` занят, а `CH` нет, то выбор `AL` привнес бы меньше ограничений. Поскольку `EAX` уже конфликтует с `AH`, выбор `AL` не уменьшает множество доступных `E`-регистров. С другой стороны, при выборе `CL` регистр `ECX` стал бы недоступен. Совмещенные классы регистров усложняют назначение столь существенно, что распределителю следует выбирать регистры, используя более сложный механизм, чем стек из раздела 13.3.

Консолидация

Компилятор должен консолидировать два интервала жизни, LR_i и LR_j, только если для результирующего интервала имеется подходящий класс регистров. Например, если оба регистра общего назначения, то объединение работает. Если *class*(LR_i) содержит только PR_2, а *class*(LR_j) – только PR_4, то распределитель должен понять, что объединенный LR_{ij} будет чрезмерно ограниченным (см. дальнейшее обсуждение в разделе 13.5.1).

Раскрашивание с непересекающимися классами

Если архитектура включает набор непересекающихся классов, то компилятор может распределять их по отдельности. Например, большинство процессоров предоставляют раздельные ресурсы для регистров общего назначения и регистров с плавающей точкой. Поэтому распределение для LR_i регистра с плавающей точкой напрямую не влияет на распределение регистров общего назначения. Поскольку выгрузка значений с плавающей точкой может приводить к созданию значений, нуждающихся в регистрах общего назначения, распределение регистров с плавающей точкой должно предшествовать распределению регистров общего назначения.

Если распределитель строит отдельные графы для непересекающихся подклассов, то может уменьшить число вершин в графе интерференции, что способно дать значительную экономию на этапе компиляции, особенно во время шага *Построение графа*.

Принудительное размещение в конкретных регистрах

Распределитель должен обрабатывать операции, которые требуют размещения интервалов жизни в конкретных PR. Это могут быть ограничения (LR_i должен находиться в PR_j) или исключения (LR_i не может находиться в PR_k).

Такого рода требования могут проистекать из нескольких источников. Соглашение о связях диктует размещение передаваемых значений в регист-

В IA-32 AX описывает 16-разрядный регистр, совмещенный с AH и AL. Аналогично DX совмещен с DH и DL.

рах; эти требования касаются УЗА, некоторых или всех фактических параметров и возвращаемого значения. Некоторые аппаратные операции требуют, чтобы их операнды находились в определенных регистрах; например, умножение 16-разрядных целых со знаком в IA-32 всегда читает аргумент из AX и записывает результат в AX и DX.

Механизм классов регистров дает простой способ обработать такие ограничения. Автор компилятора создает небольшой класс регистров для этой цели и присоединяет этот класс к соответствующим LR. Остальные детали берет на себя механизм раскрашивания.

Для обработки исключений автор компилятора может создать набор исключений – тоже список PR – и присоединить его к соответствующим LR. Механизм раскрашивания может сравнить потенциальные варианты с набором исключений. Например, между кодом, который сохраняет регистры, сохраняемые вызывающей стороной, и кодом, который их восстанавливает, распределитель не должен использовать регистры, сохраняемые вызывающей стороной, для хранения чего-либо, кроме временных значений. Простой набор исключений может гарантировать такое безопасное поведение.

ИТОГИ РАЗДЕЛА

Глобальные распределители регистров работают на уровне процедур. Наличие потока управления затрудняет глобальное распределение по сравнению с локальным. Большинство глобальных распределителей основаны на парадигме раскрашивания графа. Распределитель строит граф, представляющий интерференцию между интервалами жизни, а затем пытается найти раскраску, для которой достаточно имеющихся регистров.

В этом разделе описан глобальный распределитель, в котором используются точный граф интерференции и тщательная оценка стоимости выгрузки. Благодаря точному графу интерференции возможна мощная стадия консолидации операций копирования. Распределитель производит выгрузку с помощью жадного эвристического алгоритма выбора, применяя дисциплину выгрузки везде. Эти решения приводят к эффективной реализации.

ВОПРОСЫ

1. *Simplify* всегда удаляет тривиально раскрашенные вершины ($n^\circ < k$), прежде чем удалить вершину с ограничениями ($n^\circ \geq k$). Это наводит на мысль, что она выгружает вершину, только если у нее имеется по меньшей мере k соседей, которые сами являются ограниченными. Напишите набросок процедуры *Simplify*, в которой используется этот более точный критерий. Как ее стоимость на этапе выполнения соотносится со стоимостью оригинального алгоритма? Ожидаете ли вы, что результаты будут иными?
2. Глобальный распределитель выбирает значение для выгрузки, отыскивая LR, который минимизирует некоторую метрику, например *стоимость выгрузки / степень*. Во время работы алгоритм иногда вынужден выбирать для выгрузки несколько интервалов жизни, прежде чем какой-то другой интервал освободится от ограничений. Объясните, как такое может произойти. Можете ли вы придумать метрику выгрузки, не подверженную этой проблеме?

13.5. Дополнительные темы

Поскольку стоимость неверного шага при распределении регистров может быть высока, алгоритмам распределения было уделено весьма много внимания. Многочисленные вариации глобального распределителя на основе раскрашивания описаны в литературе и реализованы практически. В разделах 13.5.1 и 13.5.2 описываются другие подходы к консолидации и выгрузке соответственно. В разделе 13.5.3 представлены три других определения интервалов жизни; все они приводят к существенно отличным распределителям.

13.5.1. Вариации на тему консолидации

Представленный выше алгоритм консолидации объединяет интервалы жизни, не обращая внимания на раскрашиваемость результирующего интервала. В литературе было предложено несколько более консервативных подходов.

Консервативная и повторяющаяся консолидация

У консолидации имеются как положительные, так и отрицательные последствия. Как было отмечено выше, консолидация LR_i и LR_j может уменьшить степень других LR, интерферирующих с обоими. Однако $LR^\circ_{ij} \geqslant MAX(LR^\circ_i, LR^\circ_j)$. Если и LR_i, и LR_j тривиально раскрашены и $LR^\circ_{ij} \geqslant k$, то консолидация LR_i и LR_j увеличивает количество ограниченных LR в графе, но может затруднить раскрашивание без выгрузки, хотя это и не обязательно.

Цель *консервативной консолидации* – попытаться ограничить отрицательные побочные эффекты консолидации, объединяя LR_i и LR_j только тогда, когда это не усугубляет трудность раскрашивания графа интерференции. Будучи записано буквально, это утверждение сводится к следующему условию:

Консервативная консолидация
Распределитель консолидирует $LR_i \Rightarrow LR_j$, только если результирующий LR не делает раскрашивание графа более трудным.

Либо $LR^\circ_{ij} \leq MAX(LR^\circ_i, LR^\circ_j)$, *либо* LR_{ij} *имеет меньше* k *соседей степени* $> k$.

Это тонкое условие. Если один из LR_i или LR_j уже имеет большую степень и консолидация LR_i и LR_j порождает LR той же степени, то результат раскрасить не труднее, чем оригинальный граф. На самом деле консолидация понизила бы степень любого LR, интерферирующего одновременно с LR_i и LR_j.

Второе условие означает, что LR_{ij} должно обладать тем свойством, что *Simplify* и *Select* находят цвет для LR_{ij}. Допустим, что распределитель может консолидировать LR_i и LR_j для создания LR_{ij}. Если LR_{ij} имеет степень больше, чем оба LR,

При сравнении с k следует использовать надлежащие значения *class*(LR_i) и *class*(LR_j).

которые она заменяет, но все равно получает цвет, то распределитель может объединить LR_i и LR_j. (Консолидация остается консервативной.)

Консервативная консолидация привлекательна именно потому, что не может сделать задачу раскрашивания труднее. Однако она не дает компилятору консолидировать некоторые операции копирования. Поскольку степень – неточная верхняя граница раскрашиваемости, консервативная консолидация может предотвратить некоторые полезные объединения и, следовательно, привести к большему числу выгрузок, чем неограниченная консолидация.

Смещенное раскрашивание

Смещенное раскрашивание
Если LR_i и LR_j соединены операцией копирования, то распределитель пытается назначить им один и тот же цвет.

Другой способ консолидировать копирования, не усложняя раскрашивание графа, – сместить выбор в пользу конкретных цветов. *Смещенное раскрашивание* откладывает консолидацию до `Select`; оно изменяет процесс выбора цвета. Выбирая цвет для LR_i, оно сначала пробует цвета, которые были назначены интервалам жизни, соединенным с LR_j операцией копирования. Если оно может назначить LR_i цвет, уже назначенный LR_j, то копирование из LR_i в LR_j или из LR_j в LR_i избыточно, и распределитель может устранить эту операцию.

Чтобы сделать этот процесс эффективным, распределитель может построить для каждого LR список других LR, с которыми он соединен операцией копирования. Затем *Select* может использовать эти *списки партнеров*, чтобы быстро определить, допускает ли какой-нибудь из доступных цветов объединение LR с одним из его партнеров. При аккуратной реализации смещенное раскрашивание почти или совсем не увеличивает стоимость процесса выбора цвета.

Повторяющаяся консолидация

Повторяющаяся консолидация
Распределитель повторяет консервативную консолидацию, прежде чем принять решение о выгрузке LR.

В распределителе, пользующемся консервативной консолидацией, некоторые операции копирования останутся неконсолидированными, потому что у результирующего LR степень была бы выше. *Повторяющаяся консолидация* решает эту проблему, пытаясь выполнить консервативную консолидацию до принятия решения о выгрузке. *Simplify* удаляет из графа вершины, пока в нем не останется ни одной тривиально раскрашиваемой вершины. В этот момент она повторяет шаг консолидации. Операции копирования, не консолидированные в предыдущих графах, возможно, удастся консолидировать в упрощенном графе. Если консолидация порождает дополнительные тривиально раскрашиваемые вершины, то *Simplify* удаляет и их. В противном случае она выбирает в графе кандидатов для выгрузки, пока не будет создана одна или несколько тривиально раскрашиваемых вершин.

13.5.2. Вариации на тему выгрузки

Распределитель, описанный в разделе 13.4, применял дисциплину «выгружать везде». На практике распределитель может осуществлять более точную выгрузку, чтобы уменьшить спрос на регистры в тех регионах, где он высок. Это наблюдение привело к нескольким интересным улучшениям распределителя, выгружающего везде.

Выгрузка частичных интервалов жизни

Глобальный распределитель, описанный выше, выгружает интервалы жизни целиком. Эта стратегия может приводить к избыточной выгрузке, если спрос на регистры низок в большей части интервала жизни, а высок лишь в небольшом регионе. Более изощренные методы выгрузки находят регионы, где выгрузка интервала жизни продуктивная, т. е. освобождает регистр в том регионе, где он действительно необходим. Глобальный распределитель может достичь похожих результатов, выгружая только в том регионе, где имеет место интерференция. Один из методов, называется *выгрузкой в регионе интерференции*, находит множество интервалов жизни, которые интерферируют в регионе высокого спроса, и выгружает их только в этом регионе. Распределитель может оценить стоимости нескольких стратегий выгрузки для региона интерференции и сравнить их со стоимостью стандартного подхода с выгрузкой везде. Показано, что такая конкуренция оценок стоимости улучшает распределение в целом.

Чистая выгрузка

Выгружая несколько LR_i, глобальный распределитель вставляет код выгрузки после каждого определения и код восстановления после каждого использования. Если в LR_i встречается несколько использований в блоке, где спрос на регистры низок, то аккуратная реализация может оставить значение LR_i в регистре на протяжении той части интервала жизни, которая находится в данном блоке. Цель этого улучшения, которое иногда называют *чистой выгрузкой* (clean spilling), – попытаться сделать так, чтобы данный LR восстанавливался в данном блоке только один раз.

Вариант той же идеи – организация более общего последующего прохода по преобразованному распределителем коду с целью найти регионы, где имеются свободные регистры, и вернуть выгруженные значения обратно в регистры в этих регионах. Этот подход называется *прочисткой регистров* (register scavenging).

Пересчет

Некоторые значения дешевле повторно вычислить, чем выгружать. Например, небольшие целые константы лучше заново создавать с помощью операции непосредственной загрузки, чем извлекать из памяти операцией загруз-

ки. Распределитель может распознавать такие значения и пересчитывать их, а не выгружать.

Модификация глобального распределителя на основе раскрашивания графа с целью включить пересчет требует нескольких мелких изменений. Распределитель должен находить и помечать тегами SSA-имена, которые можно пересчитать. Например, любая операция, аргументы которой всегда доступны, является кандидатом. Он может распространять теги по коду, применяя вариант алгоритма распространения констант SSCP, который был описан в главе 9. При формировании интервалов жизни распределитель должен объединять только SSA-имена с одинаковыми тегами пересчета.

Автор компилятора должен сделать так, чтобы в оценке стоимости выгрузки правильно учитывались теги пересчета. Процесс вставки кода выгрузки также должен анализировать теги пересчета и генерировать упрощенный код для пересчитываемых значений. Наконец, распределитель должен использовать консервативную консолидацию, чтобы избежать преждевременного объединения интервалов жизни с разными тегами пересчета.

Расщепление интервала жизни

Вставка кода выгрузки изменяет и код, и задачу раскрашивания. Нераскрашенный LR разбивает на последовательность малых LR, по одному на каждое определение и использование. Распределитель может использовать похожий эффект, чтобы улучшить распределение; он может намеренно расщеплять LR большой степени способами, которые улучшают раскрашиваемость или локализуют выгрузку.

При расщеплении интервалов жизни используются три разных эффекта. Если расщепленные LR имеют меньшие степени, чем исходный, то их может быть проще раскрасить – возможно, даже без ограничений. Если некоторые из расщепленных LR имеют высокую степень и потому подлежат выгрузке, то расщепление может дать распределителю возможность избежать выгрузки других частей LR, которые имеют меньшую степень. Наконец, расщепление индуцирует выгрузку в точках, где производится разбиение LR на части. Тщательный выбор точек расщепления поможет контролировать размещение кода выгрузки – например, поощрять его размещение вне циклов, а не внутри их.

Было опробовано много подходов к расщеплению. Один из ранних распределителей разбивал нераскрашенные LR на LR размера блока, а затем снова консолидировал их, если консолидация не делала распределение труднее, – как при консервативной консолидации. Проводились эксперименты с подходами, где для выбора точек расщепления используются свойства графа управления потоком. Результаты могут оказаться противоречивыми, поскольку решаемые задачи остаются NP-полными.

Два метода оказались многообещающими. Метод *расщепления нулевой стоимости* основан на вставке операций `nop` в план команд с целью расщепить LR и улучшить как распределение, так и планирование. В методе *пассив-*

ного расщепления ориентированный граф интерференции применяется для решения о том, какие LR расщеплять и где именно; для этого используются оценки стоимостей каждой альтернативы.

Реализация расщепления

Механизм расщепления интервала жизни может быть непростым. Бриггс предложил отдельную операцию *расщепления* с таким же поведением, как у копирования. В его распределителе использовалась агрессивная консолидация операций копирования. После их консолидации он применял консервативную консолидацию к расщепленным интервалам.

Обработка неоднозначных значений

Если код содержит много неоднозначных значений, возникших в результате использования указателей в исходном языке, из ссылок на массивы или ссылок на объекты, класс которых нельзя определить на этапе компиляции, то неспособность распределителя хранить такие значения в регистрах является серьезной проблемой для производительности. Чтобы улучшить распределение неоднозначных значений, некоторые системы включают преобразования, направленные на то, чтобы хранить однозначные значения в скалярных локальных переменных, даже когда их «естественное» место – элемент массива или структуры, адресуемой указателем.

- При скалярной замене используется анализ индексов массива, чтобы выявить повторно используемые элементы массива и завести для их значений временные скалярные переменные.
- Техника продвижения регистров подразумевает анализ потока значений указателей с целью найти такие значения, которые можно безопасно хранить в регистре во вложенном цикле. Код переписывается, так чтобы значение хранилось в локальной скалярной переменной.

Оба преобразования кодируют результаты анализа в форме кода и делают очевидным для распределителя регистров тот факт, что значения можно хранить в регистрах.

Техника продвижения может увеличить спрос на регистры. На самом деле продвижение слишком большого числа значений может порождать код выгрузки, обходящийся дороже тех операций с памятью, которых преобразование пыталось избежать. В идеале техника продвижения должна использовать какую-то меру спроса на регистры, решая, какие значения продвигать. К сожалению, хорошие оценки спроса на регистры трудно построить.

13.5.3. Другие формы интервалов жизни

Распределитель из раздела 13.4 оперирует интервалами жизни максимального размера. В других распределителях используются иные понятия интервала жизни, которые изменяют как сам распределитель, так и порождаемое им распределение. У этих изменений есть хорошие и плохие последствия.

Более короткие интервалы жизни в некоторых случаях порождают графы интерференции, содержащие больше тривиально раскрашиваемых вершин. Рассмотрим значение, которое живо в одном блоке, где спрос на регистры больше k, и во многих блоках, где спрос меньше k. В случае LR максимального размера раскрасить весь LR нетривиально; если же LR более короткие, то некоторые из них могут оказаться тривиально раскрашиваемыми. Это может привести к улучшенному использованию регистров в областях с низким спросом. С другой стороны, более короткие LR по-прежнему представляют одно значение. Поэтому они должны соединяться операциями копирования или операциями с памятью, которые сами имеют стоимость.

Интервалы жизни максимального размера могут порождать графы общего вида. Точнее, для любого графа можно построить процедуру, граф интерференции которой будет ему изоморфен. Ограничение на форму LR может ограничить форму графа интерференции. В следующих подразделах описываются три определения интервалов жизни; каждое из них можно считать высокоуровневым описанием вытекающих из него распределителей.

В замечаниях к этой главе имеются ссылки для читателей, которые хотели бы подробнее познакомиться с этими распределителями.

Каждому из этих распределителей соответствует точка в пространстве проектирования. Изменив определение интервала жизни, мы повлияем на точность графа интерференции и на стоимость распределения. Исход компромиссов неясен в значительной части потому, что исходная задача остается NP-полной, а распределители стараются быстро вычислить аппроксимацию оптимального решения.

Распределение на основе SSA-имен

Графы интерференции, получающиеся из интервалов жизни максимального размера в программах, являются графами общего вида. Для таких графов задача нахождения k-раскраски является NP-полной. Однако существуют классы графов, для которых найти k-раскраску можно за полиномиальное время.

Хордальный граф
граф, в котором любой цикл, содержащий более трех вершин, имеет хорду – ребро, соединяющее две вершины, не являющиеся смежными в цикле.

В частности, оптимальную раскраску хордального графа можно найти за время $O(|N| + |E|)$. В оптимальной раскраске может быть меньше цветов, а стало быть, она требует меньше регистров, чем находит жадный эвристический алгоритм, описанный в разделе 13.4.5. Конечно, если оптимальная раскраска требует больше k цветов, то распределителю все равно придется производить выгрузку регистров.

Если компилятор рассматривает каждое SSA-имя как интервал жизни, то получающийся граф интерференции будет хордальным. Это наблюдение вызвало интерес к глобальному распределению регистров для кода в SSA-форме. Основанный на SSA распределитель может найти распределения,

требующие меньше регистров, чем найденные глобальным распределителем на основе раскрашивания.

Если графу нужно больше k цветов, то распределитель должен выгрузить одно или более значений. Хотя форма SSA не уменьшает сложность задачи о выборе выгружаемых значений, она может дать некоторые преимущества. Глобальные интервалы жизни обычно живут дольше, чем SSA-имена, жизнь которых прерывается φ-функциями в надлежащих местах кода, например в заголовках циклов и в блоках, следующих над циклами. Эти разрывы дают распределителю шанс выгружать значения в меньших регионах, чем если бы использовались глобальные интервалы жизни.

Если трансляция из SSA-формы должна разорвать цикл операций копирования, то ей потребуется для этого дополнительный регистр.

К сожалению, распределение на основе SSA оставляет код в SSA-форме. Распределитель или следующий за ним проход должны произвести трансляцию из SSA-формы со всеми сопутствующими сложностями, которые мы обсуждали в разделе 9.3.5. Эта трансляция может увеличить спрос на регистры. Распределитель на основе SSA должен быть готов к обработке такой ситуации.

Не менее важно и то, что трансляция вставляет в код операции копирования; некоторые из них могут быть излишни. Распределитель не может консолидировать операции копирования, которые реализуют поток значений, соответствующий φ-функции; при этом было бы утрачено хордальное свойство графа. Поэтому основанный на SSA распределитель, скорее всего, использовал бы алгоритм консолидации, которому не нужен граф интерференции. Существует несколько таких алгоритмов.

Трудно дать сравнительную оценку достоинств распределителя на основе SSA и на основе интервалов жизни максимального размера. Распределитель на основе SSA потенциально способен найти лучшую раскраску, чем традиционный распределитель, но делает он это на другом графе. Оба распределителя должны решать задачи о выборе выгружаемого значения и о месте размещения кода выгрузки, что может повлиять на производительность сильнее, чем само раскрашивание. Распределители применяют разные методы консолидации копирований. Для любого распределителя регистров детали имеют большое значение.

Распределение на основе линейных интервалов

Интервальный граф
граф, изображающий пересечения интервалов на прямой.

Интервальный граф интерференции имеет по одной вершине для каждого интервала и ребро между двумя вершинами, если их интервалы пересекаются.

Интервалы жизни, используемые при локальном распределении, образуют интервальный граф. Минимальную раскраску интервального графа можно вычислить за линейное время. Семейство распределителей с линейным сканированием основано на этом наблюдении; такие распределители эффективны с точки зрения времени, затраченного на компиляцию.

Распределители с линейным сканированием игнорируют управление потоком и рассматривают всю процедуру как линейный список операций. Распределитель представляет LR значения *v* как интервал [*i*, *j*], содержащий все операции, в которых *v* живо. То есть *i* меньше или равно порядковому номеру первой операции, в которой *v* живо, а *j* больше или равно порядковому номеру последней операции, в которой *v* живо. В результате граф интерференции оказывается интервальным графом.

Интервал [*i*, *j*] может содержать операции и блоки, которых не было бы в LR_v, построенном глобальным распределителем. Следовательно, он может давать завышенную оценку точного интервала жизни.

Для начала распределитель вычисляет информацию о живучести и строит множество интервалов для представления значений. Он сортирует интервалы по возрастанию порядкового номера их первой операции. В этот момент он применяет вариант локального алгоритма распределения из раздела 13.3. Значения распределяются по свободным регистрам, если это возможно; если доступных регистров нет, то распределитель выгружает LR, для которого последняя операция в интервале имеет наибольший порядковый номер.

Алгоритм линейного сканирования аппроксимирует поведение локального распределителя. Когда распределитель должен что-то выгрузить, он выбирает LR с наибольшим расстоянием до конца интервала (а не до следующего использования). Используется дисциплина выгрузки везде. Несомненно, эти изменения влияют на распределение, но как именно – не ясно.

Распределитель с линейным сканированием может консолидировать операцию копирования, которая является одновременно концом одного LR и началом другого. Эта эвристика объединяет меньше LR, чем мог бы консолидировать глобальный распределитель на основе раскрашивания, – неизбежный побочный эффект неявного и приближенного графа интерференции.

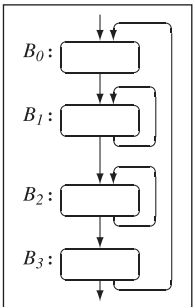
Расщепление интервала жизни – второе привлекательное дополнение к линейному сканированию. Разбиение длинных LR на более короткие может уменьшить величину MAXLIVE и дать распределителю возможность порождать распределения с меньшим объемом кода выгрузки. Чтобы реализовать расщепление интервала, автору компилятора понадобится эвристика для выбора подлежащих расщеплению LR и мест их расщепления. Без сомнения, выбор наилучшего множества расщеплений – трудная проблема.

Распределители с линейным сканированием – важный инструмент в арсенале автора компилятора. Эффективность делает их привлекательными в JIT-компиляторах (см. главу 14) и в небольших процедурах, где MAXLIVE < *k*. Если удается найти распределение процедуры без выгрузки, значит, оно достаточно хорошее.

Распределение на основе иерархического раскрашивания

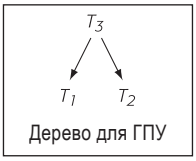
Глобальный распределитель либо назначает LR регистр на всю его жизнь, либо выгружает этот LR в каждой точке его определения и использования. Иерархический распределитель заимствует идеи расщепления интервала

жизни и включает их в способ работы с этим интервалами. Модификации позволяют распределителю в какой-то мере управлять гранулярностью и местом выгрузки.



ГПУ с вложенным циклом

В этой схеме распределитель накладывает иерархическую модель на вершины ГПУ. В модели *плитка* представляет одну или несколько вершин ГПУ и поток между ними. Плитки выбираются, так чтобы инкапсулировать циклы. В ГПУ, показанном во врезке, плитка T_1 состоит из $\{B_1\}$. Плитка T_2 состоит из $\{B_2\}$. Плитки могут быть вложенными; так, плитка T_3 содержит $\{B_0, T_1, T_2, B_3\}$. Дерево плиток во врезке улавливает эту связь; T_1 и T_2 – братья и в то же время прямые потомки T_3.



Дерево для ГПУ

Иерархический распределитель выполняет анализ потока управления с целью найти циклы и сгруппировать блоки в плитки. Чтобы предоставить конкретную реализацию вложенности плиток, он строит *дерево плиток*, в котором подплитки являются сыновними узлами содержащей их плитки.

Затем иерархический распределитель обходит дерево плиток снизу вверх. В каждой плитке T_i он строит граф интерференции для этой плитки, производит консолидацию, пытается раскрасить граф и вставляет в нужные места код выгрузки. Покончив с T_i, распределитель строит сводную плитку для представления T_i в процессе распределения родителя T_i. Сводная плитка включает свойства LiveIn и LiveOut представляемого ей региона, а также обобщенные детали распределения в регионе – количество распределенных регистров и предпочтения физическим регистрам.

После того как все плитки по отдельности раскрашены, распределитель производит нисходящий проход по дереву плиток, чтобы выполнить назначение, т. е. отобразить распределение на PR. Этот проход повторяет базовую форму глобального распределителя, но уделяет особое внимание значениям, которые остаются живыми при пересечении границы плиток.

На восходящем проходе распределения LR обнаруживаются по одной плитке за раз. Этот процесс расщепляет значения, оставшиеся живыми при пересечении границы плиток; распределитель вставляет для этих расщеплений операции копирования. Точки расщепления изолируют решения о выгрузке внутри плитки от спроса на регистры вне плитки, и общая тенденция такова, что места выгрузки концентрируются на границах плиток с высоким спросом.

Конечно, распределитель мог бы применить проход консолидации к получившемуся после распределения коду.

Межплиточные соединения между интервалами жизни становятся операциями копирования. Распределитель применяет механизм предпочтений, напоминающий смещенное раскрашивание, чтобы удалить эти операции там, где это имеет смысл (см. раздел 13.5.1). Тот же механизм позволяет распределителю моделировать требования к конкретному PR.

Эксперименты показывают, что иерархический распределитель с его более короткими интервалами жизни порождает чуть лучшие распределения,

чем прямолинейная реализация глобального распределителя на основе раскрашивания. Те же самые измерения показали, что сам распределитель потреблял больше времени на этапе компиляции, чем эталонный глобальный распределитель на основе раскрашивания. Дополнительные накладные расходы на повторяющиеся шаги распределения, похоже, перевешивают асимптотические преимущества построения графов меньшего размера.

13.6. Итоги и перспективы

Поскольку распределение регистров – важная часть современного компилятора, ему уделено много внимания в литературе. И для локального, и для глобального распределения существуют эффективные методы. Поскольку многие задачи являются NP-трудными, их решения оказываются чувствительными к небольшим нюансам алгоритмов, например к способу разрешения неоднозначности при выборе из вариантов с одинаковым рангом.

Прогресс в области распределения регистров связан с использованием парадигм, подразумевающих интеллектуальные подходы к проблеме. Так, распределители на основе раскрашивания графа обрели популярность не потому, что распределение регистров идентично раскрашиванию графа, а потому, что раскрашивание улавливает некоторые критические аспекты задачи глобального распределения. На самом деле многие улучшения распределителей на основе раскрашивания достигнуты в результате концентрации усилий на тех местах, где парадигма раскрашивания неточно отражает исходную проблему, например построения улучшенных моделей стоимости и методов расщепления интервалов жизни. По существу, эти улучшения приблизили парадигму к реальной проблеме.

Замечания к этой главе

Распределение регистров восходит к самым первым компиляторам. Бэкус сообщает, что Бэст придумал алгоритм из раздела 13.3 в середине 1950-х годов для оригинального компилятора FORTRAN [27, 28]. Алгоритм Бэста много раз переоткрывался в разных контекстах [39, 127, 191, 254]. Особенно хорошо известна его инкарнация в виде офлайнового алгоритма замещения страниц Белади MIN [39]. Горвиц [208] и Кеннеди [225] описывают осложнения, возникающие из-за чистых и грязных значений. Либераторе и др. описывает выгрузку чистых значений раньше грязных в качестве компромисса [254].

Связь между раскрашиванием графа и задачами распределения памяти была установлена Лавровым [250] в 1961 году; в проекте Alpha раскрашивание использовалось для упаковки данных в память [151, 152]. Шварц описывает ранние алгоритмы Ершова и Коке [320], в которых основное внимание обращено на уменьшение количества цветов, а выгрузка игнорируется. Пер-

вый полный распределитель на основе раскрашивания графа был разработан Хайтиным и др. для компилятора PL.8 компании IBM [80–82].

Глобальный распределитель, описанный в разделе 13.4, следует плану Хайтина с модификациями Бриггса [57, 58, 62]. В нем используется данное Хайтиным определение интерференции и алгоритмы построения графа интерференции, консолидации и обработки выгрузок. Бриггс добавил основанный на SSA алгоритм построения интервалов жизни, улучшенную эвристику раскрашивания и несколько схем расщепления интервалов жизни [57].

Обработка классов регистров заимствована из работы Смита, Рамси и Холлоуэя [331]. Хайтин, Никерсон и Бриггс обсуждают достижение тех же целей посредством добавления ребер в граф интерференции для моделирования конкретных ограничений на назначение [60, 82, 284].

Идея раскрашивания непересекающихся подграфов независимо следует из работы Смита, Рамси и Холлоуэя. Ранее Гупта, Соффа и Стил предложили разбивать граф на независимые подграфы с помощью разделителей клик [184], а Харви предложил разделять его на части для регистров общего назначения и регистров с плавающей точкой [111].

В литературе и на практике имеется много улучшения базового алгоритма Хайтина–Бриггса. К ним относятся более сильные методы консолидации [168, 289], улучшенные методы выгрузки [40, 41], прочистка регистров [193], пересчет простых значений [61] и расщепление интервалов жизни [107, 116, 244]. Продвижение регистров было предложено как преобразование, предшествующее распределению, которое переписывает код с целью увеличить множество значений, допускающих хранение в регистре [73, 77, 258, 261, 315]. Дасгупта предложил инкрементное обновление для консолидании и выгрузки, а также более быстрое обновление, хотя и с некоторой потерей информации [124]. Харви рассматривал раскрашивание мест выгрузки с целью ослабить требования к памяти со стороны выгрузки [193].

Распределители на основе SSA были независимо разработаны несколькими авторами [64, 186, 292]. И Хак, и Бучез положили в основу оригинальное наблюдение и доработали его [53, 185]. Распределение с линейным сканированием предложено Полетто и Саркаром [296]. Иерархическая схема раскрашивания предложена Кобленцом и Каллахан [75, 106].

Упражнения

К разделу 13.3

1. Примените алгоритм локального распределения к следующему простому ILOC-блоку. Можете предполагать, что r_{arp} и r_i живы при входе в блок и что в целевой машине имеется четыре физических регистра.

```
loadAI  r_arp,12 ⇒ r_a
loadAI  r_arp,16 ⇒ r_b
add     r_i,r_a  ⇒ r_c
```

```
sub     r_b,r_i   ⇒ r_d
mult    r_c,r_d   ⇒ r_e
multI   r_b,2     ⇒ r_f
add     r_e,r_f   ⇒ r_g
storeAI r_g       ⇒ r_arp,8
```

2. Рассмотрите граф потока управления на рис. 13.12. Можете предполагать, что чтение возвращает значение с внешнего носителя, а запись передает значение на внешний носитель.

a. Вычислите множества LiveIn и LiveOut для каждого блока.

b. В предположении, что имеется три физических регистра, примените локальный алгоритм распределения к блокам B_0, B_1 и B_3. Если в блоке b определено имя n и $n \in \text{LiveOut}(b)$, то распределитель должен снова сохранить n в памяти, чтобы его значение было доступно в последующих блоках. Аналогично, если в блоке b имя n используется раньше локального определения n, то распределитель должен загрузить значение n из памяти. Предъявите получившийся код, включая все операции загрузки и сохранения.

c. Предложите схему, которая позволила бы оставлять некоторые значения из LiveOut(B_0) в регистрах, избежав тем самым их начальной загрузки в последующих блоках.

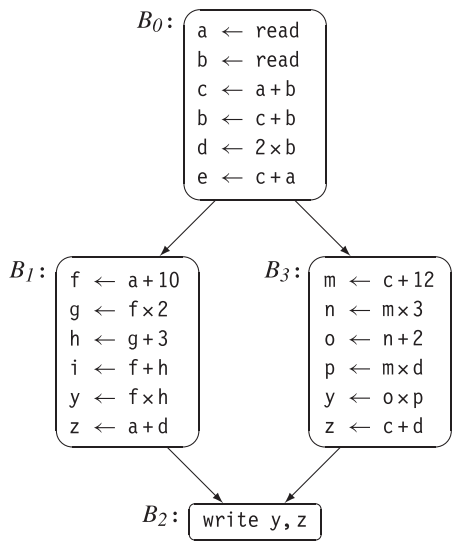


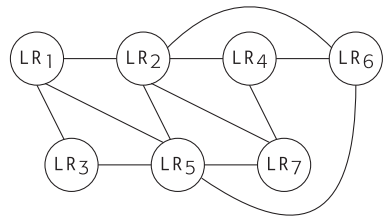
Рис. 13.12 Граф потока управления для упражнения 2

3. Цикл *Построение–Консолидация* занимает подавляющую долю времени работы глобального распределителя. Один из способов уменьшить эту долю – построить граф интерференции для *Консолидации*, который включает только LR, участвующие в одной или более операциях копирования.

a. Напишите набросок метода построения этого упрощенного графа.
b. Обсудите плюсы и минусы в предположении, что построение графа занимает время $O(n^2)$, где n – число вершин графа. Можете предполагать, что консолидация уменьшает количество интервалов жизни.

К разделу 13.4

4. Рассмотрим следующий граф интерференции:



Предположим, что целевая машина имеет три физических регистра и что для каждого интервала жизни оценка стоимости выгрузки равна 10.

a. Примените к этому графу алгоритм распределения из раздела 13.4. Какие интервалы жизни выгружаются? Какие раскрашены?
b. Играет ли роль выбор вершины для выгрузки?
c. В распределителе Хайтина использовался не такой подход к *Simplify* и *Select*, как в распределителе из раздела 13.4. В его распределителе *Simplify* удаляла вершины из графа таким же образом и в таком же порядке, как в глобальном распределителе. *Simplify* строила список вершин, которые были удалены вследствие эвристики выгрузки. В конце *Simplify*, если этот «список выгрузки» оказывался пустым, распределитель сразу вызывал алгоритм *Вставить выгрузки*, чтобы выгрузить все присутствующие в списке вершины. Затем она перезапускала процесс распределения для модифицированного таким образом кода.

Что происходит, когда алгоритм Хайтина применяется к примеру графа интерференции выше? Порождаются ли в результате такого изменения алгоритма те же или другие результаты?

5. Исследование кода после распределения регистров может обнаружить участки кода, где некоторые регистры свободны. В глобальном распределителе на основе раскрашивания это происходит из-за недостатков в способе выгрузки интервалов жизни.
a. Объясните, как может возникнуть такая ситуация.
b. Как компилятор мог бы обнаружить, возникла ли такая ситуация и где именно.
c. Что можно было бы сделать с этими неиспользуемыми регистрами внутри глобального распределителя и вне его?

6. Когда глобальный распределитель обнаруживает, что цвета для интервала жизни LRi не осталось, он выгружает или расщепляет этот интервал. Вместо этого он мог бы попробовать перекрасить одного или несколь-

ких соседей LR_i. Рассмотрим случай, когда $(LR_i, LR_j) \in I$ и $(LR_i, LR_k) \in I$, но $(LR_j, LR_k) \notin I$. Если LR_j и LR_k уже раскрашены в разные цвета, то распределитель, возможно, смог бы перекрасить один из них в цвет другого, освободив цвет для LR_i.

a. Разработайте набросок алгоритма, который обнаруживает, существует ли допустимое и продуктивное перекрашивание LR_i.

b. Как ваша схема влияет на асимптотическую сложность распределителя регистров?

c. Если распределитель не может перекрасить LR_k в цвет LR_j, потому что один из соседей LR_k уже окрашен в этот цвет, то следует ли распределителю рассмотреть возможность рекурсивного перекрашивания соседей LR_k? Объясните свой ответ.

7. В описании глобального распределителя предлагается вставлять код выгрузки для любого определения и использования в выгруженном интервале жизни.

Если в заданном блоке имеется один или несколько свободных регистров, то многократная выгрузка интервала жизни в этом блоке – пустая трата времени. Предложите улучшение механизма выгрузки в глобальном распределителе, которое позволило бы избежать данной проблемы.

8. Рассмотрите процедуру, которая содержит единственный блок с 10 000 операций и максимальным спросом на регистры $k + 15$.

a. Сравните поведение выгрузки в локальном и глобальном распределителях для такого блока.

b. Как вы думаете, какой распределитель породит код выгрузки меньшего размера?

Глава 14

Оптимизация на этапе выполнения

АННОТАЦИЯ

Оптимизация на этапе выполнения стала важной техникой реализации многих языков программирования. Переход от оптимизации на этапе компиляции к оптимизации на этапе выполнения позволяет языковой среде выполнения и компиляторам воспользоваться фактами, которые до этапа выполнения неизвестны. Если эти факты открывают возможность для специализации кода, например свернуть инвариантное значение, избежать кода преобразования типов или заменить виртуальный вызов функции прямым, то выгода от использования информации, доступной во время выполнения, может оказаться значительной.

В этой главе рассматриваются основные проблемы и компромиссы, возникающие при проектировании и реализации оптимизатора времени выполнения. Описывается работа оптимизатора горячих трасс и оптимизатора на уровне методов; оба добились успеха в реальных системах.

КЛЮЧЕВЫЕ СЛОВА

Компиляция на этапе выполнения, JIT-компиляция, динамическая оптимизация

14.1. Введение

Во многих языках программирования имеются средства, для которых трудно сгенерировать высококачественный код на этапе компиляции. Это позднее связывание, динамическая загрузка объявлений и кода (классов в JAVA) и различные виды полиморфизма. Классический компилятор, иногда называемый *упреждающим* (*ahead-of-time* – AOT), умеет генерировать код для таких средств. Но во многих случаях он не располагает достаточной информацией, чтобы хорошо его оптимизировать. Поэтому AOT-компилятор

вынужден генерировать код общего вида, который будет работать в любой ситуации, а не специализированный код, который мог бы сгенерировать, имей он более точную информацию.

Для некоторых задач необходимая информация может быть доступна на этапе компоновки или во время загрузки класса в JAVA. Для других она неизвестна до этапа выполнения. В языке, где такая поздно появляющаяся информация способна оказать значительное влияние на производительность, система может откладывать оптимизацию или трансляцию до момента, когда будет достаточно данных для порождения эффективного кода.

Авторы компиляторов применяли эту стратегию – *оптимизацию на этапе выполнения*, или *своевременную компиляцию* (*just-in-time* – JIT), – в различных контекстах, начиная с ранних LISP-систем и кончая современными скриптовыми языками. Она использовалась для создания инструментов поиска по регулярному выражению и быстрых эмуляторов системы команд. В этой главе описываются технические проблемы, возникающие при трансляции и оптимизации на этапе выполнения, и показано, как успешные системы преодолевали некоторые из этих проблем.

JIT-компиляторы, без сомнения, самые активно используемые из всех созданных в информатике. Большинство веб-браузеров включают JIT-компиляторы для скриптовых языков, применяемых на веб-сайтах. Среды выполнения для таких языков, как JAVA, обычно включают JIT для компиляции часто выполняемого кода. Поскольку такие системы компилируют код при каждом запуске, они выполняют гораздо больше компиляций, чем традиционный AOT-компилятор.

Концептуальная дорожная карта

Классические AOT-компиляторы принимают все решения на основе фактов, которые выводят из исходного текста программы. Такие компиляторы могут генерировать весьма эффективный код для императивных языков с объявлениями. Однако в некоторых языках имеются средства, из-за которых компилятор не может получить важную информацию до момента выполнения. Это и динамическая типизация, и некоторые виды полиморфизма, и инфраструктура открытых классов.

Оптимизация на этапе выполнения включает фундаментальный компромисс между временем, затраченным на компиляцию, и качеством кода. Оптимизатор этапа выполнения исследует состояние программы для вывода более точной информации, а затем использует ее, чтобы специализировать исполняемый код. Таким образом, оптимизатор этапа выполнения, чтобы быть эффективным, должен выводить полезную информацию. Он должен повысить производительность на этапе выполнения настолько, чтобы компенсировать дополнительные затраты на оптимизацию и генерирование кода. Следовательно, у автора компилятора есть сильные стимулы использовать эффективные и широко применимые методы.

Несколько слов о времени

Оптимизация на этапе выполнения добавляет новый уровень сложности в наши рассуждения о времени. Эти методы работают как на этапе компиляции, так и на этапе выполнения и привносят затраты на компиляцию при каждом выполнении программы.

Этап JIT
Так мы называем время, в течение которого выполняется оптимизатор этапа выполнения или JIT-компилятор.

На концептуальном уровне различие между временем компиляции и временем выполнения остается. Оптимизатор этапа выполнения планирует поведение и структуры данных на этапе выполнения точно так же, как это сделал бы AOT-компилятор. Он генерирует код создания и обслуживания среды выполнения. Таким образом, действия на этапе JIT отличаются от действий на этапе выполнения. Все рассуждения о времени, встречающиеся в предыдущих главах, остаются в силе, пусть даже промежуток, в течение которого действия выполняются, сдвинулся.

Еще сильнее осложняет ситуацию тот факт, что в некоторых системах оптимизация на этапе выполнения опирается на интерпретатор, работающий в режиме выполнения по умолчанию. Такие системы интерпретируют код, пока не обнаружат участок, который следует откомпилировать. В этот момент они компилируют и оптимизируют код, а затем организуют дело так, чтобы при последующих выполнениях использовался откомпилированный код этого участка. В подобных системах интерпретация, JIT-компиляция и выполнение откомпилированного кода перемешаны.

Краткий обзор

Чтобы эффективно реализовать такие средства, как поздняя привязка имен к типам или классам, динамическая загрузка и компоновка кода и полиморфизм, авторы компиляторов обратились к оптимизации на этапе выполнения. Такой оптимизатор может исследовать состояние работающей программы и обнаружить информацию, которая была недоступна или не видна раньше.

Компиляция на этапе выполнения также предлагает естественные механизмы для работы с изменениями, внесенными в исходный код программы во время выполнения (см. раздел 14.5.4).

К моменту выполнения система в основном знает, какой код включен в исполняемый файл. Поздно связываемые имена уже разрешены. Память для структур данных выделена, так что их размеры известны. Объекты созданы и содержат всю информацию о классе. Пользуясь знаниями о состоянии программы во время выполнения, компилятор может специализировать код способами, недоступными AOT-компилятору.

Компиляция на этапе выполнения имеет долгую историю. В ранних LISP-системах Маккарти компиляция новых функций в платформенный код про-

изводилась на этапе выполнения. Построение Томпсона НКА из регулярного выражения было придумано, чтобы компилировать регулярное выражение в платформенный код в момент выполнения команды поиска для редактора QED, – это один из первых хорошо известных примеров компилятора, работавшего на этапе выполнения. В последующих системах эти методы применялись для разных целей: от реализации динамических объектно-ориентированных языков, таких как SMALLTALK-80, до эмуляции кода в интересах переносимости. Расцвет Всемирной сети был отчасти обусловлен широким использованием языков JAVA и JAVASCRIPT, эффективность которых зависела от компиляции на этапе выполнения.

Оптимизация на этапе выполнения представляет для автора компилятора новый набор проблем и возможностей. Время, проведенное внутри компилятора, увеличивает общее время работы программы, поэтому автор JIT-компилятора должен искать компромисс между затратами на JIT и ожидаемыми улучшениями. Методы, которые перемещают бремя вычислений с редко выполняемого, или холодного, кода, на часто выполняемый, или горячий, код, способны повысить выигрыш от оптимизации.

В нашей интерпретации термин JIT охватывает все оптимизаторы этапа выполнения, не важно, подается им исходный код, как в ранних LISP-системах Маккарти, какая-то высокоуровневая нотация, как в компиляторе регулярных выражений в машинный код Томпсона, некая промежуточная форма, как в JAVA-системах, или даже машинный код, как в Dynamo. Во врезках сообщаются сведения о некоторых таких системах, чтобы читатель мог познакомиться с долгой историей и разнообразными применениями этих идей.

Последствия JIT-компиляции

JIT-компиляция может сильно повлиять на скорость выполнения приложения. В качестве конкретного примера на рис. 14.1 показано время выполнения написанной на JAVA реализации алгоритма локального распределения регистров из раздела 13.3. На рисунке (a) показано время выполнения распределителя на последовательности блоков с 1000, 2000, 4000 и т. д. до 128 000 строк ILOC-кода. Серая линия с нанесенными квадратиками показывает время работы с выключенной JIT-компиляцией, черная линия с нанесенными треугольниками – с включенной. На рисунке (b) поведение на начальной стадии изображено в увеличенном масштабе, т. е. на меньших наборах данных.

Данные были собраны для версии OpenJDK 1.8.0_292, работающей на процессоре Intel ES2640 с тактовой частотой 2.4 ГГц

Во входном коде был равномерный спрос на регистры – 20 значений. Распределителю было выделено 15 регистров.

JIT-компиляция дает шестикратную разницу на самом большом наборе данных; это с лихвой компенсирует время, затраченное на саму компиляцию. На рисунке (a) показан вклад JIT в производительность кода, а на рисунке (b) – что эмуляция кода виртуальной машины быстрее на небольших

Эти числа относятся к данному конкретному приложению. У вас могут получиться другие результаты.

наборах данных. Время, потраченное на JIT-компиляцию, замедляет выполнение на ранних стадиях выполнения, но по истечении примерно половины секунды выигрыш откомпилированного кода в скорости начинает перевешивать затраты вследствие JIT.

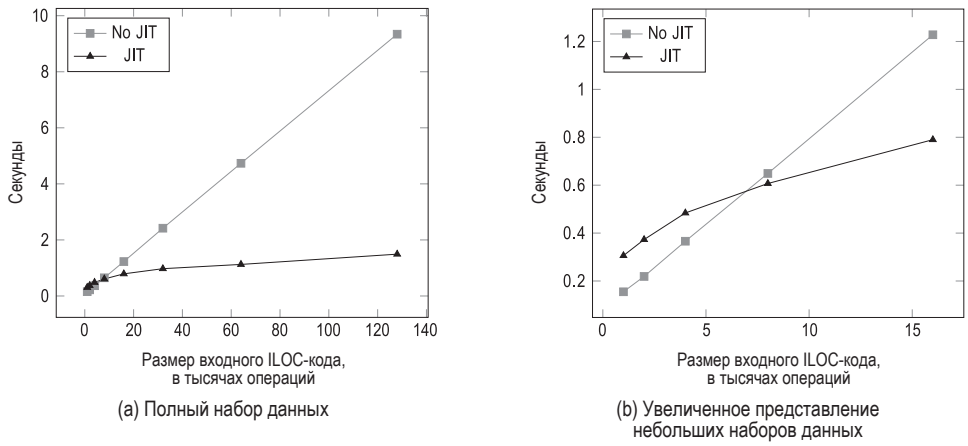


Рис. 14.1 Масштабируемость JAVA-приложения

Дорожная карта

Дизайн JIT-компилятора подразумевает фундаментальные компромиссы между объемом работы, выполняемой заранее, объемом работы, выполняемой JIT-компилятором, и улучшениями, достигаемыми за счет JIT. По мере развития языков, архитектур и техники выполнения акценты этих компромиссов смещались. Они и сейчас продолжают эволюционировать с накоплением сообществом опыта в построении и развертывании JIT-компиляторов. Наши методы и понимание почти наверняка улучшатся, но фундаментальный компромисс между эффективностью и результативностью останется.

В этой главе представлено состояние предмета на момент публикации. В разделе 14.2 описаны четыре основных вопроса, играющих важную роль в формировании структуры системы с JIT-компилятором. В двух следующих разделах приведены высокоуровневые описания двух JIT-компиляторов, занимающих разные точки в пространстве проектирования. В разделе 14.3 описан оптимизатор горячих трасс, а в разделе 14.4 – оптимизатор горячих методов; оба дизайна взяты из успешных систем. В разделе «Дополнительные темы» рассматривается еще несколько вопросов, возникающих при проектировании и конструировании практических систем на основе JIT.

14.2. Предварительные сведения

Оптимизация на этапе выполнения появилась как технология, позволяющая среде выполнения максимально близко подогнать исполняемый код к контексту, в котором он выполняется. В частности, отложив оптимизацию

до момента, когда компилятор будет располагать более полными знаниями о типах, константах и поведении во время выполнения (например, информацией из профиля), JIT-компилятор может устранить некоторые накладные расходы, сопряженные с такими языковыми средствами, как объектная ориентация, динамическая типизация и позднее связывание.

Однако для успешной оптимизации на этапе выполнения необходимо уделять внимание эффективности и результативности JIT-компиляции. Фундаментальный принцип оптимизации на этапе выполнения:

Оптимизатор этапа выполнения должен сэкономить больше тактов, чем использовал сам.

Если компилятор не удовлетворяет этому ограничению, то он не ускоряет, а замедляет работу приложения.

Это критическое ограничение определяет форму как JIT-компилятора, так и среды выполнения, с которой он взаимодействует. Эффективность самого компилятора вознаграждается. Поскольку теперь время компиляции складывается со временем выполнения, эффективность реализации JIT-компилятора напрямую влияет на время работы приложения. Важны как область применения, так и цели, поставленные перед JIT; существенны как асимптотическая сложность, так и фактические накладные расходы. Не менее важно и то, что принцип выбора участков кода, подлежащих оптимизации, оказывает прямое влияние на общую стоимость выполнения приложения.

Регулярные выражения в редакторе QED

Кен Томпсон встроил механизм поиска по регулярному выражению (РВ) в редактор QED в конце 1960-х годов. Эта команда стала одним из первых JIT-компиляторов, который вызывался по прямому указанию пользователя. Когда пользователь вводил РВ, редактор вызывал JIT-компилятор, который генерировал машинный код для IBM 7094. Затем редактор вызывал этот код для выполнения поиска. По завершении поиска код отбрасывался.

Это был JIT-компилятор. Он сначала разбирал РВ для проверки его синтаксиса. Затем он преобразовывал РВ в постфиксную нотацию. И наконец, генерировал машинный код выполнения поиска. JIT-кодогенератор, который применял этот метод, теперь называется построением Томпсона и используется для неявного создания НКА (см. раздел 2.4.2). Сгенерированный код эмулировал этот НКА. В нем использовался поиск, чтобы избежать дублирования термов, что могло бы вызвать экспоненциальный рост размера состояния на этапе выполнения.

Команда поиска стала важным средством в текстовом редакторе QED, работавшем на процессоре 0.35 MIP с оперативной памятью 32 КБ. Это раннее использование технологии JIT позволило создать быстрый инструмент для работы в среде с чрезвычайно ограниченными ресурсами.

Это ограничение также придает высокую значимость результативности алгоритмов, задействованных в JIT. Автор компилятора должен сосредоточиться на методах, которые, с одной стороны, широко применяются, а с дру-

гой – дают выигрыш в большинстве случаев. JIT-компилятор должен применять эти методы к регионам, где появление удобных возможностей вероятно, а улучшения окупаются сторицей. Удачный JIT-компилятор улучшает время выполнения кода достаточно часто, чтобы пользователи считали потраченное на JIT-компиляцию время выгодным вложением.

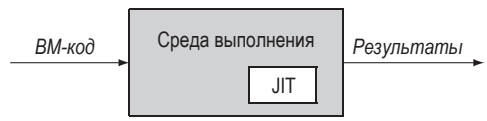
Эта ситуация отличается от имеющей место в AOT-компиляторе. Авторы компиляторов предполагают, что код, порожденный AOT-компилятором, выполняется в среднем много раз в расчете на одну компиляцию. Поэтому стоимость оптимизации не слишком существенна. AOT-компиляторы применяют множество преобразований от таких общих, как нумерация значений, до таких узкоспециальных, как снижение стоимости операций. Они применяют методы, дающие как много мелких улучшений, так и немного, но серьезных. Выигрыш AOT-компилятора обеспечивается накоплением улучшений в результате целого набора оптимизаций, используемых в каждой подходящей точке кода. Конечный пользователь не замечает стоимости компиляции и оптимизации.

С другой стороны, ограничения, в условия которых работает JIT-компилятор, означают, что его автор должен тщательно выбирать, какие преобразования имеет смысл реализовать, и применять к участкам, которые исполняются часто. На рис. 14.1 показано улучшение, которое JIT-компиляция дает в компиляторе HotSpot Server Compiler. В этом тесте HotSpot порождал существенные улучшения для участков кода, работавших дольше половины секунды. Пристальное внимание к затратам и результатам позволяет этому JIT-компилятору играть важнейшую роль в производительности JAVA на этапе выполнения.

14.2.1. Модель выполнения

Выбор модели выполнения оказывает большое влияние на форму системы оптимизации на этапе выполнения. Он влияет на скорость эталонного выполнения, на объем выполняемой системой компиляции и потому на накопительные накладные расходы на оптимизацию. Он также влияет на сложность реализации.

Система оптимизации на этапе выполнения может быть весьма сложной. На входе она принимает код для какой-то виртуальной машины (ВМ). Это может быть код для абстрактной машины, такой как ВМ JAVA (JVM) или ВМ Smalltalk-80. Но возможно также, что ВМ-код – это платформенный машинный код. На выходе оптимизатор порождает результаты выполнения программы.



Среда выполнения может порождать результаты путем выполнения машинного кода, путем интерпретации ВМ-кода или путем JIT-компиляции

Различие между этими режимами практически не видно пользователю.

ВМ-кода в машинный и его выполнения. Связь между JIT-компилятором, кодом и остальной средой выполнения определяет режим выполнения. Выполняет ли система по умолчанию машинный или ВМ-код? У любого варианта есть плюсы и минусы.

- Выполнение машинного кода обычно подразумевает JIT-компиляцию до выполнения, если только ВМ-код *уже* не является машинным.
- Выполнение ВМ-кода обычно подразумевает интерпретацию на каком-то уровне. Код компактный, он может быть более абстрактным, чем машинный.

Машинный код почти всегда быстрее кода ВМ. Он полагается на оборудование, которое реализует цикл выборки–декодирования–выполнения, тогда как в случае эмуляции ВМ-кода эти функции приходится выполнять программно. Более низкие затраты на одну операцию обращаются в значительный выигрыш в производительности для любого нетривиального выполнения.

Система на основе кода ВМ может отложить сканирование и разбор процедуры до момента ее вызова. Экономия времени запуска может оказаться заметной.

С другой стороны, для систем на основе ВМ-кода ниже затраты на запуск, поскольку система не нуждается в компиляции перед началом выполнения приложения. Поэтому короткие программы выполняются быстрее, как показано на рис. 14.1(b). Для коротких или редко выполняемых процедур эмуляция ВМ-кода может обойтись дешевле, чем JIT-компиляция плюс выполнение машинного кода.

В системе SMALLTALK-80 Дойча–Шиффмана использовалось три формата записи активации. Она транслировала один формат в другой в зависимости от того, обращался ли код к ЗА как к данным или нет.

Внедрение JIT-компиляции в систему на основе ВМ-кода обычно приводит к созданию смешанной платформы, которая выполняет и ВМ-код, и машинный код. В смешанной системе бывает необходимо представлять критические структуры данных, например записи активации, одновременно в формате, специфицированном для ВМ, и в формате, поддерживаемом платформенной ISA. Двойственные представления могут потребовать трансляции структур ВМ-кода в платформенные структуры, а эти трансляции, в свою очередь, влекут затраты на этапе выполнения.

Альтернатива, выполнение машинного кода, распределяет затраты другим способом. Такая система должна откомпилировать весь код ВМ в машинный – либо заранее, либо на этапе выполнения. AOT-компиляция приводит к коду общего вида, а значит, к дорогому неоптимизированному платформенному выполнению. JIT-компиляция ведет к системе, которая выполняет больше компиляции на этапе выполнения и порождает эти затраты при каждом выполнении программы.

Не существует проектного решения, подходящего для всех случаев. Автор компилятора должен тщательно взвесить разнообразные компромиссы и реа-

лизовать систему, памятуя об эффективности и результативности. В этом пространстве проектирования было создано несколько удачных систем.

Adaptive Fortran

Adaptive Fortran – оптимизатор этапа выполнения для FORTRAN IV, созданный Хансеном и вошедший в его диссертацию 1974 года. Он использовал его для исследований практичности и целесообразности оптимизации на этапе выполнения. В Adaptive Fortran были впервые применены многие идеи, вошедшие в современные системы.

В системе использовался быстрый AOT-компилятор, который генерировал промежуточное представление (ПП) программы, состоявшее из простых блоков. На этапе выполнения ПП интерпретировалось до тех пор, пока счетчик выполнений не начинал показывать, что оптимизация блока могла бы дать выигрыш. (AOT-компилятор вычислял зависящие от блока и оптимизации пороги, основанные на длине блока, глубине вложенности и стоимости JIT-оптимизации.)

Руководствуясь счетчиками выполнений и порогами, супервизор вызывал JIT-оптимизатор для оптимизации блоков и окружающего их контекста. Использование нескольких порогов для блока позволяло производить прогрессивную оптимизацию – к блокам, на которые приходилась бóльшая доля времени выполнения, применялось больше оптимизаций.

Одна из ключевых оптимизаций, которую Хансен называл *слиянием*, агрегировала несколько блоков, чтобы сгруппировать циклы и вложенные циклы в *сегменты*. Эта стратегия позволяла Adaptive Fortran применять основанные на циклах оптимизации, например перемещение кода.

14.2.2. Триггеры компиляции

Среда выполнения должна решать, когда и где вызывать JIT-компилятор. Это решение сильно влияет на общую производительность, потому что от него зависит, как часто будет запускаться JIT-компилятор и на чем он должен сосредоточить усилия.

Оптимизаторы этапа компиляции используют JIT-компиляцию разными способами. Если система JIT-компилирует *весь* код, как бывает в некоторых системах с машинным кодом, то триггер может быть совсем простым, например «компилируй каждую процедуру перед ее первым вызовом». Если же система компилирует только горячий код, то триггеру могут понадобиться данные профилирования на уровне процедуры или блока. В средах с платформенным кодом и в смешанных средах для сбора данных профилирования могут применяться разные механизмы.

В среде с платформенным кодом автор компилятора должен выбирать между (1) системой, которая работает с BM-кодом и компилирует его в машинный перед выполнением, и (2) системой, которая работает с AOT-компилированным машинным кодом и вызывает JIT-компиляцию только для часто выполняемого, или горячего, кода. Эти два подхода сталкиваются с совершенно разными проблемами.

Выполнение ВМ-кода

В смешанной среде система может начать выполнение немедленно и собрать данные профилирования, чтобы определить, когда JIT-компилировать код в машинный. Обычно такие системы запускают компиляцию, когда данные профилирования показывают, что превышен некоторый порог, за которым код считается горячим. Это помогает системе не тратить время на JIT-компиляцию кода, который мало или совсем не влияет на производительность.

Пороговые значения играют важную роль в определении общего характера выполнения. Если порог велик, то число JIT-компиляций уменьшается. Но в то же время возрастает доля времени, проведенного в эмуляторе ВМ-кода, который обычно выполняется медленнее, чем машинный код. Изменение пороговых значений меняет поведение системы.

Обратный переход
В этом контексте обратный условный или безусловный переход – это переход на адрес, меньший указателя программы.

Условные переходы, замыкающие цикл, обычно являются обратными.

Чтобы собрать точные данные профилирования, среда выполнения ВМ-кода может оснастить измерительными средствами условные и безусловные переходы в приложении. Чтобы уменьшить накладные расходы на сбор профиля, эти системы часто ограничивают множество точек сбора данных. Например, блоки, являющиеся целями обратного условного перехода, – хорошие кандидаты на профилирование, потому что, вероятно, являются заголовками циклов. Аналогично блок, начинающий пролог процедуры, – очевидный кандидат на профилирование. Система может собрать данные о местах вызова, оснастив предварительные последовательности. Все эти и другие метрики нашли применение в практических эффективных системах.

Выполнение платформенного кода

Если система выполняет платформенный код, то она должна откомпилировать каждую процедуру, прежде чем код можно будет выполнить. Система может запустить компиляцию в момент загрузки либо пакетно для всего выполняемого файла (Speed Doubler), либо по мере загрузки модулей (ранние версии системы V8 для JAVASCRIPT). Альтернативно система может запускать компилятор для трансляции каждой процедуры при ее первом выполнении. Чтобы добиться такого эффекта, система может прикомпоновать заглушку вместо всех еще только подлежащих компиляции процедур; заглушка находит код ВМ для вызываемой стороны, JIT-компилирует и компонует его, а затем выполняет вызов заново.

Стратегии на этапе загрузки вынуждены JIT-компилировать каждую процедуру, даже если она никогда не выполняется. Любая задержка на этой стадии прибавляется ко времени запуска приложения. Стратегия компиляции при вызове сдвигает стоимость начальной компиляции на более поздние стадии выполнения. Она позволяет избежать компиляции кода, который

никогда не выполняется, однако компилирует любой исполняемый код вне зависимости от того, холодный он или горячий.

Speed Doubler

Speed Doubler – коммерческий продукт, разработанный компанией Connectix в 1990-х годах. В нем использовалась компиляция на этапе загрузки, чтобы перенести унаследованные приложения на новую ISA. Как раз тогда Apple перевела свою линейку компьютеров Macintosh с процессоров Motorola MC 68000 на IBM POWER PC. Поддержка унаследованных приложений осуществлялась с помощью эмулятора, встроенного в MacOS.

Speed Doubler представлял собой JIT-компилятор этапа загрузки, который транслировал приложения для MC 68000 в машинный код POWER PC. За счет устранения накладных расходов на эмуляцию он давал значительное ускорение. После установки он размещался между загрузчиком ОС и началом исполнения приложения. Он быстро производил трансляцию, после чего передавал управление коду запуска приложения.

Первая версия Speed Doubler, похоже, выполняла покомандную трансляцию, и улучшение уже было достаточным, чтобы оправдать название (удвоитель скорости). В последующих версиях производительность повысилась; мы полагаем, что это стало результатом улучшенной оптимизации и генерирования кода.

В Speed Doubler использовалось выполнение платформенного кода с дисциплиной компиляции во время загрузки, чтобы обеспечить простой и прозрачный механизм улучшения времени выполнения. У пользователей складывалось впечатление, что затраты на JIT-компиляцию с лихвой окупаются достигнутым ускорением, так что продукт пользовался успехом.

Уменьшение времени JIT-компиляции уменьшает общее время выполнения.

Если система начинает работу с кода, сгенерированного AOT-компилятором, то может избежать этих компиляций в момент запуска. AOT-компилятор может вставить код, необходимый для сбора данных профилирования. Он также может аннотировать код информацией, способной помочь при последующих JIT-компиляциях. Системе, в которой используется заранее откомпилированный машинный код, нужно лишь запустить оптимизатор, когда она обнаруживает, что некоторый фрагмент кода горячий, т. е. потребляет достаточно времени, чтобы затраты на JIT-компиляцию были оправданы.

14.2.3. Гранулярность оптимизации

Оптимизаторы этапа выполнения работают с разной гранулярностью. Это решение, принимаемое на этапе проектирования, оказывает сильное влияние на общую эффективность, потому что определяет, какие виды оптимизаций может применить JIT-компилятор, а также размер фрагментов кода, подвергаемых оптимизации. Особенно широкое распространение получили две области оптимизации.

Трасса (повторно)
Трассой называется ациклическая последовательность блоков. В контексте оптимизации на этапе выполнения трасса может проходить через вызовы процедур.

Горячие трассы. Оптимизатор трасс наблюдает за условными и безусловными переходами во время выполнения, чтобы найти горячие трассы. После того как счетчик выполнений трассы превысил заранее заданное пороговое значение, система вызывает JIT-компилятор для построения оптимизированного машинного кода трассы.

Оптимизаторы трасс выполняют локальную или региональную оптимизацию горячей трассы с последующим генерированием платформенного кода – включая распределение регистров и планирование команд. Поскольку трасса может включать вызовы и возвраты, такая «региональная» оптимизация может вносить улучшения, которые AOT-компилятором были бы расценены как межпроцедурные.

Горячие методы. Оптимизатор методов находит процедуры, на которые приходится значительная доля общего времени выполнения, наблюдая за различными счетчиками. К ним относятся счетчики вызовов, встроенные в код пролога, счетчики итераций цикла, модифицируемые перед обратным переходом, и данные о местах вызова, собираемые в предварительных последовательностях. После того как метод признан горячим, система вызывает JIT-компилятор, который генерирует оптимизированный машинный код метода.

Поскольку это делается для всего метода, оптимизатор может произвести нелокальные оптимизации, такие как перемещение кода, региональное планирование команд, устранение мертвого кода, глобальное устранение избыточностей или снижение стоимости операций. Некоторые оптимизаторы методов выполняют также встраивание. Они могли встроить код часто исполняемого вызова в горячем методе. Если большинство обращений к горячему методу производится из одного места, то оптимизатор мог бы встроить вызываемую сторону в это место.

Выбор уровня гранулярности оказывает большое влияние на стоимость оптимизаций и на возможности, которые способен обнаружить оптимизатор.

Предполагается, что трасса имеет одну точку входа, но может иметь преждевременные выходы

- Оптимизатор трасс мог бы применить ЛНЗ ко всей трассе, чтобы найти избыточности, свернуть константы и упростить тождества. Большинство оптимизаторов методов используют глобальный алгоритм устранения избыточностей, который дороже, но обычно находит больше возможностей для улучшения.

Линейное сканирование дает некоторые преимущества глобального распределения с меньшими затратами, чем раскрашивание графа.

- Оптимизатор трасс мог бы применить быстрый локальный распределитель регистров по типу алгоритма из раздела 13.3. С другой стороны, оптимизатор методов вынужден работать с потоком управления, поэтому нуждается в глобальном

распределителе регистров, например на основе раскрашивания или линейного сканирования (см. разделы 13.4 и 13.5.3).

Компромисс опять же сводится к выбору между стоимостью оптимизации и общим улучшением на этапе выполнения.

14.2.4. Источники улучшения

JIT-компилятор может обнаружить факты, неизвестные до этапа выполнения, и воспользоваться ими для обоснования оптимизации. К этим фактам могут относиться информация из профиля, типы объектов, размеры структур данных, границы циклов, константные значения или типы и другие системно-зависимые сведения. В той мере, в какой эти факты открывают возможность оптимизации, которая не могла быть произведена AOT-компилятором, они помогают обосновать компиляцию на этапе выполнения.

Система SMALLTALK-80 Дойча–Шиффмана

В реализации SMALLTALK-80 Дойча–Шиффмана использовалась JIT-компиляция для создания платформенной среды в системе с процессором серии Motorola MC 68000. Smalltalk-80 распространялся в виде образа для виртуальной машины Smalltalk-80.

Эта система исполняла только машинный код. Механизм поиска и диспетчеризации методов вызывал JIT-компилятор для любого метода, который еще не имел тела на машинном коде; это дисциплина компиляции при вызове.

Ускорение работы системы было обусловлено главным образом заменой эмуляции ВМ выполнением машинного кода. В системе использовался глобальный кеш методов, и она была первой, в которой применялись встроенные кеши методов. Авторы очень внимательно подошли к трансляции структур ВМ в платформенные структуры, особенно это касалось записей активации. В результате получилась система, поражавшая своим быстродействием по сравнению с другими существовавшими на тот момент реализациями Smalltalk-80 для стандартного оборудования.

Система работала в среде с небольшим объемом памяти (16 МБ в то время считалось большой памятью). Поскольку размер платформенного кода превышал размер ВМ-кода в 2–5 раз, система тщательно управляла пространством кода. Когда нужно было освободить место в пространстве кода, система отбрасывала машинный код, а не выгружала его на диск. Эта стратегия, которую иногда называют *генерированием кода с отбрасыванием*, была выгодна из-за больших различий в производительности между эмуляцией ВМ и исполнением машинного кода и между JIT-компиляцией и выгрузкой на удаленный диск (по сети Ethernet со скоростью 10 Мбит/с).

На практике оптимизаторы этапа выполнения находят возможности для улучшения разными способами, например:

- *устранение накладных расходов ВМ.* Если JIT-компилятор работает в смешанном ре-

В системе Дойча–Шиффмана машинный код работал достаточно быстро, что компенсировало затраты на JIT-компиляцию.

жиме, то трансляция в машинный код уменьшает затраты на эмуляцию. Машинный код заменяет программную эмуляцию аппаратным исполнением, которое почти всегда быстрее.

Некоторые ранние JIT-компиляторы, например компилятор регулярных выражений Томпсона в редакторе QED, производили лишь минимальную оптимизацию. Их преимущества почти полностью были связаны с устранением накладных расходов ВМ;

- *улучшение размещения кода в памяти.* Оптимизатор трасс естественным образом достигает улучшений за счет размещения кода в памяти. Создавая копию горячей трассы, JIT-компилятор размещает блоки последовательно в порядке выполнения, что дает некоторые преимущества, которые обычно относят на счет глобального размещения кода (см. раздел 8.6.2).

  В откомпилированной копии горячей трассы JIT-компилятор может заставить путь вдоль трассы проходить по неявным ветвям всех условных переходов. В то же время любой безусловный переход в конце блока на трассе становится переходом на следующую операцию, поэтому JIT-компилятор может просто удалить их;

- *устранение избыточности.* Большинство JIT-компиляторов устраняют избыточность. Оптимизатор трасс может применить алгоритмы ЛНЗ или СНЗ, которые заодно производят распространение констант и упрощение алгебраических выражений. Сложность обоих алгоритмов составляет O(1) на одну операцию.

В частности, система Dynamo получала преимущества от линеаризации трасс.

  Оптимизатор методов может применить алгоритм НЗД или такие методы обработки потока данных, как отложенное перемещение кода или глобальная нумерация значений, чтобы достичь аналогичных результатов. Стоимость этих алгоритмов может меняться, как и состав конкретных обнаруживаемых ими возможностей (см. разделы 10.6 и 10.3.1);

- *снижение накладных расходов на вызов.* Встраивание уменьшает накладные расходы на вызовы процедур. Оптимизатор этапа выполнения может использовать данные профилирования, чтобы выявить места вызова, где встраивание может принести пользу. Оптимизатор трасс может включить вызов или возврат прямо в трассу. Оптимизатор методов может встроить места вызова в горячий метод. Он также может использовать данные профилирования для принятия решения о том, стоит ли встраивать горячий метод в одну или несколько вызывающих его процедур;

- *адаптация кода к системе.* Поскольку результаты JIT-компиляции эфемерны – в конце выполнения они отбрасываются, – JIT-компилятор может оптимизировать код для конкретной модели процессора, на котором будет исполняться. Он может подстроить цикл

Нам неизвестен JIT-компилятор, который производил бы зависящую от модели процессора оптимизацию. Но для таких машинно-зависимых задач, как планирование команд, выигрыш мог бы быть значительным.

с большим количеством вычислений под имеющееся оборудование SIMD или GPU. Его планировщик может извлечь выгоду из такой информации о модели, как число функциональных блоков и их операционные задержки;

- *воспользоваться информацией, доступной во время выполнения*. Программы часто содержат вещи, о которых нельзя узнать до момента выполнения. Особый интерес представляют константные, т. е. неизменные, значения. Например, границы цикла могут быть привязаны к размеру структуры данных, прочитанной с внешнего носителя, которая читается один раз и больше во время выполнения не изменяется. JIT-компилятор может узнать эти значения и использовать их для улучшения кода. Например, он мог бы вынести проверки диапазона из цикла (см. раздел 7.3.3).

  AOT-компилятор мог бы идентифицировать значения, способные повлиять на JIT-оптимизацию, и включить методы для опроса этих значений.

  В языках с поздним связыванием информацию о типе и классе бывает трудно или невозможно установить на этапе AOT-компиляции. JIT-компилятор может использовать знания о типах и классах, доступные во время выполнения, и адаптировать откомпилированный код к обстоятельствам. В частности, он мог бы преобразовать общую диспетчеризацию метода в вызов метода конкретного класса.

Оптимизатор горячих трасс Dynamo

Система Dynamo – оптимизатор горячих трасс в машинном коде для системы PA-8000 компании Hewlett-Packard. В основу системы было положено фундаментальное предположение о том, что она может эффективно находить и улучшать часто выполняемые трассы, а код, работающий редко, исполнять в эмуляторе.

Для нахождения горячих трасс Dynamo подсчитывала, сколько раз выполнялись блоки, которые могли бы стать кандидатами на начало трассы. В случае превышения заранее заданного порога (50) JIT-компилятор строил трассу и оптимизировал ее. При последующих выполнениях трассы работал откомпилированный код. Система поддерживала собственный программно-управляемый кеш откомпилированных трасс.

Dynamo достигала улучшений в результате локальной и суперлокальной оптимизаций, повышения степени локальности кода, выпрямления ветвей и включения трасс в более крупные фрагменты. Трассы могли пересекать границы вызовов процедур, что позволяло Dynamo оптимизировать межпроцедурные трассы.

Dynamo доказала, что JIT-компиляция может быть целесообразной и даже конкурировать с кодом, оптимизированным AOT-компилятором. Последующая работа других авторов позволила создать Dynamo-подобную систему для архитектуры IA-32 ISA, получившую название DynamoRIO.

JIT-компиляция может налагать тонкие ограничения на оптимизацию. Например, традиционная AOT-оптимизация часто сосредоточена в циклах. Поэтому такие техники, как разворачивание цикла, перемещение инвари-

антного относительно цикла кода и снижение стоимости операций, доказали свою важность в AOT-модели. Оптимизаторам горячих трасс, которые исключают циклические пути, нелегко продублировать эти действия.

ILOC включает псевдооперацию `tbl` для фиксации и сохранения такого рода знаний.

Такие оптимизации потока управления, как разворачивание цикла и клонирование, обычно требуют графа потока управления. Реконструировать ГПУ по ассемблерному коду может быть трудно. Если в коде встречается операция *переход по регистру* (`jump` в ILOC), то узнать фактическую цель перехода может оказаться трудно или невозможно. В промежуточном представлении кода такие цели условных переходов можно переупорядочить и проанализировать. Даже при наличии операций *переход на метку* (`jumpI` в ILOC) оптимизация может запутать поток управления до такой степени, что его будет трудно или невозможно реконструировать. Например, на рис. 12.17 показан однотактовый программно конвейеризованный цикл, который начинается пятью операциями *перехода на метку*; реконструировать оригинальный цикл в ГПУ на рис. 12.17(b) – непростая задача.

14.2.5. Построение оптимизатора этапа выполнения

Построение JIT-компилятора – упражнение на программную инженерию. Здесь не требуется новых теорий или алгоритмов. Нужен лишь тщательный дизайн с упором на эффективность и результативность и реализация, которая ставит во главу угла минимизацию фактических затрат. Успех системы на основе JIT-компиляции будет зависеть от синергии отдельных проектных решений.

Далее в этой главе мы проиллюстрируем, какие компромиссы могут возникать при разработке оптимизатора этапа выполнения. Будут рассмотрены два сценария: оптимизатор горячих трасс в разделе 14.3 и оптимизатор горячих методов в разделе 14.4. Гипотетический оптимизатор горячих трасс многое заимствует из проекта системы Dynamo. Оптимизатор горячих методов черпает идеи в оригинальном компиляторе HotSpot Server Compiler и в системе SMALLTALK-80 Дойча–Шиффмана. Наконец, в разделе 14.5 мы на основе этих обсуждений рассмотрим некоторые более тонкие решения, которые приходится принимать проектировщику JIT-компилятора.

ИТОГИ РАЗДЕЛА

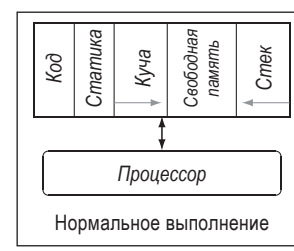
При проектировании JIT-компилятора разработчик должен ответить на несколько принципиальных вопросов. Он должен выбрать модель выполнения: будет ли система исполнять неоптимизированный код в эмуляторе или машинный код? Он должен определиться с гранулярностью компиляции: типичные варианты – компиляция трасс или целых процедур (либо методов). Он должен выбрать, как будет запускаться компиляция, т. е. при каких условиях система будет оптимизировать (или повторно оптимизировать) код. Наконец, автор компилятора должен понимать, на какие источники улучшения направить усилия, и выбрать те виды оптимизаций, которые помогут справиться с этими конкретными проблемами.

В процессе проектирования и реализации автор компилятора должен постоянно думать о том, окупится ли дополнительное время, потраченное на JIT-компиляцию, уменьшением времени работы кода. Каждое такое решение оказывает большое влияние на эффективность системы в целом и на время выполнения прикладной программы.

ВОПРОСЫ

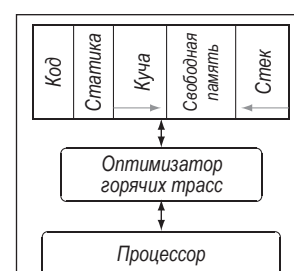
1. Как система, которая по умолчанию выполняет машинный код, может собрать необходимые ей данные профилирования? Как она могла бы предоставить эти данные JIT-компилятору?
2. Устранение накладных расходов на выполнение ВМ-кода потенциально является крупным источником улучшений. В каких ситуациях эмуляция могла бы оказаться более эффективной, чем JIT-компиляция в машинный код?

14.3. Оптимизация горячих трасс



Нормальное выполнение

В классической модели выполнения откомпилированного кода процессор читает операции и данные непосредственно из адресного пространства работающего процесса. Рисунок во врезке, подписанный «Нормальное выполнение», изображает эту ситуацию (это упрощенный вариант рис. 5.15). В цикле выборка–декодирование–выполнение используется оборудование процессора.



Исполнение, опосредованное оптимизатором этапа выполнения

Концептуально оптимизатор трасс машинного кода расположен между адресным пространством исполняемого процесса и процессором. Он «мониторит» выполнение, пока не наберет «достаточно» контекста, чтобы определить, что какая-то часть кода является горячей и ее следует оптимизировать. В этот момент он оптимизирует код и гарантирует, что при будущих выполнениях оптимизированной последовательности будет работать оптимизированный, а не оригинальный код. Эта ситуация изображена во врезке.

Перед оптимизатором горячих трасс стоит трудная задача. Он должен улучшить трассы в достаточной мере, чтобы оправдать затраты на их поиск и компиляцию. Каждый такт, потраченный на JIT-компиляцию, должен быть компенсирован одним или несколькими тактами, выигранными благодаря оптимизации. Кроме того, если процесс замедляет выполнение холодного кода, то оптимизированный код должен восполнить и этот дефицит.

В этом разделе представлен дизайн оптимизатора горячих трасс. Он заимствован из системы Dynamo, созданной подразделением Hewlett-Packard Research примерно в 2000 году, и послужит как введением в возникающие

проблемы, так и конкретным примером исследования компромиссов в процессе проектирования.

Dynamo выполняла платформенный код посредством эмуляции, пока не идентифицировала горячую трассу. Эмулятор выполнял весь холодный код, собирая при этом данные профилирования и выявляя горячие трассы. Поэтому машинный код работал медленнее, чем если бы просто выполнялся оборудованием. В основе Dynamo лежало предположение о том, что улучшение, достигаемое в результате оптимизации горячих трасс, сможет окупить как накладные расходы на эмуляцию, так и затраты на JIT-компиляцию.

Эти проектные соображения поднимают несколько критических вопросов. Как система определяет, что такое трасса? Как она может найти эти трассы? Как она решает, что трасса является горячей? Где находятся оптимизированные трассы? Как эмулируемый холодный код связывается с горячим и наоборот?

Входные блоки трассы

В Dynamo профилирование трасс, их идентификация и связывание холодного кода с горячим основаны на понятии *входного блока трассы*. Каждая трасса начинается входным блоком. Входной блок трассы удовлетворяет одному из двух простых условий. Либо он является целью обратного условного или безусловного перехода, либо целью выхода из существующей откомпилированной трассы.

Первое условие выбирает блоки, которые, скорее всего, являются блоками заголовков циклов. Эти блоки можно определить путем сравнения адресов: если целевой адрес меньше текущего указателя команд (PC), то он принадлежит входному блоку трассы.

Второе условие выбирает блоки, которые могут представлять альтернативные пути в цикле. Любой побочный выход из трассы становится входным блоком трассы. JIT-компилятор находит эти блоки, когда компилирует горячую трассу.

Чтобы идентифицировать и профилировать трассы, оптимизатор трасс находит входные блоки трассы и подсчитывает, сколько раз они выполняются. Ограничив число профилируемых блоков, можно удерживать накладные расходы на низком уровне. Обнаружив входной блок, оптимизатор вносит его в таблицу входов, которая содержит счетчик выполнений и указатель на каждый блок. Это основная структура данных оптимизатора горячих трасс.

14.3.1. Поток выполнения

На рис. 14.2(а) представлен высокоуровневый алгоритм оптимизатора трасс. Комбинация таблицы входов и кеша трасс инкапсулирует текущее состояние системы. Алгоритм определяет, как выполнять блок, ориентируясь на присутствие или отсутствие блока в таблице входов, а также на значение его счетчика выполнений и указателя на его код.

```
L1: эмулировать код, пока не встретится выбранная ветвь
    условного перехода или безусловный переход

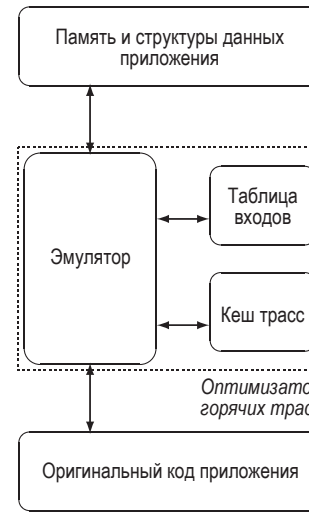
  искать целевой адрес в таблице входов
  if адреса нет в таблице входов then
    if адрес < эмулированного PC then
      создать в таблице входов запись с адресом
      положить счетчик выполнений для адреса равным 1

  else  // адрес имеется в таблице входов
    if фрагмент уже был откомпилирован then
      перейти на откомпилированный фрагмент

    увеличить счетчик для целевого адреса
    if счетчик > порога горячей трассы then
      построить, откомпилировать и выполнить трассу

  // если попали сюда, то блок работает в режиме эмуляции
  положить PC эмулятора равным целевому адресу
  перейти назад на L1
```

(a) Высокоуровневый алгоритм



(b) Компоненты оптимизатора трасс

Рис. 14.2 Концептуальная структура оптимизатора горячих трасс

Блоки исполняются в режиме эмуляции, пока не станут частью откомпилированной горячей трассы. Начиная с этого момента блок в составе трассы исполняется как часть оптимизированного кода. Если управление попадает в блок из другого пути, лежащего вне трассы, то блок исполняется в режиме эмуляции оригинального кода.

Критические решения принимаются, когда эмулятор встречает выбранную ветвь или безусловный переход. В этот момент эмулятор ищет целевой адрес в таблице входов.

- Если целевой адрес отсутствует в таблице входов и численно меньше текущего PC, то система классифицирует целевой адрес как входной блок трассы. Она создает запись в таблице входов и инициализирует счетчик выполнения значением 1. Затем она устанавливает PC эмулятора равным целевому адресу и продолжает выполнение с целевого блока.

  Меньший целевой адрес означает, что этот условный или безусловный переход является обратным.

- Если, напротив, целевой адрес уже присутствует в таблице входов и в этой записи хранится действительный указатель на код, то эмулятор передает управления откомпилированному фрагменту кода. В зависимости от реализации эмулятора, обсуждаемой ниже, эта передача может потребовать короткого переходника, похожего на предварительную последовательность в классическом соглашении о связях.

  Откомпилированная трасса хранится в кеше трасс.

  Каждый путь выхода из откомпилированной трассы либо связывается с другой откомпилированной трассой, либо заканчива-

  В разделе 14.3.2 обсуждается, как компилятор связывает откомпилированные трассы.

ется короткой заглушкой, которая устанавливает PC эмулятора равным адресу следующего блока и переходит обратно в эмулятор – на метку *L1* на рис. 14.2(а).

- Если целевой адрес присутствует в таблице, но еще не был откомпилирован, то система увеличивает его счетчик выполнений и сравнивает его с порогом горячей трассы. Если счетчик меньше или равен порогу, то система исполняет целевой блок в режиме эмуляции. Когда счетчик целевого адреса оказывается больше порога, система строит ПП-образ горячей трассы, исполняет и компилирует эту трассу, сохраняет код в кеше трасс, а указатель на ее код – в поле записи таблицы входов.

  При выходе из откомпилированной трассы выполнение продолжается со следующего блока. Код либо связывается с другой откомпилированной трассой, либо использует зависящую от пути заглушку, чтобы начать эмуляцию следующего блока.

Алгоритм на рис. 14.2(а) показывает эмуляцию блоков, перемежающуюся с выполнением оптимизированных трасс. Эмулятор переходит на оптимизированный код; оптимизированные трассы завершаются кодом, который устанавливает PC эмулятора и возвращается обратно в эмулятор. Далее в этом разделе детали описываются более подробно.

Эмуляция

Значения, находящиеся в памяти, могут располагаться по одинаковым адресам в обоих режимах выполнения.

Повторяя Dynamo, оптимизатор выполняет холодный код в режиме эмуляции. Автор JIT-компилятора мог бы реализовать эмулятор в виде полноценного интерпретатора с программным циклом выборки–декодирования–выполнения. При таком подходе понадобился бы эмулированный набор регистров и код для пересылки значений между эмулированными и физическими регистрами при переходах от эмулированного кода к откомпилированному и обратно. Этот переходный код может напоминать части стандартного соглашения о связях.

В качестве альтернативы система могла бы «эмулировать» выполнение путем прогона оригинального откомпилированного кода блока и перехвата управления на каждой выбранной ветви или безусловном переходе. Если аппаратная поддержка такого перехвата недоступна, то система может сохранить оригинальную операцию и заменить ее недопустимой командой – такой прием применялся в отладчиках начиная с 1960-х годов.

Когда PC доходит до конца блока, недопустимая команда вызывает исключение. Далее обработчик исключений выполняет алгоритм на рис. 14.2(а), используя сохраненную операцию для определения целевого адреса. При таком подходе для отдельных блоков выполняется машинный код, что может оказаться быстрее, чем программный цикл выборки–декодирования–выполнения.

Построение трассы

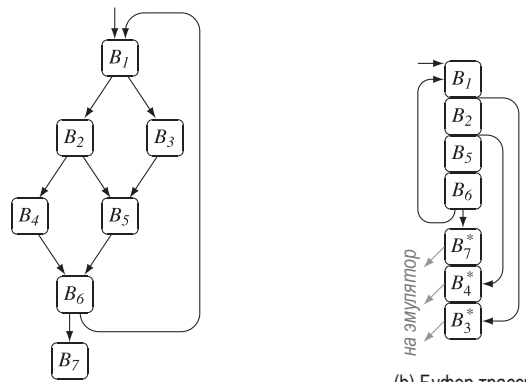
Когда счетчик входного блока трассы превышает пороговое значение, система вызывает оптимизатор, передавая ему адрес входного блока. Оптимизатор должен построить копию трассы, оптимизировать ее и поместить оптимизированный фрагмент в кеш трасс.

> Оптимизатор оставляет оригинальный код без изменения и на прежнем месте, чтобы другие пути, например входящие в трассу сбоку, могли и дальше использовать эти блоки в режиме эмуляции.

Хотя система знает, что входной блок был выполнен больше чем *threshold* раз, она не знает, на каком пути или путях лежали эти выполнения. Dynamo предполагает, что текущее выполнение пойдет по горячему пути. Поэтому оптимизатор начинает с входного блока и выполняет код в режиме эмуляции, пока не дойдет до конца трассы – выбранной обратной ветви или передачи управления на входной блок откомпилированной трассы. Эти условия, как и раньше, проверяются путем сравнения адресов.

В процессе выполнения кода оптимизатор копирует каждый блок в буфер. В каждой выбранной ветви или безусловном переходе он проверяет условия завершения трассы. Неявная ветвь обозначает побочный выход из трассы, поэтому оптимизатор запоминает целевой адрес, чтобы можно было связать побочный выход с подходящей трассой или заглушкой. Достигнув конца трассы, оптимизатор выполнил эту трассу и построил линеаризованную версию кода, которую может оптимизировать JIT-компилятор.

Рассмотрим пример на рис. 14.3(a). Когда эмулятор видит, что счетчик B_1 превысил порог, он вызывает оптимизатор. Тот выполняет B_1 и копирует все его операции в буфер. Следующий условный переход передает управление B_2; эмулятор выполняет B_2 и добавляет его в буфер. Затем управление попадает в B_5 и в B_6. Переход в конце B_6 возвращается в B_1, и трасса на этом завершается. В этот момент буфер содержит B_1, B_2, B_5 и B, как показано на рисунке (b).



(a) Пример ГПУ

(b) Буфер трассы для (B_1, B_2, B_3, B_6)

Рис. 14.3 Построение трассы

На рисунке предполагается, что все побочные выходы ведут на холодный код. Таким образом, JIT-компилятор стоит заглушку для обработки каждого побочного выхода и выхода в конце трассы. Заглушка, помеченная B_i^*, устанавливает PC эмулятора равным началу блока B_i и переходит на эмулятор. Заглушка также предоставляет место, в которое оптимизатор может вставить любой код, необходимый для связывания откомпилированного кода с эмулированным. На рисунке (b) показаны заглушки для B_3, B_4 и B_7.

Оптимизатор строит трассу, исходя из динамического поведения выполняемого кода, который может порождать сложные эффекты. Например, трасса может пройти через вызов процедуры и – в случае когда вызываемая сторона простая – через возврат. Поскольку вызов и возврат реализованы безусловными, а не условными переходами, они не вызовут срабатывания критерия выходного блока.

Оптимизация трассы

Построив копию трассы, оптимизатор совершает один или несколько проходов по ней, чтобы проанализировать и оптимизировать код. Если начальный проход является обратным, то оптимизатор может собрать информацию Live и другие полезные сведения. С точки зрения оптимизации, трасса напоминает один путь по расширенному простому блоку (см. раздел 8.5). В нашем примере трассы операция в B_6 может опираться на факты, выведенные из любого его предшественника, поскольку все они должны быть выполнены перед тем, как управление достигнет этой копии B_6.

Сам факт построения трассы должен привести к некоторым улучшениям кода. Компилятор может устранить все лежащие на трассе безусловные переходы. Для каждого преждевременного выхода оптимизатор должен сделать путь, принадлежащий трассе, неявным. Линеаризация кода должна дать небольшое повышение производительности за счет устранения некоторых задержек условных и безусловных переходов и улучшения локальности кеша команд.

Автор компилятора должен выбрать, какие оптимизации будут применены к трассе JIT-компилятором. Нумерация значений – очевидный выбор; она устраняет избыточности, сворачивает константы и упрощает алгебраические тождества.

Если трасса заканчивается условным переходом на свой входной блок, то оптимизатор может развернуть этот путь через цикл. В цикле с потоком управления результатом может стать цикл, развернутый на одних путях и не развернутый на других, – такая ситуация в традиционном AOT-оптимизаторе не возникает.

Преждевременные выходы из трассы привносят осложнения. Те же проблемы с компенсирующим кодом, которые возникают при региональном планировании, касаются и перемещения кода через точки преждевременного выхода (например, на концах B_1 и B_2). Если оптимизация перемещает операцию через такую точку, то, возможно, понадобится вставить код в заглушку для нее.

Частично мертвая операция
Операция называется *частично мертвой* в точке кода p, если она жива на некоторых путях, начинающихся в p, и мертва на остальных.

Оптимизатор может распознать некоторые вхождения мертвого или частично мертвого кода. Рассмотрим операцию, которая определяет r_i. Если r_i переопределяется раньше следующего использования на трассе, то оригинальное определение можно перенести в заглушку для любого преждевременного выхода между оригинальным определением и переопределением. Если он не переопределяется, но и не используется на трассе, то определение может быть перенесено в заглушки для преждевременных выходов и в конечный блок трассы.

После оптимизации компилятор должен спланировать операции и произвести распределение регистров. И снова можно применить локальные версии этих преобразований, вставив компенсационный код в точках преждевременного выхода.

Размер кеша трасс

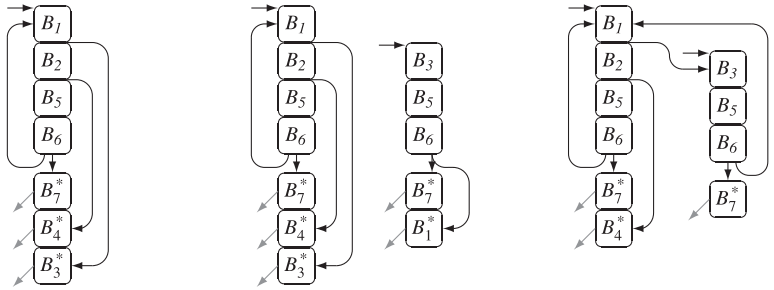
От размера кеша трасс может зависеть производительность. Размер влияет на несколько аспектов поведения кеша трасс – от локальности памяти до стоимости поиска и управления вытеснением. Если кеш слишком мал, то JIT может отбрасывать все еще горячие фрагменты, что приведет к снижению производительности и последующим перекомпиляциям. Если кеш слишком велик, то в нем может оставаться код, ставший холодным, что повредит локальности и увеличит затраты на поиск. Без сомнения, авторы компиляторов должны настраивать размер кеша трасс под характеристики конкретной системы.

14.3.2. Связывание трасс

Одно из важных условий эффективного выполнения – умение распознать, когда другие пути в ГПУ становятся горячими, и оптимизировать их так, чтобы они хорошо работали совместно с уже находящимися в кеше фрагментами.

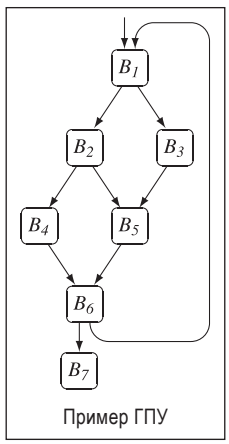
В нашем примере блок B_1 стал горячим, и оптимизатор построил фрагмент для пути $\langle B_1, B_2, B_5, B_6\rangle$, как показано на рис. 14.4(a). Тогда преждевременные выходы в B_3 и B_4 делают эти блоки входными блоками трассы. Если B_3 станет горячим, то оптимизатор построит для него трассу. Единственная трасса, которую он может построить, – это $\langle B_3, B_5, B_6\rangle$, как показано на рис. 14.4(b).

Если оптимизатор будет хранить небольшое количество информации о входах и выходах из трасс, то сможет связать обе трассы на рисунке (b), создав код, показанный на рисунке (c). Он может переписать условный переход на B_1^* в виде прямого безусловного перехода на B_1. Аналогично он может переписать условный переход на B_3^* в виде прямого безусловного перехода на B_3. Тогда связанные трассы создают быстрые пути выполнения для $\langle B_1, B_2, B_5, B_6\rangle$ и $\langle B_1, B_3, B_5, B_6\rangle$, как показано на рисунке (c). Выходы в B_4 и B_7 по-прежнему работают через заглушки, связывающие их с интерпретатором.



(a) Оригинальная трасса (b) Трасса для $\langle B_1, B_5, B_6 \rangle$ (c) После связывания трасс

Рис. 14.4 Добавление второй трассы



Пример ГПУ

Если бы в процессе оптимизации $\langle B_1, B_2, B_5, B_6 \rangle$ JIT-компилятор переместил операции в B_3^*, то процесс связывания должен был бы либо (1) оставить B_3^* на пути из B_1 в B_3, либо (2) доказать, что операции в B_3^* мертвы. При наличии небольшого количества контекста, например набора регистров, определенных до использования во фрагменте, это помогло бы распознать мертвый компенсационный код в заглушке.

Подобное перекрестное связывание также устраняет одно слабое место в эвристике построения трасс. Конструктор трасс предполагал, что $(k + 1)$-е выполнение B_1 имело место на горячем пути. Поскольку система оснащает измерительными средствами только заголовочные блоки трасс, счетчик выполнений B_1 мог бы увеличиваться на нескольких путях между B_1 и B_6. А что будет, если $(k + 1)$-е выполнение приходится на наименее горячий из этих путей?

Благодаря связыванию трасс $(k + 1)$-е выполнение будет строить оптимизированный фрагмент. Если это выполнение не находится на горячем пути, то один или несколько преждевременных выходов во фрагменте станут горячими; оптимизатор откомпилирует их и свяжет в трассу, подхватив один или несколько горячих путей. Оптимизатор будет постепенно восстанавливаться в процессе построения связанного множества трасс.

Промежуточные входы в трассу

В нашем примере, когда B_1 становился горячим, система строила оптимизированную трассу для $\langle B_1, B_2, B_5, B_6 \rangle$. Когда B_3 становился горячим, она оптимизировала $\langle B_3, B_5, B_6 \rangle$.

Как было объяснено, алгоритм строит одну трассу для $\langle B_1, B_2, B_5, B_6 \rangle$ и игнорирует промежуточный вход в нее с ребра (B_3, B_5). Затем система выполняет путь $\langle B_3, B_5, B_6 \rangle$ в режиме эмуляции до тех пор, пока счетчик B_3 не запустит компиляцию этого пути. Эта последовательность действий порождает две

копии кода для B_5 и B_6, а также требует дополнительного времени JIT-компилятора для их оптимизации.

Ребро (B_3, B_5) можно было бы обработать и по-другому: построить промежуточный вход в трассу $\langle B_1, B_2, B_5, B_6 \rangle$. Как было сказано выше, алгоритм построения трасс игнорирует эти промежуточные точки входа, что упрощает внутренний учет. Если бы эмулятор знал, что B_5 является промежуточной точкой входа, то мог бы расщепить трассу при входе в B_5. Он построил бы оптимизированную трассу для $\langle B_1, B_2 \rangle$ и еще одну для $\langle B_5, B_6 \rangle$. И связал бы ожидаемый выход из $\langle B_1, B_2 \rangle$ с началом $\langle B_5, B_6 \rangle$.

B_4 имеет одного предшественника, а B_5 – двух.

Чтобы реализовать расщепление трассы, оптимизатору необходим эффективный механизм распознавания промежуточной точки входа в трассу – дабы различить, например, B_4 и B_5. Выше было отмечено, что оптимизатор горячих трасс не строит явное представление ГПУ. Один из вариантов – поручить AOT-компилятору аннотировать BM-код этой информацией.

Расщепление $\langle B_1, B_2, B_5, B_6 \rangle$ после B_2 может порождать менее эффективный код, чем компиляция нерасщепленной трассы. Расщепление трассы позволяет избежать повторной компиляции $\langle B_5, B_6 \rangle$ и сохранения дополнительного кода в кеше. Оно требует дополнительной записи в таблице входных блоков. Этот компромисс, по-видимому, неизбежен. Наилучшее решение вполне может зависеть от длины общего суффикса двух путей, которые может быть трудно распознать при компиляции первой трассы.

ИТОГИ РАЗДЕЛА

Оптимизатор горячих трасс идентифицирует часто выполняемые трассы в исполняемом коде, оптимизирует их и перенаправляет будущие выполнения на оптимизированный код. Он предполагает, что частое выполнение в прошлом предсказывает частое выполнение в будущем и направляет усилия JIT-компилятора на такой «горячий» код. Поскольку трассы ациклические, открывается возможность использовать локальные и суперлокальные оптимизации. Эти методы работают быстро и способны обнаруживать многие из имеющихся возможностей.

Использование связанных и межпроцедурных трасс позволяет оптимизатору горячих трасс реализовывать такие виды частичной оптимизации, на которые AOT-компилятор не способен. Цель – сфокусировать усилия JIT-компилятора на тех местах, которые могут дать максимальный эффект, и не напрягать его оптимизацией участков, где ожидаемый эффект мал.

ВОПРОСЫ

1. После того как входной блок трассы стал горячим, оптимизатор выбирает оставшуюся часть трассы на основе следующего выполнения входного блока. Сравните эту стратегию с алгоритмом обнаружения трасс, который используется при планировании трасс. Насколько могут различаться результаты этих двух подходов?
2. Предположим, что оптимизатор трасс заполнил кеш трасс и должен вытеснить какую-то трассу. Какие шаги придется предпринять, чтобы вернуть трассу в состояние, когда она выполняется в режиме эмуляции BM-кода?

14.4. Оптимизация горячих методов

Оптимизация на уровне методов сталкивается с другими проблемами и компромиссами по сравнению с оптимизацией на уровне трасс. Чтобы исследовать эти вопросы, мы сначала рассмотрим оптимизатор горячих методов, включенный в среду JAVA. На выбранный нами дизайн оказал влияние компилятор HotSpot Server (далее просто HotSpot). Это смешанная среда, в которой холодные методы выполняются как байт-код JAVA, а горячие – как машинный код. Мы завершим данный раздел обсуждением различий в оптимизаторе машинного кода горячих методов.

14.4.1. Горячие методы в смешанной среде

На рис. 14.5 показано абстрактное представление виртуальной машины JAVA, или JVM. Классы и ассоциированные с ними методы загружаются в среду загрузчиком классом. Оказавшись в ВМ, методы выполняются эмулятором – на рисунке он назван «движок байт-кода». JVM работает в смешанном режиме, пользуясь реализациями многих стандартных методов в системных библиотеках, представленными в машинном коде.

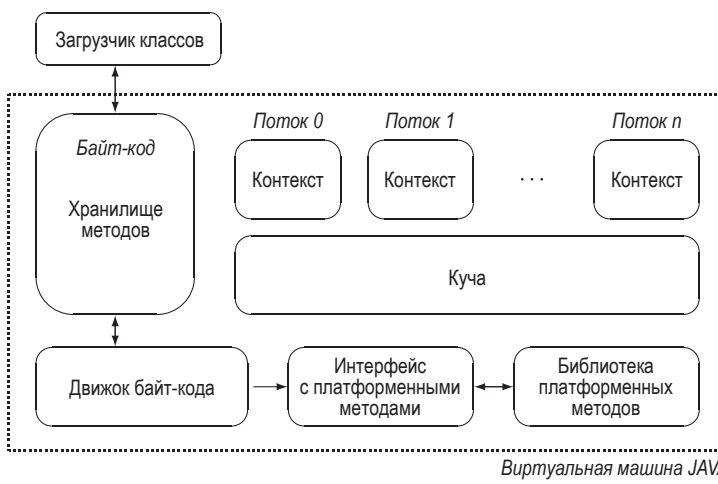


Рис. 14.5 Среда выполнения JAVA

Чтобы добавить JIT-компилятор уровня методов, автор компилятора должен включить в JVM несколько средств: сам JIT-компилятор, программно-управляемый кеш тел методов в машинном коде и необходимые интерфейсы. Эти модификации показаны на рис. 14.6.

С точки зрения выполнения, наличие JIT-компилятора несет с собой несколько изменений. Холодный код по-прежнему выполняется в режиме эмуляции ВМ-кода; методы из платформенных библиотек по-прежнему выполняются как машинный код. Когда система решает, что метод горячий, она JIT-компилирует ВМ-код в машинный и сохраняет новый код в кеше

машинного кода. При последующих вызовах метода исполняется машинный код, если только система не решит вернуться к эмуляции ВМ-кода для этого метода (см. обсуждение деоптимизации на стр. 771).

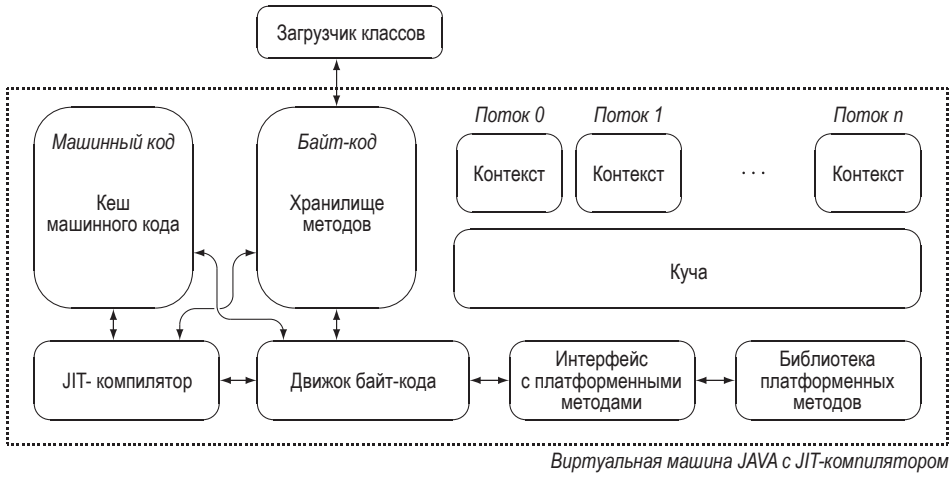


Рис. 14.6 Добавление JIT-компилятора в среду выполнения JAVA

Автор компилятора должен принять несколько ключевых решений. Системе необходим механизм, позволяющий решить, какие методы следует компилировать. Нужна стратегия эффективного сбора информации профилирования. Нужно решить, будет ли машинный код работать с различными структурами данных, например записями активации, представленными в ВМ-коде или в машинном коде. Автор компилятора должен спроектировать и реализовать эффективный, хотя и ограниченный JIT-компилятор. Наконец, он должен спроектировать и реализовать механизм отката к эмуляции ВМ-кода метода, когда откомпилированный метод оказывается непригодным. Мы рассмотрим эти вопросы в порядке очередности.

> Если использовать записи активации для машинного кода, то придется осуществлять трансляцию машинного кода в ВМ-код и обратно.
>
> В сообществе JAVA записи активации часто называют кадрами стека.

Триггер компиляции

Концептуально оптимизатор горячих методов должен компилировать метод, если тот потребляет значительную долю времени выполнения. Найти горячий метод, отвечающий этому критерию, труднее, чем горячую трассу, потому что понятие «значительной доли» неточно и не определимо до момента завершения программы.

Поэтому оптимизаторы горячих методов прибегают к использованию счетчиков и пороговых значений для оценки активности метода. Этот подход опирается на неявное пред-

> Итерации бывают как у цикла, так и у рекурсии. Механизм должен улавливать оба случая.

положение о том, что метод, уже потребивший значительную долю времени выполнения, будет потреблять столько же и в будущем. В нашем дизайне, как и в HotSpot, измеряется: (1) сколько раз метод вызывался; (2) сколько итераций цикла он выполняет. Ни та, ни другая метрика не позволяет утверждать, что метод использует значительную долю времени выполнения. Однако для любого метода, который таки потребляет значительную долю времени, хотя бы одна из этих метрик почти наверняка будет велика.

Компилятор HotSpot Server

Примерно в 2000 году компания Sun Microsystems поставила два JIT-компилятора для своей среды JAVA: один для выполнения на стороне клиента, другой – на стороне сервера. В первоначальном компиляторе HotSpot Server использовалась более дорогая и всеобъемлющая техника, чем в клиентском JIT-компиляторе. Компилятор HotSpot Server был примечателен тем, что применял сильные методы глобальной оптимизации, соблюдая пространственные и временные ограничения JIT-компиляции. Авторы выбрали ПП, в котором были явно представлены поток управления и поток данных [92]. Это ПП было приспособлено для устранения избыточности распространения констант и перемещения кода. Разреженность ПП способствовала быстрому выполнению этих оптимизаций.

В JIT-компиляторе применялся новый алгоритм глобального планирования и распределитель на основе полного раскрашивания графа (см. раздел 13.4). Чтобы распределитель оставался практичным, авторы разработали метод обрезания графа интерференции, позволявший значительно уменьшить его размер. В результате появился ультрасовременный JIT-компилятор, в котором использовались алгоритмы, ранее считавшиеся слишком дорогими для применения в контексте JIT.

Система умеет «суммировать» счетчики, используя одно место для всех счетчиков в методе.

Таким образом, система должна подсчитывать как вызовы метода, так и итерации цикла внутри метода. Размещенные в стратегически важных местах счетчики могут уловить оба этих условия. Чтобы подсчитывать вызовы, система может вставить счетчик в пролог каждого метода. Для подсчета итераций счетчик вставляется перед каждым условным переходом, замыкающим цикл. Для запуска компиляции можно использовать либо отдельные пороги для циклов и вызовов, либо один порог для суммы этих счетчиков.

HotSpot подсчитывал вызовы и итерации и запускал компиляцию, когда комбинированный счетчик превышал порог – 10 000 событий. Этот порог намного больше, чем в Dynamo (50). Он отражает тот факт, что в HotSpot компиляция более агрессивная и дорогостоящая.

Данные профилирования на этапе выполнения

Для сбора данных профилирования авторы компиляторов могут оснащать либо код приложения для ВМ, либо реализацию движка ВМ-кода.

Оснащенный ВМ-код. Система может вставлять в метод ВМ-код, который увеличивает и проверяет счетчики профилирования. При таком дизайне на-

кладные расходы на профилирование включаются в ВМ-код. Оснащать код измерительными средствами может АОТ-компилятор или загрузчик классов. Счетчик вызовов можно поместить в пролог кода, а счетчики для конкретных мест вызова включить в состав предварительной последовательности.

Для профилирования итераций циклов преобразование может вставить счетчик в любой блок, являющийся целью обратного условного или безусловного перехода. АОТ-компилятор может придерживаться стратегии уменьшения стоимости оснащения; например, если компилятор знает число итераций, то может увеличить счетчик один раз для всего цикла.

Оснащенный движок. Автор компилятора может встроить поддержку профилирования прямо в реализацию движка кода ВМ. При такой схеме код эмулятора для условных и безусловных переходов, а также вызовов (например, операции JVM `invokestatic` или `invokevirtual`) может увеличивать и проверять нужные счетчики, которые размещены в заранее запланированных местах. Поскольку код профилирования исполняется как машинный, этот подход должен работать быстрее, чем оснащение ВМ-кода.

Эмулятор мог бы применять стратегию сравнения адресов, используемую оптимизатором трасс, для идентификации заголовочных блоков циклов. Если целевой адрес численно меньше PC, то расположенный по нему блок может быть заголовком цикла. Или же эмулятор мог бы рассчитывать на то, что АОТ-компилятор предоставит аннотацию, помечающую заголовочный блок цикла.

Сам же АОТ-компилятор мог бы находить заголовки циклов с помощью доминаторов (см. раздел 9.2.1).

Компиляция горячего метода

Когда данные профилирования запускают JIT-компиляцию некоторого метода *x*, компилятор может просто извлечь ВМ-код для *x* и откомпилировать его. JIT-компилятор действует как полнофункциональный компилятор. Он разбирает ВМ-код, строит ПП, применяет к нему один или несколько проходов оптимизации и генерирует машинный код, для чего выполняет выбор команд, планирование и распределение регистров. JIT-компилятор записывает этот машинный код в кеш (см. рис. 14.6). Наконец, он обновляет таблицы или указатели, которые система использует для вызова методов, так чтобы последующие вызовы вели на кешированный машинный код.

Методы сопоставления с деревом-образцом хорошо согласуются с ограничениями JIT-компилятора (см. раздел 11.4).

В смешанной среде преимущества JIT-компиляции должны проявляться ярче, чем в платформенной, потому что холодный код выполняется медленнее. Таким образом, оптимизатор горячих методов в смешанной среде может позволить себе потратить больше времени на оптимизацию и генерирование кода каждого метода. В таких оптимизаторах применялись многие классические скалярные оптимизации, например нумерация значений, распространение констант, устранение мертвого кода и перемещение кода (см. главы 8 и 10).

Глобальная нумерация значений

В литературе по JIT-компиляции на уровне методов часто упоминается *глобальная нумерация значений* как одна из основных оптимизаций. Прилежный студент не найдет единодушного мнения по поводу смысла этого термина, который применялся для обозначения совершенно различных алгоритмов.

Один из подходов обобщает идеи локальной нумерации значений на глобальную область, следуя по пути, проложенному суперлокальной и основанной на доминировании нумерацией значений (НЗД). Эти алгоритмы дороже, чем НЗД, и соотношение затрат и преимуществ между НЗД и глобальным алгоритмом не очевидно.

В другом подходе используется алгоритм разбиения Хопкрофта для нахождения операций, вычисляющих одно и то же значение, а затем код переписывается с целью отразить эти факты. В компиляторе HotSpot Server применяется именно эта идея, которая хорошо ложится на его ПП в виде графа зависимостей.

Наконец, автор JIT-компилятора мог бы отталкиваться от идей отложенного перемещения кода (ОПК). Этот подход комбинирует перемещение кода и устранение избыточности. Поскольку реализации ОПК решают несколько задач анализа потока данных, автор JIT-компилятора должен обращать пристальное внимание на стоимость анализа.

Оптимизации

JIT-компиляторы горячих методов применяют локальные, региональные и глобальные оптимизации. Поскольку JIT-компилятор работает во время выполнения, автор может предоставить ему доступ к состоянию исполняемой программы, чтобы получить интересующие значения и использовать их в процессе оптимизации.

Нумерация значений. JIT-компиляторы уровня методов применяют тот или иной вариант нумерации значений. Это может быть региональный алгоритм, например НЗД, или один из многих глобальных алгоритмов.

Нумерация значений интересна автору JIT-компилятора тем, что такие алгоритмы достигают нескольких эффектов сравнительно небольшой ценой. Обычно они выполняют некоторое подмножество устранения избыточности, перемещения кода, распространения констант и упрощения алгебраических выражений.

Встроенные кеши методов могут предоставить сведения о типе получателя в конкретном месте вызова. Эту идею можно обобщить для сбора информации о типах параметров.

Специализация для данных времени выполнения. JIT-компилятор может получить доступ к данным о поведении кода в текущем выполнении, точнее о значениях переменных, информации о типах и о данных профилирования. Информация о типах во время выполнения позволяет JIT-компилятору строить предположения; зная, что значение всегда имело тип *t* в прошлом, JIT-компилятор предполагает, что оно будет иметь тип *t* и в будущем.

Такие предположения могут принимать форму реализации быстрого и медленного пути. На рис. 14.7 показано, как такая схема могла бы работать

на концептуальном уровне. Предполагается, что *x* и *y* – 32-разрядные целые, и это предположение проверяется; если оно не выполнено, то вызывается общая функция сложения. Если медленный путь выполняется слишком много раз, то система могла бы перекомпилировать код с новыми предположительными типами (см. обсуждение «деоптимизации» на стр. 771).

```
// x ← y + z
if actual_type(y) = 32_bit_integer and
  actual_type(z) = 32_bit_integer then
   x ← 32_bit_integer_add(y, z)
else
   x ← generic_add(y, actual_type(y), z, actual_type(z));
```

Рис. 14.7 Код с быстрым путем для ожидаемого случая

Встраивание. JIT-компилятор может встраивать вызовы, что позволяет ему устранять накладные расходы на поиск метода и на вызов. Встраивание позволяет адаптировать тело вызываемой стороны к окружению в месте вызова. JIT-компилятор может использовать данные из встроенных кешей методов, чтобы специализировать код на основе конкретного типа данных в месте вызова. Он также может встраивать место вызова в вызываемую сторону; он должен иметь доступ к данным профилирования и данным кеша методов для всех таких вызовов.

Когда JIT-компилятор рассматривает встраивание, у него есть доступ ко всему коду приложения.

В случае AOT-компилятора такая ситуация маловероятна.

JIT-компилятор может взглянуть на вызывающие стороны, чтобы оценить возможный выигрыш от встраивания метода в одну или несколько вызывающих его сторон. И снова данные профилирования о вызывающей стороне и информация о типе из встроенного кеша могут помочь JIT-компилятору в принятии хороших решений о том, когда встраивание имеет смысл.

Генерирование кода. Выбор команд, планирование и распределение регистров могут еще улучшить качество JIT-компилированного кода. Для эффективного вызова команд следует с толком использовать такие возможности ISA, как режимы адресации. При планировании команд нужно разумно распорядиться параллелизмом на уровне команд, избегая простоев и блокировок оборудования. При распределении регистров компилятор стремится минимизировать дорогостоящие операции выгрузки.

Вызовом для автора JIT-компилятора является эффективная реализация этих проходов. Метод сопоставления с деревом-образцом, применяемый при выборе команд, сочетает выбор локально оптимального кода с исключительной эффективностью компиляции во время выполнения. И планировщик, и распределитель могут воспользоваться разреженностью, чтобы сделать глобальные алгоритмы достаточно эффективными для реализации в JIT-компиляторе. Компилятор HotSpot продемонстрировал, что при эффек-

тивной реализации эти глобальные алгоритмы могут оказаться не только приемлемыми, но и выигрышными.

Как в любой JAVA-системе, JIT-компилятор должен приложить усилия, чтобы устранить проверки на нулевой указатель или переместить их туда, где они выполняются не так часто. Анализ локальности может обнаружить объекты, время жизни которых гарантированно включено во время жизни какого-то метода. Память для таких объектов можно выделить в ЗА метода, а не в куче, что уменьшит стоимость выделения и сборки мусора. Финальные и статические методы можно встраивать.

Деоптимизация

Деоптимизация
JIT-компилятор генерирует в меньшей степени оптимизированный код из-за изменения информации времени выполнения.

Если JIT-компилятор пользуется информацией времени выполнения для оптимизации кода, то рискует тем, что изменение этой информации может сделать откомпилированный код плохо оптимизированным или недействительным. Автор компилятора должен иметь план разумного поведения на случай ложного допущения о типе. В таком случае система может прибегнуть к *деоптимизации* кода.

Чтобы «заметить», система должна оснастить медленный путь измерительными средствами.

Снова рассмотрим рис. 14.7. Если система в какой-то момент заметит, что большинство выполнений этого оператора шли по пути `generic_add`, то сможет перекомпилировать код, сделав другое предположение о типе, предположив, что типов может быть несколько, или не делая вообще никаких предположений. Если изменение вызвано постоянным сдвигом в поведении программы, то повторная оптимизация может помочь.

Но если просто пропала согласованность между типами *x*, *y* и *z*, то повторная оптимизация может оказаться неверным решением. Если откомпилированный код не выполняется достаточно часто, чтобы окупить стоимость JIT-компиляции, то повторная компиляция только замедлит выполнение.

Альтернатива заключается в том, чтобы деоптимизировать код. В зависимости от ситуации и глубины знаний JIT-компилятора о временно́й локальности типа можно воспользоваться одной из следующих стратегий:

- если JIT-компилятор знает, что фактическим является тип малого числа, то может сгенерировать код быстрого пути для каждого из таких типов;
- если JIT-компилятор не знает ничего, кроме того что предположение о типе часто оказывается неверным, то может сгенерировать неоптимизированный машинный код, который просто вызывает *generic_add*, или встроить *generic_add* в место операции;
- если JIT-компилятор слишком часто вызывался для этого кода из-за меняющегося паттерна, то может пометить код как непригодный для JIT-компиляции и далее выполнять его в режиме эмуляции.

Стратегия деоптимизации позволяет JIT-компилятору строить предположения, но ограничивает риски, связанные с его неправотой.

14.4.2. Горячие методы в платформенной среде

Характер нескольких проблем меняется, когда автор JIT-компилятора пытается применить оптимизацию горячих методов в платформенной среде. Этот раздел основан на идеях, примененных в реализации SMALLTALK-80 Дойча–Шиффмана.

Начальная компиляция

Платформенная среда должна гарантировать, что каждый метод компилируется в платформенный код до его выполнения. Работоспособны многие схемы, включая прямолинейную AOT-компиляцию, компиляцию на этапе загрузки и JIT-компиляцию при первом вызове, которую мы будем называть *компиляцией по вызову*. Первые два варианта реализовать проще, чем последний. Но ни один не дает возможности использовать информацию времени выполнения в процессе этой начальной компиляции.

Использование косвенного указателя на тело кода (указателя на указатель) может упростить реализацию.

Система компиляции по вызову сначала генерирует код для функции `main` программы. В каждом месте вызова она вставляет заглушку, которая (1) находит ВМ-код метода; (2) вызывает JIT-компилятор, чтобы сгенерировать машинный код метода; (3) перекомпонует место вызова, так чтобы оно указывало на только что откомпилированный машинный код. Когда в процессе выполнения впервые вызывается метод *m*, возникает задержка, во время которой *m* JIT-компилируется, после чего выполняется машинная версия *m*. Если имеется какая-то информация времени выполнения, то первый вызов *m* может ей воспользоваться.

Если JIT-компилятор поддерживает несколько уровней оптимизации, то его автор должен выбрать, какой уровень использовать для начальных компиляций. Более низкий уровень должен уменьшить стоимость начальной JIT-компиляции ценой более медленного выполнения. Более высокий уровень может увеличить стоимость компиляции, зато выполняться метод будет быстрее.

Компромиссное решение может состоять в том, чтобы использовать низкий уровень оптимизации при начальной компиляции и перекомпилировать метод с повышением уровня, когда он станет горячим. Разумеется, этот подход нуждается в данных о частоте выполнения и локальности типов.

Сбор данных профилирования

В платформенной среде система может собирать данные профилирования двумя способами: оснастить код для сбора данных в определенных точках

или перейти на управление от прерываний, чтобы понять, где исполняемая программа тратит больше времени.

Оснащенный код. JIT-компилятор может оснастить код, как было описано ранее при рассмотрении оптимизатора горячих методов в смешанном режиме. Можно вставить в прологи методов код, который будет подсчитывать общее число обращений к методу. Можно получить счетчики для каждого места вызова, вставив код в предварительные последовательности. Можно вставить код подсчета итераций циклов перед условными переходами, замыкающими цикл.

С оснащенным кодом вызов JIT-компилятора производится так же, как в смешанной среде. Компилятор вызывается, когда счетчики выполнений, обычно счетчики вызовов и итераций циклов, превысили заранее заданный порог. Для счетчика итераций также следует вставить код проверки порога и запуска компиляции.

Чтобы сделать возможными предположения о типах и специализацию кода на основе типов, JIT-компилятор может организовать запоминание типа или класса определенных значений – как правило, параметров, переданных методу, или значений, участвующих в вызове метода. JIT-компилятор должен иметь доступ к этой информации.

Профилирование, управляемое прерываниями. Счетчик вызова методов говорит системе о том, как часто вызывается метод. Счетчик итераций говорит о том, как часто выполняется тело цикла. Ни та, ни другая метрика не проливает свет на то, какую долю времени выполнения в действительности занимает оснащенный код.

Система должна генерировать таблицы, отображающие адреса на конкретные местоположения в исходном коде.

Платформенная среда тратит практически все время на выполнение кода приложения. Поэтому она может применить другую стратегию обнаружения горячего кода: профилирование, управляемое прерываниями. При таком подходе система периодически останавливает выполнение по прерыванию от таймера. Она сопоставляет адрес в указателе команды в момент прерывания с конкретным методом и увеличивает счетчик этого метода. Сравнение счетчика метода с общим количеством прерываний дает приближение к доле времени, потраченного на выполнение этого метода.

В некоторых системах использовалась комбинация оснащенного кода и данных профилирования, управляемого прерываниями.

Поскольку профилирование, управляемое прерываниями, измеряет нечто, имеющее тонкие отличия от того, что измеряется оснащенным кодом, автор JIT-компилятора должен ожидать, что схема с управлением от прерываний будет оптимизировать не те же методы, что схема с оснащенным кодом.

JIT-компилятору по-прежнему нужны данные о типах во время выполнения, чтобы направлять оптимизацию в нужную сторону. Хотя часть таких данных присутствует во встроенных кешах методов в местах их вызова, де-

тальную информацию система может собрать, только если компилятор добавляет в исполняемый файл код для профилирования типов.

Деоптимизация машинного кода

Начиная выполнение, откомпилированный метод должен выяснить, верны ли еще предусловия, при которых он был откомпилирован (например, предположение о типе или классе, параметры с константными значениями и т. д.). Код пролога метода может проверить любые предусловия, которые JIT-компилятор считал истинными при последней компиляции. В смешанной среде система могла бы выполнять ВМ-код, если предусловие нарушено, а в платформенной она обязана вызвать JIT-компилятор для перекомпиляции кода таким образом, который позволил бы продолжить выполнение.

В процессе перекомпиляции система должна попытаться обеспечить эффективное выполнение, избегая ситуаций, когда частые перекомпиляции сводят на нет все преимущества JIT. Может быть полезна любая из нескольких стратегий деоптимизации:

- можно было бы просто перекомпилировать код с учетом наилучшей на данный момент времени информации о типах времени выполнения. Если изменение предусловий было вызвано постоянным сдвигом в поведении программы, то текущие предусловия могут некоторое время оставаться истинными;
- если JIT-компилятор поддерживает несколько уровней оптимизации – особенно это относится к предположениям о типах, – то система могла бы поручить ему использовать уровень пониже, при котором порождается более общий, не столь адаптированный к контексту код. Этот подход позволяет избежать ситуации, когда код некоторого метода осциллирует между двумя или более оптимизированными состояниями;
- агрессивный JIT-компилятор мог бы сгенерировать новую версию кода с текущими предусловиями и вставить заглушку, которая выбирает нужный вариант кода в зависимости от предусловий. При этом увеличение размера кода обменивается на возможность улучшить производительность.

> В этой стратегии предполагается наличие счетчика, ограничивающего количество «постоянных сдвигов», с которыми JIT-компилятор может смириться для данного метода.

Лучшая стратегия зависит от того, насколько агрессивно JIT-компилятор использует информацию времени выполнения, чтобы обосновать целесообразность оптимизаций, и от экономики JIT-компиляции. Если компилятор потребляет малую долю времени выполнения, то его автор и пользователь могут смириться с перекомпиляциями. Напротив, если для компенсации стоимости JIT-компиляции нужно несколько выполнений метода, то перекомпиляция может оказаться куда менее привлекательной, чем понижение уровня оптимизации и уменьшение количества предположений.

Экономика JIT-компиляции

Фундаментальным компромиссом для автора JIT-компилятора является разность между числом тактов, потраченных и сэкономленных компилятором. В платформенной среде улучшение, достигнутое благодаря JIT-компиляции, может быть ниже просто потому, что неоптимизированный код работает быстрее, чем аналогичный код в смешанной среде.

Это наблюдение должно лечь в основу решений о том, какие оптимизации реализовывать и какое количество перекомпиляций можно считать терпимым. Автор JIT-компилятора должен найти баланс между затратами, выигрышем и политиками, чтобы создать систему, которая действительно повышает производительность на этапе выполнения.

ИТОГИ РАЗДЕЛА

Оптимизатор горячих методов находит процедуры, которые либо выполняются часто, либо занимают значительную долю времени выполнения. Он оптимизирует каждую процедуру с учетом установленных во время выполнения фактов. Поскольку оптимизатор методов может столкнуться с потоком управления, он может получить выигрыш от региональных и глобальных оптимизаций, таких как глобальная нумерация значений или перемещение кода; эти преобразования обходятся дороже, но зато в случае успеха могут принести больший выигрыш, чем локальные и суперлокальные методы, доступные оптимизатору трасс.

Компромиссы, принятые в конкретном дизайне, зависят от среды выполнения, от того, какие средства исходного языка являются источниками неэффективности, и от того, какого рода информация собирается в среде выполнения. Для проектирования оптимизатора горячих методов требуется понимать язык, систему, алгоритмы и поведение оптимизируемых приложений.

ВОПРОСЫ

1. Какие виды данных эмулятор мог бы собирать для упрощения специализации в смешанной среде? Как он мог бы идентифицировать такие возможности и собрать необходимые данные?
2. Профилирование, управляемое прерываниями, может хорошо работать в среде, где основное время выполнения приходится на машинный код. Назовите некоторые трудности, с которыми мог бы столкнуться автор компилятора при использовании профилирования, управляемого прерываниями, в смешанной среде.

14.5. Дополнительные темы

В предыдущих разделах мы познакомились с основными проблемами, которые возникают при проектировании оптимизатора этапа выполнения. Однако для построения такой системы автор компилятора должен принимать мириады решений, большинство из которых прямо влияют на эффективность системы. В этом разделе затрагивается несколько важных тем, которые обсуждаются в литературе, относящейся к JIT-компиляции. Каждая из них имеет практические последствия для проектирования системы. Каждая может изменить ее общую эффективность.

14.5.1. Уровни оптимизации

На практике мы знаем немногих разработчиков, которые думают об этапе компиляции, выбирая уровень оптимизации.

AOT-компиляторы обычно предлагают конечному пользователю выбрать один из нескольких уровней оптимизации. Теоретически это позволяет использовать более сильную оптимизацию там, где это важно, сэкономив на времени компиляции в местах, где от дополнительной оптимизации мало что зависит.

Система на базе JIT-компиляции могла бы предлагать несколько уровней оптимизации по ряду причин:

- потому что время выполнения приложения включает время работы JIT-компилятора, автор которого может решить, что в стандартный режим следует включать только те оптимизации, которые всегда дают улучшение;
- система может обнаружить, что фрагмент машинного кода выполняется достаточно часто для того, чтобы расширенный анализ и оптимизация были оправданы, поскольку хотя при этом и требуется больше времени на JIT-компиляцию, зато и время выполнения уменьшается;
- если JIT-компилятор строит какие-то предположения на основе информации времени выполнения, например о типах и классах, то впоследствии, возможно, придется деоптимизировать код, поэтому, быть может, стоит уменьшить уровень оптимизации.

Если система обнаруживает цикл с большим числом итераций, то может применить специальные оптимизации циклов, такие как разворачивание, снижение стоимости операций или перемещение кода. Чтобы эти оптимизации возымели немедленный эффект, система могла бы выполнить замещение в стеке (см. раздел 14.5.2).

Если система обнаруживает, что в какой-то один метод попадает значительная доля точек профилирования, управляемого прерываниями, то может провести более глубокий анализ и более интенсивную оптимизацию. Например, она могла бы встроить вызовы, выполнить анализ для устранения неоднозначности типов и классов и оптимизировать код повторно.

При любом из этих сценариев JIT-компилятор с несколькими уровнями оптимизации нуждается в четко определенном наборе политик, определяющих, когда и где использовать каждый уровень. Одной из ключевых частей этой стратегии будет механизм, препятствующий компилятору слишком часто изменять уровень оптимизации, что лишь увеличивало бы затраты на компиляцию без достаточного числа выполнений, чтобы эти затраты амортизировать.

14.5.2. Замещение в стеке

JIT-компилятор методов может столкнуться с ситуацией, когда один из счетчиков профилирования пересекает порог во время выполнения долго работающего метода. Например, рассмотрим метод, в котором имеется трижды

вложенный цикл с числом итераций 100 на каждом уровне вложенности. В случае порога 10 000 счетчик во внутреннем цикле запустил бы компиляцию после выполнения всего лишь одного процента итераций.

По сути дела, система ведет себя так, будто в долго работающем цикле порог запуска компиляции выше.

Счетчик показывает, что метод горячий и его следует оптимизировать. Если система ожидает следующего вызова, чтобы «установить» оптимизированный код, то текущий код будет выполняться до конца текущего вызова. В трижды вложенном цикле 99 % итераций будет выполнено уже после того, как счетчик превысил порог оптимизации.

Чтобы не упустить возможность, система могла бы приостановить выполнение, оптимизировать и откомпилировать код, после чего возобновить выполнение с уже улучшенным кодом. Этот подход предполагает, что ускорение работы откомпилированного кода на большинстве итераций цикла окупит все затраты. Однако чтобы возобновить выполнение заново оптимизированного кода, система должна отобразить состояние приостановленного выполнения на состояние нового кода.

Замещение в стеке
Техника, позволяющая среде приостановить выполнение, JIT-компилировать исполняемую процедуру и возобновить выполнение уже с заново откомпилированным кодом.

Такую оптимизацию процедуры способом, позволяющим продолжить ее выполнение, часто называют *замещением в стеке*. JIT-компилятор строит код, который в максимально возможной степени способен продолжить работу в текущей среде выполнения. Если он не может сохранить значения, то должен организовать отображение значений из текущей среды в новую.

JIT-компилятор может воспользоваться своими детальными знаниями о старом коде, чтобы создать новую среду. Он может сгенерировать небольшую заглушку, которая преобразует текущую среду – значения в регистрах плюс текущую запись активации – в среду, необходимую новому коду.

- Возможно, заглушка должна будет переместить некоторые значения. Карты памяти нового и старого кодов могут не совпадать.
- Возможно, заглушке придется вычислить некоторые значения. Такие оптимизации, как перемещение кода или снижение стоимости операций, могут создавать новые значения, которых не было в оригинальном коде.
- Возможно, заглушка отбросит некоторые значения. Состояние оригинального кода может содержать значения, мертвые или не используемые в новом коде.

Заглушка срабатывает один раз, перед первым выполнением нового кода. После этого ее можно отбросить. Если JIT-компилятор работает в отдельном потоке, как часто бывает, то необходим какой-то протокол взаимодействия между JIT-компилятором и исполняемым кодом, позволяющий определить, когда следует переключаться на новый код.

У автора компилятора есть несколько способов уменьшить сложность замещения в стеке.

- Он может ограничить количество мест в коде, где система может произвести замещение. Начало итерации цикла – одно из таких естественных мест. Выполнение следующей итерации начинается после компиляции и отображения состояния.
- Он может упростить задачу отображения состояния, ограничив набор оптимизаций, доступных JIT-компилятору в случае компиляции для замещения в стеке. В частности, компилятор мог бы избегать алгоритмов, которые требуют значительной работы для отображения старой среды на новую.

Приемы создания компенсационного кода или внедрения новых значений могут усложнить отображение. Примерами могут служить перемещение кода, программный конвейер и встраивание.

Реализация замещения в стеке фундаментально связана с интерфейсами между эмулируемым и откомпилированным кодом и средами их выполнения. Детали зависят от конкретной системы. Эта стратегия потенциально может принести значительное улучшение производительности долго работающих методов.

14.5.3. Управление кешем методов

Почти все основанные на JIT-компиляции системы строят и обслуживают кеш кода – специальный программно-управляемый блок памяти, в котором хранится JIT-компилированный код. Автор JIT-компилятора должен спроектировать политики и создать механизмы управления кешем кода. Концептуально кеш кода объединяет задачи и политики аппаратного и программно-управляемого кешей.

Чтобы избежать путаницы, мы будем называть кеш JIT-компилятора *кешем кода*, а процессорные кеши – *аппаратными кешами*.

- Аппаратные кеши определяют местоположение объекта путем арифметического отображения виртуальных адресов в физические. В случае кучи программа ищет свободный блок, в котором будет храниться объект. Кеши кода ближе к модели размещения в куче.
- Аппаратные кеши имеют дело с блоками фиксированной длины. Кучи работают с блоками произвольного размера, но часто округляют запросы до некоторых стандартных величин. Кеши кода должны подстраиваться под блоки машинного кода разных размеров.
- В аппаратных кешах используются автоматические, основанные на политиках схемы вытеснения, как правило, знакомые с закономерностями предшествующего использования. Кучи обычно выполняют этап сбора, чтобы найти более неживые блоки (см. раздел 6.6.2). В кешах кода используются политики и механизмы, похожие на применяемые в аппаратных кешах.

В большинстве основанных на JIT-компиляции систем имеется отдельный кеш кода в каждом процессе или потоке. Некоторые авторы экспериментировали с глобальным кешем кода, чтобы JIT-компилированный код можно

было повторно использовать в нескольких процессах. Основное преимущество такого дизайна – снижение общего потребления памяти; они могут улучшать производительность в случае многозадачной среды в системе с ограниченной памятью. Встретив межпроцессное совместное использование, такие системы также пытаются избежать повторного запуска JIT-компиляции уже откомпилированного кода, что снижает общее время выполнения.

Если виртуальная память достаточно быстрая, то система может сделать код большим и оставить решение проблемы на усмотрение алгоритмов страничного обмена.

Использование кеша кода ограниченного размера может означать, что стандартный механизм страничного обмена для виртуальной памяти либо слишком медленный, либо слишком грубый для эффективной поддержки JIT-компилированного кода. Также это означает, что вытеснение из кеша кода приводит к отбрасыванию JIT-компилированного кода, а значит, разрыву его связи с исполняемой программой. Следовательно, при будущем вызове код придется выполнять в режиме эмуляции или откомпилировать заново.

Алгоритм замещения

JIT-компилятор должен записать откомпилированный код в кеш кода. Если программа управления кешем не может найти блок неиспользуемой памяти, достаточно большой для размещения кода, то должна вытеснить из кеша какой-то сегмент.

В кеше прямого отображения размер множества равен единице.

Замещение в кеше кода отличается от замещения в аппаратном кеше. Множественно-ассоциативный кеш определяет, на какое множество отображается новый блок, и вытесняет один из блоков этого множества. В литературе предлагается вытеснять последний использованный (LRU) блок; во многих аппаратных кешах используется или аппроксимируется LRU-замещение.

Алгоритмы управления кешем кода должны вытеснять достаточно сегментов, чтобы освободить место для вновь откомпилированного кода. В аппаратном кеше вытесняется одна строка кеша фиксированного размера. В программно-управляемом кеше кода выделение памяти производится на уровне гранулярности сегмента откомпилированного кода (трасса, метод или несколько методов). Это усложняет и политику, и реализацию алгоритма замещения.

Программа управления кешем должна вытеснять из кеша кода один или несколько сегментов, которые недавно не использовались. Вытесненные сегменты должны освободить достаточно места, чтобы новый код поместился, но при этом не расходовалось впустую «слишком много» памяти. Выбор LRU-сегмента мог бы быть неплохой отправной точкой, но потенциальная необходимость вытеснять несколько сегментов осложняет это решение. Если новый код требует вытеснения нескольких сегментов, то эти сегменты долж-

ны быть смежными. Следовательно, реализация механизма LRU требует дополнительной работы.

Последнее ограничение состоит в том, что алгоритмы замещения должны быть быстрыми; всё время, потраченное на их выполнение, прибавляется ко времени работы приложения. Творческий инженерный подход необходим для минимизации стоимости выбора замещаемого блока и сопровождения структур данных, поддерживающих это решение.

Фрагментация

Повторяющееся выделение памяти и замещение могут фрагментировать кеш кода. Для куч со сборкой мусора проблема фрагментации решается уплотнением; в кучах без сборки мусора нужно объединять соседние свободные блоки. У кешей кода нет команды `free`; в общем случае невозможно узнать, когда какой-то фрагмент кода будет исполняться в будущем и будет ли вообще.

Если система работает в среде виртуальной памяти, то может избежать части трудностей, связанных с управлением фрагментацией, если будет использовать больше виртуального адресного пространства, чем выделенная память. Пока рабочее множество кеша кода не выходит за границы предполагаемого размера кеша, штраф за использование лишнего виртуального адресного пространства должен быть минимален.

14.5.4. Управление изменениями исходного кода

Некоторые языки и системы позволяют вносить изменения в исходный код приложения прямо во время выполнения. В интерпретируемых средах обработать такие изменения сравнительно просто. Но если среда выполнения включает JIT-компилированный код, то системе необходим механизм, позволяющий распознать, когда изменение инвалидирует один или несколько фрагментов машинного кода, и заместить или перекомпилировать их.

> Это не новая идея. И APL в 1960-х годах, и Smalltalk в 1970-х предоставляли возможность редактировать исходный код. Но те системы были интерпретируемыми.

Среда выполнения нуждается в трех механизмах. Она должна понимать, что изменение имело место. Она должна идентифицировать код и данные, на которые изменение повлияло. Наконец, она должна привести текущее состояние выполнения в соответствие с новым исходным кодом.

Распознавание изменений

Система должна знать, когда исходный код изменился. Самый распространенный способ распознать изменения – ограничить механизмы их внесения. Например, для изменения JAVA-кода необходим загрузчик классов; в APL внесение изменений в код требовало использования оператора *кавычка*

в квадрате (quote-quad). Интерфейс, позволяющий вносить изменения, может уведомлять систему.

Идентификация области действия изменения

> Похожие проблемы возникают с межпроцедурной оптимизацией в AOT-компиляторе (см. раздел 8.7.3).

Система должна понимать, где произошло изменение. Если текст процедуры `fee` изменился, то машинный код `fee`, без сомнения, перестал быть действительным. Системе необходимо отображение имени процедуры на ее реализацию в машинном коде. Более тонкие проблемы возникают, когда изменение `fee` влияет на другие процедуры или методы.

Если, например, JIT-компилятор раньше встраивал процедуру `fee` в вызвавшую ее процедуру `foe`, то изменение `fee` инвалидирует также прежнюю компиляцию `foe`. Если `fee` – хороший кандидат на встраивание, скажем размер ее кода меньше оговоренного в стандартном соглашении о связях, то изменение `fee` могло бы запустить целый каскад перекомпиляций. Отображение имен процедур на фрагменты кода становится многозначным.

Изменения интерфейса метода, например изменение состава параметров, должны инвалидировать как измененную процедуру, так и все вызывающие ее процедуры. Детали предварительной и заключительной последовательностей выводятся из интерфейса; если он изменяется, то, скорее всего, нужно изменить и эти последовательности.

Перекомпиляция откомпилированного кода

Как минимум система должна гарантировать, что будущие вызовы метода выполняют самый актуальный код. В смешанной среде, возможно, будет достаточно удалить JIT-компилированный код измененного метода и вернуться в режим эмуляции BM-кода. Когда метод снова станет горячим, система откомпилирует его. В платформенной среде система должна устроить все так, чтобы был откомпилирован новый код – агрессивно или при следующем вызове.

Чтобы упростить перекомпиляцию, автор JIT-компилятора мог бы добавить уровень косвенности для каждого вызова. Тогда предварительная последовательность ссылалась бы на фиксированный адрес вызываемой стороны; в этом месте JIT-компилятор сохраняет указатель на реальный код. Дополнительный уровень косвенности позволяет не искать все вызывающие стороны, чтобы обновить их указатели на код. Чтобы перекомпоновать метод, JIT-компилятору достаточно заменить один указатель.

В случае когда изменение кода инвалидирует результаты компиляции других процедур, количество инвалидаций может расти, но те же самые базовые механизмы продолжат работать.

Изменение объявлений

Изменение исходного кода во время выполнения привносит еще одну связанную проблему, которая возникает как для компиляции, так и для интер-

претации. Если исходный код может быть изменен, то могут изменяться и определения объектов данных. Рассмотрим, к примеру, изменение, которое добавляет новый член данных в класс. Если уже созданы объекты этого класса, то в них не будет нового члена. Исходный язык должен определить, как быть в такой ситуации, но в худшем случае системе предстоит найти и переформатировать все существующие экземпляры класса – потенциально дорогостоящая перспектива.

Чтобы упростить нахождение всех объектов класса, система может связать их в список. Первые SMALLTALK-системы производили с этой целью исчерпывающий поиск в памяти, и только благодаря ограниченной памяти таких систем этот подход был практически осуществим.

14.6. Итоги и перспективы

Системы JIT-компиляции производят оптимизацию и генерирование кода на этапе выполнения. При таком подходе JIT-компилятор может получить доступ к большему объему информации об именах, значениях и привязках. А эти знания, в свою очередь, могут помочь ему адаптировать код к условиям, складывающимся во время выполнения.

JIT-системы работают в условиях жестких временных ограничений. Хорошо спроектированная и реализованная система может дать значительные улучшения. Ускорение, обеспечиваемое JIT-компиляцией, должно компенсировать время, потраченное на сбор информации, принятие решений и компиляцию кода. Таким образом, авторам JIT-компиляторов необходимо хорошо разбираться в реализации языка и глубоко понимать методы компиляции и оптимизации.

Несмотря на долгую историю оптимизации на этапе выполнения, эта область все еще развивается. Например, один из самых активно используемых JIT-компиляторов, V8 JAVASCRIPT от Google, первоначально был написан как платформенная система с компиляцией в момент вызова. Накопленный опыт побудил переписать ее с использованием смешанного режима, когда компилируются только горячие методы. В обоснование этого изменения в литературе указывается на необходимость уменьшить занимаемую кодом память и время инициализации; кроме того, компиляция горячих методов позволила избежать разбора неиспользуемого кода. Изменения языков и сред выполнения, а также опыт эксплуатации направляли работы по оптимизации на этапе выполнения в последние десять лет. Мы предвидим, что развитие продолжится и в будущем.

Замечания к этой главе

У компиляции на этапе выполнения долгая история. Маккарти включил механизм такой компиляции в свою раннюю LISP-систему, чтобы обеспечить

эффективное выполнение кода, построенного во время выполнения, – прямое следствие унифицированного подхода LISP к коду и данным [266].

Томпсон использовал компиляцию «во время редактирования» регулярных выражений в машинный код для IBM 7094, чтобы создать мощный инструмент текстового поиска в перенесенной им версии редактора QED [345]; соответствующее построение описано в разделе 2.4.2.

Хансен построил свою систему Adaptive Fortran для исследования практичности и целесообразности оптимизации на этапе выполнения [188]. Она поддерживала подмножество FORTRAN IV и небольшой набор оптимизаций. Образцом для поведения системы послужили широко известные в то время компиляторы FORTRAN. Его диссертация содержит важное обсуждение того, как оценивать преимущества оптимизации и как запускать оптимизацию во время выполнения.

Реализация Smalltalk-80 Дойча–Шиффмана, созданная для ранней рабочей станции Sun Microsystems, продемонстрировала потенциал компиляции на этапе выполнения для улучшения динамических языков [137]; тогдашние реализации, опиравшиеся на интерпретацию байт-кода Smalltalk, работали медленнее.

Системы HotSpot Server [288] и Dynamo [32] были широко распространены и пользовались авторитетом. HotSpot продолжал оказывать влияние на проектные решения в системах на основе JIT-компиляции еще более десяти лет. Dynamo стала для целого поколения источником вдохновения при работе над различными задачами – от управления кешем кода до динамической трансляции программ.

Большинство оптимизаторов на уровне методов принимают форму глобальной нумерации значений. Эти алгоритмы варьируются от глобальных обобщений идей локальной нумерации значений [59, 167], алгоритмов, основанных на методе минимизации ДКА Хопкрофта [90, 312], до реализаций отложенного перемещения кода (см. замечания к главе 10).

Временные ограничения JIT-компиляции стали стимулом для применения эффективных алгоритмов. Селекторы команд на основе сопоставления с деревом-образцом можно сделать сверхэффективными: от пяти до десяти операций на этапе компиляции на одну сгенерированную операцию [162, 163, 297]. Распределение регистров методом линейного сканирования позволяет избежать самой дорогостоящей части полнофункционального распределителя на основе раскрашивания: построения графа интерференции [296]. В среде, где есть много мелких методов и не требуется выгружать код, линейное сканирование работает отлично. В компиляторе HotSpot Server Compiler обрезка графа интерференции использовалась, чтобы уменьшить стоимость полнофункционального распределителя на основе раскрашивания [93].

Упражнения

К разделу 14.2

1. Снова рассмотрим график на рис. 14.1 (масштабируемость JAVA-приложения с JIT-компиляцией и без нее). Как изменение величины порога

JIT-компиляции могло бы повлиять на поведение кривой с JIT, особенно ее нижнего отрезка, показанного на рисунке (b)?

К разделу 14.3

2. Напишите псевдокод обратного прохода по ациклической трассе, который обнаруживает мертвые и частично мертвые операции. Можете предполагать, что JIT-компилятор имеет информацию о множестве Live на каждом выходе из трассы.
Как JIT-компилятор мог бы получить такую информацию?

3. Один из вопросов при проектировании оптимизатора горячих трасс – как обрабатывать промежуточные входы в трассу. В разделе 14.3 промежуточные входы игнорируются, из-за чего создается несколько копий некоторых блоков.

В качестве альтернативы автор компонента мог бы поручить алгоритму построения трасс расщепить трассу в точке промежуточного входа. При такой стратегии генерировались бы оптимизированные трассы для участков до и после промежуточного входа. Затем алгоритм напрямую связал бы первую и последнюю части.

a. Как алгоритм построения трасс мог бы распознать, что блок является промежуточной точкой входа?
b. Эффективность каких видов JIT-оптимизаций могла бы снизиться в результате расщепления трасс в промежуточных точках входа?

4. Если у оптимизатора трасс кеш кода ограничен и оказывается заполненным, то ему придется вытеснить одну или несколько трасс.
a. Какие осложнения привносят связанные трассы?
b. Какие подходы к управлению вытеснением связанных трасс могли бы взять на вооружение алгоритмы управления кешем кода?

К разделу 14.4

5. Когда система с оптимизатором горячих методов обнаруживает, что какой-то метод запускает слишком много перекомпиляций, она может принять решение о деоптимизации этого метода.

JIT-компилятор мог бы обрабатывать отдельные места вызова по-разному, связывая каждое с подходящим образом оптимизированным кодом тела метода.

> Временно можете предполагать, что размер памяти для кода кеша – не проблема.

a. Какую информацию должна собирать система, чтобы такая оптимизация и деоптимизация, зависящая от места вызова, стала возможной?
b. Какие дополнительные структуры данных могли бы понадобиться JIT-компилятору на этапе выполнения, чтобы реализовать такую зависящую от места оптимизацию?
c. Одна из очевидных затрат такой схемы – место в кеше кода. Как автор компилятора мог бы ограничить размножение вариантов кода тел одного метода?

6. Некоторые JIT-компиляторы горячих методов компилируют код в отдельном потоке, асинхронно. Чем это может быть хорошо с точки зрения конечного пользователя? Какие неудобства это может создать?
7. Деоптимизация должна что-то делать с результатами встраивания. Предположим, что JIT-компилятор встроил `fie` в `fee`, а позже решил, что должен деоптимизировать `fee`. Какие стратегии может реализовать компилятор, чтобы упростить деоптимизацию метода, включающего встроенный код?
8. AOT-компиляторы выбирают, какие процедуры встраивать, основываясь на некоторой комбинации статического анализа с данными профилирования, собранными в ходе предшествующего прогона на «репрезентативных» данных. С другой стороны, JIT-компилятор принимает решение о встраивании процедуры, полагаясь почти исключительно на информацию из профиля, собранного на этапе выполнения.
   a. Предложите три эвристики, которыми JIT-компилятор горячих методов мог бы воспользоваться, чтобы определить, стоит или не стоит встраивать вызываемую сторону в данное место вызова.
   b. Какие виды данных могла бы собирать среда выполнения, чтобы помочь в принятии решения о том, стоит или не стоит встраивать данное место вызова.
   c. Сравните свои стратегии и ожидаемые от их применения результаты с результатами, которых вы ожидаете от AOT-компилятора.

Приложение A

ILOC

A.1. Введение

ILOC – линейный ассемблерный код для простой абстрактной RISC-машины. Использованный в этой книге вариант ILOC – упрощенная версия промежуточного представления (ПП) для проекта массово-скалярного компилятора (Massively Scalar Compiler Project – MSCP), разработанного в университете Райса. Например, в определенном здесь ILOC имеются все два базовых типа данных: целое неуказанной длины и одна литера. В компиляторе MSCP ПП поддерживало гораздо более широкий набор типов данных.

Абстрактная ILOC-машина имеет неограниченное количество регистров. Имеются операции типа регистр-регистр, операции загрузки и сохранения, сравнения и ветвления. Поддерживается четыре режима адресации: *прямой*, *с адресом и непосредственным операндом*, *с адресом и смещением* и *непосредственный*. Исходные операнды читаются в начале такта, в котором операция запущена. Результирующие операнды определяются в конце такта, в котором операция завершилась.

Прочие – помимо системы команд – детали машины оставлены неопределенными. В большинстве примеров в этой книге предполагается простая машина с одним функциональным блоком, которая выполняет операции ILOC в порядке их подачи. Если используются другие конфигурации, это оговаривается явно.

ILOC-программа представляет собой последовательный список команд:

```
IlocProgram     → InstructionList
InstructionList → Instruction
                | Instruction InstructionList
```

Команда состоит из одной или нескольких операций и факультативной метки:

```
Instruction → OperationList
            | label : OperationList
```

Метка – это алфавитно-цифровая строка, ([*A*–*Z*] |[*a*–*z*]) ([*A*–*Z*] |[*a*–*z*] |[0–9])*. Метка должна заканчиваться двоеточием. Если в каком-то месте кода должно быть несколько меток, мы вставляем в нужные места команды `nop`.

Список операций *OperationList* состоит либо из одной операции *Operation*, либо из заключенного в квадратные скобки списка операций, разделенных точкой с запятой:

OperationList → *Operation*
| [*ListOfOps*]
ListOfOps → *Operation*
| *Operation* ; *ListOfOps*
Operation → *NormalOp*
| *ControlFlowOp*

Операция ILOC соответствует одной машинной операции. *Operation* – это либо *NormalOp*, которая выполняет вычисление или перемещение данных, либо *ControlFlowOp*, которая изменяет поток управления в программе.

Операция *NormalOp* состоит из кода операций, списка исходных операндов через запятую и списка результирующих операндов через запятую. Исходные операнды отделяются от результатов символом ⇒, который произносится как «в».

NormalOp → *Opcode OperandList* ⇒ *OperandList*
OperandList → *Operand*
| *Operand* , *OperandList*
Operand → *register*
| *number*
| *label*

К сожалению, как и в настоящем языке ассемблера, связь между кодом операции и формой ее операндов не единообразная. Самый простой способ описать форму операндов каждой операции – табличный. В таблице в разделе A.6 приведен полный список кодов операций и их операндов для каждой операции ILOC, встречающейся в книге.

Операнды *Operand* могут быть трех типов: `register`, `number` и `label`. Тип каждого операнда определяется кодом операции и положением операнда внутри операции. В примерах используются как числовые (r_{10}), так и символические (r_i) имена регистров. `number` – просто положительные целые числа. Метки `label`, как было сказано выше, – алфавитно-цифровые строки, ([*A*–*Z*] |[*a*–*z*])([*A*–*Z*] |[*a*–*z*] |[0–9])*.

Хотя определение `label` допускает использование `r10` и `ri` в качестве меток, мы избегаем меток, которые могли бы быть интерпретированы по-другому.

У арифметических операций имеется единственный результат; некоторые операции `store`, а также все операции условного перехода имеют два результата.

Например, `storeAI`, операция сохранения *с адресом и непосредственным операндом*, имеет один исходный операнд (`register`) и два результирующих

(register и number). Операция storeAI $r_i \Rightarrow r_j,4$ прибавляет 4 к значению в регистре r_j, чтобы образовать адрес, а затем сохраняет значение r_i по этому адресу в памяти. Она выполняет действие MEMORY(r_j + 4) ← CONTENTS(r_i).

A.2. Соглашения об именовании

В ILOC-коде, используемом в примерах в тексте, действует простой набор соглашений об именовании.

1. Смещения переменных в памяти представляются путем дописывания в начало имени переменной литеры @.
2. Пользователь может предполагать, что запас регистров не ограничен. Они именуются с использованием целых чисел, например r_{1776}, или символических имен, например r_i.
3. Регистр r_{arp} зарезервирован для указателя на текущую запись активации. Таким образом, операция loadAI r_{arp}, @x ⇒ r_1 загружает целое число со смещением @x от начала текущей записи активации в регистр r_1.

Комментарии в ILOC начинаются парой литер // и продолжаются до конца строки. Мы предполагаем, что они удаляются сканером.

A.3. Вычислительные операции

Первая крупная категория операций в ILOC – набор базовых вычислительных операций. ILOC предлагает эти операции как в трехадресном формате регистр-регистр, так и в двухадресном формате с непосредственной константой во втором исходном операнде.

Операции типа регистр-регистр			
Код операции	**Исходные**	**Результаты**	**Семантика**
add	r_1,r_2	r_3	$r_1 + r_2 \Rightarrow r_3$
sub	r_1,r_2	r_3	$r_1 - r_2 \Rightarrow r_3$
mult	r_1,r_2	r_3	$r_1 \times r_2 \Rightarrow r_3$
div	r_1,r_2	r_3	$r_1 \div r_2 \Rightarrow r_3$
lshift	r_1,r_2	r_3	$r_1 \ll r_2 \Rightarrow r_3$
rshift	r_1,r_2	r_3	$r_1 \gg r_2 \Rightarrow r_3$

Первые четыре строки – стандартные арифметические операции, а последние две – операции логического сдвига.

В реальном ILOC-процессоре или в полнофункциональном компиляторе понадобилось более одного арифметического типа данных. Это потребовало бы типизированных или полиморфных кодов операций. В компиляторе MSCP были различные арифметические операции для целых разной длины, чисел с плавающей точкой и указателей.

Следующая группа кодов относится к операциям с регистром и непосредственным операндом. Каждая из них принимает регистр и непосредственную константу. У некоммутативных операций имеется две разные формы, позволяющие указывать константу в первом или во втором операнде. Например, `rsubI` вычитает значение в `r1` из непосредственной константы `c2`.

Операции типа регистр – непосредственный операнд			
Код операции	**Исходные**	**Результаты**	**Семантика**
`addI`	r_1, c_2	r_3	$r_1 + c_2 \Rightarrow r_3$
`subI`	r_1, c_2	r_3	$r_1 - c_2 \Rightarrow r_3$
`rsubI`	r_1, c_2	r_3	$r_1 - c_2 \Rightarrow r_3$
`multI`	r_1, c_2	r_3	$r_1 \times c_2 \Rightarrow r_3$
`divI`	r_1, c_2	r_3	$r_1 \div c_2 \Rightarrow r_3$
`rdivI`	r_1, c_2	r_3	$r_1 \div c_2 \Rightarrow r_3$
`lshiftI`	r_1, c_2	r_3	$r_1 \ll c_2 \Rightarrow r_3$
`rshiftI`	r_1, c_2	r_3	$r_1 \gg c_2 \Rightarrow r_3$

Операция с регистром и непосредственным операндом полезна по трем причинам:

- она позволяет компилятору или программисту на языке ассемблера использовать небольшие константы прямо в операции и тем самым уменьшает спрос на регистры. Она также может устранить операцию загрузки константы в регистр;
- она «читает» константу из потока команд, а значит, из кеша команд. У большинства процессоров имеются различные кеши и пути данных для команд и для данных; таким образом, операция с непосредственным операндом использует меньше ресурсов на стороне данных интерфейсов с памятью;
- в непосредственной форме операции значение константы хорошо видно и легко доступно, что помогает средствам оптимизации и генерирования кода увидеть и использовать это значение.

A.4. Операции перемещения данных

Вторая крупная категория операций в ILOC позволяет компилятору или программисту на языке ассемблера определять перемещение данных. В этот раздел мы включили две специальные операции: условного перемещения и представления φ-функции (см. раздел 4.6.2).

Операции с памятью

ILOC предоставляет набор операций для перемещения значений между памятью и регистрами: операции загрузки и сохранения. В этой книге мы ограничились целыми и литерными данными; в полной версии ILOC понадо-

бились бы операции загрузки и сохранения для гораздо более широкого спектра базовых типов.

Для порождения эффективного кода фаза синтеза должна эффективно использовать режимы адресации в операциях `load` и `store`. ILOC предоставляет целый ряд таких режимов.

> В большинстве процессоров используется отдельный сумматор для выполнения арифметических операций с адресами. Поэтому перемещение сложения в режим адресации освобождает слот запуска в другом функциональном блоке.

Операции загрузки			
Код операции	**Исходные**	**Результаты**	**Семантика**
`load`	r_1	r_2	MEMORY(r_1) ⇒ r_2
`loadAI`	r_1,c_2	r_3	MEMORY(r_1 + c_2) ⇒ r_1
`loadAO`	r_1,r_2	r_3	MEMORY(r_1 + r_2) ⇒ r_3
`cload`	r_1	r_2	`load` для литеры
`cloadAI`	r_1,c_2	r_3	`loadAI` для литеры
`cloadAO`	r_1,r_2	r_3	`loadAO` для литеры
`loadI`	c_1	r_2	c_1 ⇒ r_2

Операции загрузки различаются поддерживаемыми режимами адресации.

- В операциях *прямой загрузки*, `load` и `cload`, в качестве адреса используется значение в r_1.
- В операциях с адресом и непосредственным операндом, `loadAI` и `cloadAI`, для образования адреса используются константа и значение в исходном регистре.
- В операциях с адресом и смещением, `loadAO` и `cloadAO`, для образования адреса используется содержимое двух исходных регистров.
- Операция *непосредственной загрузки*, `loadI`, перемещает константу в целевой регистр.

В полном ПП типа ILOC с несколькими базовыми типами данных понадобилась бы своя операция непосредственной загрузки для каждого базового типа.

Операции сохранения поддерживают такие же режимы адресации.

Операции сохранения			
Код операции	**Исходные**	**Результаты**	**Семантика**
`store`	r_1	r_2	r_1 ⇒ MEMORY(r_2)
`storeAI`	r_1	r_2,c_3	r_1 ⇒ MEMORY(r_2 + c_3)
`storeAO`	r_1	r_2,r_3	r_1 ⇒ MEMORY(r_2 + r_3)
`cstore`	r_1	r_2	`store` для литеры
`cstoreAI`	r_1	r_2,c_3	`storeAI` для литеры
`cstoreAO`	r_1	r_2,r_3	`storeAO` для литеры

В ILOC нет операции непосредственного сохранения.

Операции копирования типа регистр-регистр

Для перемещения значений из одного регистра в другой ILOC включает набор операций копирования типа регистр-регистр.

i2i	r_1	r_2	$r_1 \Rightarrow r_2$ для целых значений
c2c	r_1	r_2	$r_1 \Rightarrow r_2$ для литерных значений
c2i	r_1	r_2	преобразовать литеру в целое
i2c	r_1	r_2	преобразовать целое в литеру

i2i и c2c просто копируют значение из одного регистра в другой. А c2i и i2c не только копируют значение, но и преобразуют его в другой тип.

Операция условного перемещения

Многие ISA предоставляют операцию условного перемещения, которая выбирает значение из одного из двух исходных регистров, исходя из булева значения в третьем регистре. Для поддержки примеров, в которых обсуждается этот вид операций, ILOC включает операции условного перемещения с четырьмя операндами и тем самым позволяет компилятору избежать ветвлений для очень простых условных конструкций.

Операции условного перемещения			
Код операции	**Исходные**	**Результаты**	**Семантика**
c_i2i	r_b,r_1,r_2	r_3	$r_1 \Rightarrow r_3$, *если* r_b = true $r_2 \Rightarrow r_3$ *в противном случае*
c_c2c	r_b,r_1,r_2	r_3	$r_1 \Rightarrow r_3$, *если* r_b = true $r_2 \Rightarrow r_3$ *в противном случае*

Представление φ-функций

Строя SSA-форму, компилятор должен представлять φ-функции. В ILOC естественный способ записи φ-функции дает операция

```
phi r_i, r_j, r_k ⇒ r_m
```

для φ-функции $r_m \leftarrow \varphi(r_i, r_j, r_k)$. В силу самой природы SSA-формы операция phi может принимать произвольное число исходных операндов и всегда определяет один целевой регистр. Представление φ-функций таким способом порождает ряд интересных инженерных проблем при реализации ПП.

A.5. Операции потока управления

Операции потока управления позволяют ILOC-программе упорядочивать выполнение помеченных блоков операций. ILOC включает операции безусловного перехода, сравнения и условного перехода. Для поддержки об-

суждения в разделе 7.4.2 ILOC также включает синтаксис для представления предикатных операций.

Хотя операции потока управления имеют трехадресную форму, как и большинство операций ILOC, смысл операндов в них иной. Для наглядности в операциях, включающих передачу управления (или присваивание указателю команд), используется одиночная стрелка, чтобы отделить исходные операнды от целевых.

```
ControlFlowOp → jumpI            → label
              | jump             → register
              | Branch register  → label , label
              | Compare register, register ⇒ register
```

ILOC предлагает два стиля записи операций *Branch* и *Compare*, чтобы в разделе 7.4.2 можно было обсудить трансляцию управляющих конструкций для каждого из них.

Безусловные переходы

ILOC поддерживает обе операции: *непосредственный переход* и *переход по регистру*. В большинстве примеров используется непосредственный переход, потому что в нем явно указана целевая метка.

Код операции	Исходные	Результаты	Семантика
jumpI	--	l_1	$l_1 \rightarrow$ PC
jump	--	r_1	$r_1 \rightarrow$ PC

Операция jump неоднозначна, потому что целевой адрес находится в регистре. Компилятор может не справиться с выводом правильного множества возможных целевых меток для jump. Это затрудняет построение ГПУ и может снизить качество информации, которую способен вывести анализ потока данных. По этим причинам мы отдаем предпочтение jumpI перед jump всюду, где это возможно и практично.

Иногда компилятор вынужден генерировать jump, а не jumpI. Для таких ситуаций в ILOC имеется псевдооперация, которая позволяет компилятору запомнить множество возможных меток для операции jump. У псевдооперации tbl два аргумента: регистр и непосредственная метка.

Код операции	Исходные	Результаты	Семантика
tbl	r_1,l_2	--	r_1 *мог бы хранить* l_2

Операция tbl может находиться только сразу после jump. Последовательность

```
jump          → ri
tbl   ri, L01
tbl   ri, L03
tbl   ri, L05
tbl   ri, L08
```

утверждает, что целями `jump` могут быть метки `L01`, `L03`, `L05`, `L08` и никакие другие.

Сравнения и условные переходы

В ILOC у всех ветвей имеется метка для пути, по которому идет выполнение, когда условие истинно, и для пути – когда условие ложно. В ILOC не существует понятия «неявной» ветви, по которой выполнение «проваливается».

ILOC поддерживает две модели условных переходов. В стандартной модели ILOC, показанной на рис. A.1(a), используется простая операция условного перехода `cbr` и набор из шести булевых операций сравнения. `cbr` проверяет булево значение, порожденное сравнением, и переходит на соответствующую непосредственную метку.

Код операции	Исходные	Результаты	Семантика	
cmp_LT	r_1,r_2	r_3	true ⇒ r_3,	*если* $r_1 < r_2$
			false ⇒ r_3	*в противном случае*
cmp_LE	r_1,r_2	r_3	true ⇒ r_3,	*если* $r_1 \le r_2$
			false ⇒ r_3	*в противном случае*
cmp_EQ	r_1,r_2	r_3	true ⇒ r_3,	*если* $r_1 = r_2$
			false ⇒ r_3	*в противном случае*
cmp_GE	r_1,r_2	r_3	true ⇒ r_3,	*если* $r_1 \ge r_2$
			false ⇒ r_3	*в противном случае*
cmp_GT	r_1,r_2	r_3	true ⇒ r_3,	*если* $r_1 > r_2$
			false ⇒ r_3	*в противном случае*
cmp_NE	r_1,r_2	r_3	true ⇒ r_3,	*если* $r_1 \ne r_2$
			false ⇒ r_3	*в противном случае*
cbr	r_1	l_2,l_3	l_2 → PC,	*если* r_1 = true
			l_3 → PC	*в противном случае*

(a) Стандартный синтаксис операции условного перехода

Код операции	Исходные	Результаты	Семантика	
comp	r_1,r_2	cc_3	*устанавливает* cc_3	
cbr_LT	cc_1	l_2,l_3	l_2 → PC,	*если* cc_3 = LT
			l_3 → PC	*в противном случае*
cbr_LT	cc_1	l_2,l_3	l_2 → PC,	*если* cc_3 = LE
			l_3 → PC	*в противном случае*
cbr_EQ	cc_1	l_2,l_3	l_2 → PC,	*если* cc_3 = EQ
			l_3 → PC	*в противном случае*
cbr_GE	cc_1	l_2,l_3	l_2 → PC,	*если* cc_3 = GE
			l_3 → PC	*в противном случае*
cbr_GT	cc_1	l_2,l_3	l_2 → PC,	*если* cc_3 = GT
			l_3 → PC	*в противном случае*
cbr_NE	cc_1	l_2,l_3	l_2 → PC,	*если* cc_3 = NE
			l_3 → PC	*в противном случае*

(b) Синтаксис, моделирующий ISA с кодами условий

Рис. A.1 Синтаксис операции условного перехода

Во второй модели условного перехода, показанной на рис. A.1(b), имеется одна операция сравнения и набор из шести операций условного перехода. Она моделирует те ISA, в которых сравнение устанавливает «код условия» (см. раздел 7.4.2). В этой модели выбор реляционного оператора перенесен в саму операцию ветвления.

В примерах мы обозначаем цель `comp` как регистр кода условия, записывая его в виде cc_i. Операция `comp` записывает в cc_i одно из значений {`LT`, `LE`, `EQ`, `GE`, `GT`, `NE`}. Для каждого из шести значений имеется свой вариант операции условного перехода.

Существующие ISA различаются числом поддерживаемых регистров кодов условий.

Предикатное выполнение

Предикация – еще одно средство, введенное для того, чтобы позволить компилятору или программисту на языке ассемблера избежать условных переходов. Предикатная операция принимает булево значение, которое определяет, вступил ли в силу результат операции. В ILOC предикатная операция записывается следующим образом:

(r_p) ? `add` $r_1, r_2 \Rightarrow r_3$

Эта нотация означает, что в r_3 записывается сумма значений r_1 и r_2, если r_p содержит значение `true`. Если r_p не содержит `true`, то r_3 не изменяется.

В принципе, любую операцию ILOC можно сделать предикатной. Но на практике для некоторых операций, например `phi` или `jump`, предикация не имеет смысла. (Предикатная `jump` – это просто `cbr`, для которой в случае ложности условия выполняется следующая операция.)

A.6. Таблицы кодов операций

Здесь и далее приведены таблицы, содержащие список всех операций ILOC с описанием их семантики. Они предназначены для справочных целей.

Арифметические операции			
Код операции	**Исходные**	**Результаты**	**Семантика**
`nop`	*нет*	*нет*	*однотактовая операция без операндов и побочных эффектов*
`add`	r_1, r_2	r_3	$r_1 + r_2 \Rightarrow r_3$
`sub`	r_1, r_2	r_3	$r_1 - r_2 \Rightarrow r_3$
`mult`	r_1, r_2	r_3	$r_1 \times r_2 \Rightarrow r_3$
`div`	r_1, c_2	r_3	$r_1 \div r_2 \Rightarrow r_3$
`addI`	r_1, r_2	r_3	$r_1 + c_2 \Rightarrow r_3$
`subI`	r_1, r_2	r_3	$r_1 - c_2 \Rightarrow r_3$
`rsubI`	r_1, r_2	r_3	$c_2 - r_1 \Rightarrow r_3$
`multI`	r_1, r_2	r_3	$r_1 \times c_2 \Rightarrow r_3$

продолжение таблицы ⇒

Арифметические операции			
Код операции	**Исходные**	**Результаты**	**Семантика**
divI	r_1,r_2	r_3	$r_1 \div c_2 \Rightarrow r_3$
rdivI	r_1,r_2	r_3	$c_2 \div r_1 \Rightarrow r_3$
lshift	r_1,r_2	r_3	$r_1 \ll r_2 \Rightarrow r_3$
lshiftI	r_1,c_2	r_3	$r_1 \ll c_2 \Rightarrow r_3$
rshift	r_1,r_2	r_3	$r_1 \gg r_2 \Rightarrow r_3$
rshiftI	r_1,c_2	r_3	$r_1 \gg c_2 \Rightarrow r_3$
and	r_1,r_2	r_3	$r_1 \wedge r_2 \Rightarrow r_3$
andI	r_1,c_2	r_3	$r_1 \wedge c_2 \Rightarrow r_3$
or	r_1,r_2	r_3	$r_1 \vee r_2 \Rightarrow r_3$
orI	r_1,c_2	r_3	$r_1 \vee c_2 \Rightarrow r_3$
xor	r_1,r_2	r_3	r_1 *xor* $r_2 \Rightarrow r_3$
xorI	r_1,c_2	r_3	r_1 *xor* $c_2 \Rightarrow r_3$

Операции перемещения данных			
Код операции	**Исходные**	**Результаты**	**Семантика**
loadI	c_1	r_2	$c_1 \Rightarrow r_2$
load	r_1	r_2	MEMORY(r_1) $\Rightarrow r_2$
loadAI	r_1,c_2	r_3	MEMORY(r_1+c_2) $\Rightarrow r_3$
loadAO	r_1,r_2	r_3	MEMORY(r_1+r_2) $\Rightarrow r_3$
cload	r_1	r_2	load *для целых*
cloadAI	r_1,c_2	r_3	loadAI *для целых*
cloadAO	r_1,r_2	r_3	loadAO *для целых*
store	r_1	r_2	$r_1 \Rightarrow$ MEMORY(r_2)
storeAI	r_1	r_2,c_3	$r_1 \Rightarrow$ MEMORY(r_2+c_3)
storeAO	r_1	r_2,r_3	$r_1 \Rightarrow$ MEMORY(r_2+r_3)
cstore	r_1	r_2	store *для целых*
cstoreAI	r_1	r_2,c_3	storeAI *для целых*
cstoreAO	r_1	r_2,r_3	storeAO *для целых*
i2i	r_1	r_2	$r_1 \Rightarrow r_2$ *для целых*
c2c	r_1	r_2	$r_1 \Rightarrow r_2$ *для целых*
c2i	r_1	r_2	*преобразовать литеру в целое*
i2c	r_1	r_2	*преобразовать целое в литеру*
phi	r_1,r_2,r_3	r_4	$\varphi(r_1,r_2,r_3) \Rightarrow r_4$ *с произвольным числом источников*
c_i2i	r_b,r_2,r_3	r_4	$r_1 \Rightarrow r_3$, *если* r_b = true $r_2 \Rightarrow r_3$ *в противном случае*
c_c2c	r_b,r_2,r_3	r_4	$r_1 \Rightarrow r_3$, *если* r_b = true $r_2 \Rightarrow r_3$ *в противном случае*

Безусловные переходы			
Код операции	**Исходные**	**Результаты**	**Семантика**
jump	--	r_1	$r_1 \rightarrow$ PC
jumpI	--	l_1	$l_1 \rightarrow$ PC
tbl	r_1,l_2	--	r_1 *мог бы содержать* l_2

Стандартный синтаксис условных переходов			
Код операции	**Исходные**	**Результаты**	**Семантика**
cmp_LT	r_1, r_2	r_3	true ⇒ r_3, *если* $r_1 < r_2$ false ⇒ r_3 *в противном случае*
cmp_LT	r_1, r_2	r_3	true ⇒ r_3, *если* $r_1 \le r_2$ false ⇒ r_3 *в противном случае*
cmp_EQ	r_1, r_2	r_3	true ⇒ r_3, *если* $r_1 = r_2$ false ⇒ r_3 *в противном случае*
cmp_GE	r_1, r_2	r_3	true ⇒ r_3, *если* $r_1 \ge r_2$ false ⇒ r_3 *в противном случае*
cmp_GT	r_1, r_2	r_3	true ⇒ r_3, *если* $r_1 > r_2$ false ⇒ r_3 *в противном случае*
cmp_NE	r_1, r_2	r_3	true ⇒ r_3, *если* $r_1 \ne r_2$ false ⇒ r_3 *в противном случае*
cbr	r_1	l_2, l_3	l_2 → PC, *если* r_1 = true l_3 → PC *в противном случае*

Альтернативный синтаксис условных переходов			
Код операции	**Исходные**	**Результаты**	**Семантика**
comp	r_1, r_2	cc_3	*устанавливает* cc_3 *в одно из* {LT, LE, EQ, GE, GT, NE}
cbr_LT	cc_1	l_2, l_3	l_2 → PC, *если* cc_3 = LT l_3 → PC *в противном случае*
cbr_LT	cc_1	l_2, l_3	l_2 → PC, *если* cc_3 = LE l_3 → PC *в противном случае*
cbr_EQ	cc_1	l_2, l_3	l_2 → PC, *если* cc_3 = EQ l_3 → PC *в противном случае*
cbr_GE	cc_1	l_2, l_3	l_2 → PC, *если* cc_3 = GE l_3 → PC *в противном случае*
cbr_GT	cc_1	l_2, l_3	l_2 → PC, *если* cc_3 = GT l_3 → PC *в противном случае*
cbr_NE	cc_1	l_2, l_3	l_2 → PC, *если* cc_3 = NE l_3 → PC *в противном случае*

Приложение B

Структуры данных

B.1. Введение

Компилятор состоит из множества частей. Поэтому если мы хотим, чтобы компилятор оказался успешным, необходимо уделять внимание многочисленным деталям. В этом приложении описываются некоторые проблемы, связанные с алгоритмами и их реализацией, которые возникают при построении компилятора. В большинстве случаев эти детали могли бы отвлечь нас от темы обсуждения в основном тексте. Мы собрали их в этом приложении, где их можно рассмотреть, как они того заслуживают.

Это приложение посвящено инженерным вопросам, возникающим при проектировании и реализации инфраструктуры поддержки компилятора. От того, как автор компилятора решит эти вопросы, в значительной степени зависит скорость работы компилятора, а также легкость его расширения и сопровождения.

Во многих приложениях программисту имеет смысл положиться на реализации структур данных в стандартных библиотеках. Необычайная стабильность JAVA и ширящееся распространение PYTHON сделали этот подход практичным и популярным. Однако когда некоторая процедура выполняется достаточно часто, улучшение, достигнутое благодаря специальным структурам данных, может оказаться существенным. В масштабе таких компиляторов, как GCC или LLVM, накопительный эффект вполне может оправдать усилия, потраченные на реализацию специальных представлений множества, графа, отображения или другой структуры данных.

В качестве примера проблем, возникающих при компиляции, рассмотрим размер собственных структур данных компилятора. Компилятор не может знать размер некоторых из этих структур, пока не прочтет входную программу; следовательно, фазу анализа следует проектировать, так чтобы размер ее структур данных мог постепенно увеличиваться в процессе обработки больших файлов. Однако отсюда вытекает, что последующие проходы компилятора должны знать приблизительные размеры большинства своих внутренних структур данных; фаза анализа может запомнить эти размеры для последующих проходов. Если фазе анализа понадобилось 10 000 имен

в ПП программы, то компилятор не должен начинать следующий проход с таблицей символов, рассчитанной на 512 имен. Заголовок любого внешнего представления ПП должен включать спецификацию приблизительных размеров основных структур данных.

Далее, последующие проходы компилятора могут предполагать, что предъявленное им ПП программы было сгенерировано этим компилятором. Они, конечно, должны обнаруживать все ошибки, но сообщения о них могут быть более краткими, чем ожидается от фазы анализа. Общая стратегия заключается в том, чтобы написать контрольный проход, который производит тщательную проверку ПП программы и может быть вставлен между другими проходами для отладки, а если речь не идет об отладке компилятора, то допустимо полагаться на менее усердное обнаружение ошибок. Однако авторы компиляторов всегда должны помнить, что именно они, скорее всего, будут смотреть на код между проходами. Усилия, направленные на то, чтобы сделать внешние представления ПП более удобочитаемыми, часто вознаграждают именно тех людей, которые тратили на них свое время.

Стрелка в ILOC – пример применения этого принципа. Она делает поток значений явным и устраняет один из источников путаницы при чтении ILOC-кода.

В качестве последнего примера рассмотрим реализацию разреженного графа. Эта задача возникает при оптимизации и генерировании кода; граф зависимостей планировщика и граф интерференции распределителя регистров обычно оказываются разреженными. Разработчик мог бы взять на вооружение представление разреженного графа с помощью хеш-таблицы. Графы зависимостей для локального планирования, как правило, невелики (ограничены размером блока), но графы интерференции могут вырастать до таких размеров, что потребление памяти становится серьезной проблемой.

Идея проста. Каждое ребро является парой (*начало*, *конец*). Хешировав пары, мы сможем реализовать быструю проверку принадлежности. В хеш-таблице можно хранить записи для каждой пары, определяющей ребро. При условии что хеш-функция ведет себя правильно, эта схема дает представление, занимающее немного места в памяти и обладающее хорошим асимптотическим поведением по времени.

Однако автор компилятора должен тщательно выбирать хеш-функцию, которая хорошо работает с ребрами. Если *начало* и *конец* – небольшие целые числа, то у хеш-функции может оказаться слишком мало битов информации для полезного использования. Опыт показывает [111], что представления графа интерференции на основе хеш-таблиц сильно зависят от качества хеш-функции.

B.2. Представление множеств

Многие задачи компиляции формулируются в терминах множеств. Они встречаются во многих местах книги, например построение подмножеств (глава 2), построение канонического набора множеств LR(1)-элементов (гла-

ва 3), анализ потока данных (главы 8 и 9) и рабочие списки, такие как очередь готовых в задаче планирования списка (глава 12). В каждом контексте автор компилятора должен выбрать подходящее представление множества. Во многих случаях от этого зависит эффективность алгоритма (см., например, основанный на множествах *IDom* алгоритм в разделе 9.5.2).

Фундаментальное различие между построением компилятора и других видов системного программного обеспечения, например операционной системы, заключается в том, что многие проблемы компиляции можно решать в офлайновом режиме. Например, локальный алгоритм распределения регистров, описанный в главе 13, более широко известен как алгоритм MIN Белади для офлайнового замещения страниц; он уже давно используется как эталон, с которым сравнивается эффективность алгоритмов онлайнового замещения страниц. В сообществе разработчиков операционных систем этот алгоритм представляет лишь академический интерес, поскольку работает в режиме офлайн. Так как операционная система не может знать, какие страницы понадобятся в будущем, офлайновые алгоритмы для нее не годятся. Но в компиляторе офлайновый алгоритм практически полезен, потому что компилятор может просмотреть весь блок, прежде чем принимать какие-то решения.

Офлайновая природа компиляции позволяет автору компилятора использовать широкий спектр представлений множеств. Были исследованы многие представления. В частности, офлайновая компиляция часто позволяет ограничить элементы множества S универсальным множеством фиксированного размера U ($S \subseteq U$). Это, в свою очередь, дает возможность использовать более эффективные представления множеств, чем в онлайновой ситуации, когда размер U определяется динамически.

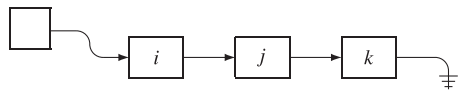
К типичным теоретико-множественным операциям относятся *member*, *insert*, *delete*, *clear*, *choose-one*, *cardinality*, *forall*, *copy*, *compare*, *union*, *intersect*, *difference* и *complement*. В конкретном приложении обычно используется лишь малая часть этих операций. Стоимость операции зависит от выбранного представления множеств. При выборе эффективного представления для конкретного приложения важно учитывать, насколько часто будут использоваться операции каждого типа. Из других факторов отметим требования представления к памяти и ожидаемую разреженность S относительно U.

Далее в этом разделе мы сосредоточимся на трех представлениях множеств, часто применяемых авторами компиляторов: упорядоченные связанные списки, битовые векторы и разреженные множества. В последнем подразделе кратко обсуждается роль хеширования в реализации множеств.

B.2.1. Представление множеств упорядоченными списками

Если размер каждого множества мал, иногда имеет смысл использовать простое представление в виде связанного списка. Для множества S это представление состоит из связанного списка и указателя на его первый элемент.

Каждый узел списка содержит представление одного элемента S и указатель на следующий узел. В последнем узле указатель содержит стандартное значение, обозначающее конец списка. При таком представлении реализация может ввести отношение порядка на элементах, создав упорядоченный список. Например, упорядоченный связанный список для множества $S = \{i, j, k\}$, $i < j < k$, концептуально мог бы выглядеть так:



Элементы хранятся в порядке возрастания. Размер представления S пропорционален числу элементов S, а не размеру U. Если $|S|$ много меньше $|U|$, то экономия от представления только элементов, присутствующих в S, может многократно превысить дополнительные затраты на хранение указателя в каждом узле.

Представление в виде списка особенно гибкое. Поскольку ничто в списке не зависит ни от размера U, ни от размера S, его можно использовать в ситуациях, когда компилятор выявляет U, S или оба. Такие ситуации возникают в фазе анализа, когда компилятор выполняет какие-то действия, еще не видя входа целиком, а также в местах, где одно пространство имен транслируется в другое, как при построении интервалов жизни в распределителе регистров.

В таблице на рис. B.1 показаны асимптотические сложности типичных операций над множествами при использовании этого представления. Большинство операций над упорядоченным списком занимают время $O(|S|)$, потому что для их выполнения необходимо обходить списки. Если освобождение памяти не требует обхода списка для освобождения узлов элементов, как в системе со сборкой мусора или на основе арены, то операция *clear* занимает постоянное время.

member	$O(\|S\|)$	$O(1)$	$O(1)$
insert	$O(\|S\|)$	$O(1)$	$O(1)$
delete	$O(\|S\|)$	$O(1)$	$O(1)$
clear	$O(1)$	$O(\|U\|)$	$O(1)$
choose-one	$O(1)$	$O(\|U\|)$	$O(1)$
cardinality	$O(\|S\|)$	$O(\|U\|)$	$O(1)$
forall	$O(\|S\|)$	$O(\|U\|)$	$O(\|S\|)$
copy	$O(\|S\|)$	$O(\|U\|)$	$O(\|S\|)$
compare	$O(\|S\|)$	$O(\|U\|)$	$O(\|S\|)$
union	$O(\|S\|)$	$O(\|U\|)$	$O(\|S\|)$
intersect	$O(\|S\|)$	$O(\|U\|)$	$O(\|S\|)$
difference	$O(\|S\|)$	$O(\|U\|)$	$O(\|S\|)$
complement	–	$O(\|U\|)$	$O(\|U\|)$

Рис. B.1 Асимптотическая временная сложность операций над множествами

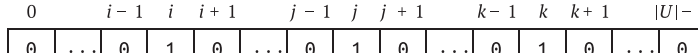
Если размер универсального множества неизвестен, но может оказаться достаточно большим, то автор компилятора может уменьшить накладные расходы на хранение указателей и выделение памяти, создав узлы, содержащие несколько элементов множества. В каждом узле имеется место для k элементов и счетчика, в котором хранится количество занятых элементов. Тогда для построения множества из n требуется $\left\lceil \frac{n}{k} \right\rceil$ выделений памяти, $\left\lceil \frac{n}{k} \right\rceil + 1$ указателей и $\left\lceil \frac{n}{k} \right\rceil + 1$ счетчиков, тогда как множество, реализованное списком одноэлементных узлов, потребовало бы n выделений памяти и $n + 1$ указателей.

Но у такой схемы есть цена. Для вставки и удаления может понадобиться переместить больше данных, чем в случае одноэлементных узлов. Впрочем, асимптотическая сложность остается равной $O(|S|)$. Хорошая реализация может ограничить такое тасование k элементами при любой вставке или удалении ценой непроизводительной траты части памяти.

Массив *IDoms*, который мы использовали для быстрого вычисления доминирования (см. раздел 9.5.2), – изобретательное применение спискового представления множеств в специальном случае. А именно компилятор знает размер универсального множества и количество множеств. Компилятор также знает, что упорядоченные множества обладают тем свойством, что если $e \in S_1$ и $e \in S_2$, то каждый элемент, следующий за e в S_1, также является элементом S_2. Таким образом, элементы, начиная с e, могут быть общими. При использовании представления с помощью массива имена элементов можно также использовать как указатели. Следовательно, один массив n элементов может представлять n разреженных множеств. Для таких множеств возможна также быстрая реализация оператора пересечения.

B.2.2. Представление множеств битовыми векторами

Авторы компиляторов часто используют для представления множеств *битовые векторы*, особенно в анализе потока данных (см. разделы 8.6.1 и 9.2). Для ограниченного универсального множества U множество $S \subseteq U$ можно представить битовым вектором длины $|U|$, который называется *характеристическим вектором S*. Для любого $i \in U$, $0 \leqslant i < |U|$, если $i \in S$, то i-й элемент характеристического вектора равен 1. В противном случае i-й элемент равен 0. Например, характеристический вектор множества $S \subseteq U$ такого, что $S = \{i, j, k\}$, $i < j < k$, имеет вид:



При представлении в виде битового вектора всегда выделяется достаточно места для представления всех элементов U, поэтому его можно использовать только в приложениях, где U известно, т. е. в офлайновых приложениях.

В таблице на рис. B.1 приведены асимптотические сложности типичных операций над множествами, представленными таким образом. Хотя сложность многих из них равна O(|*U*|), представление все равно будет эффективным, если *U* мало. В одном слове хранится много элементов; налицо кратный выигрыш по сравнению с представлениями, требующими одного слова на каждый элемент. Так, например, если размер слова равен 64 бита, то для представления универсального множества, содержащего 64 или меньше элементов, потребуется всего одно слово.

Компактность представления отражается на скорости операций. Для множеств, помещающихся в одно слово, многие теоретико-множественные операции сводятся к одной машинной команде; например, операция *union* становится логическим ИЛИ, а операция *intersection* – логическим И. Даже если для представления множества требуется несколько слов, количество машинных команд, требуемых для выполнения многих операций, уменьшается в число раз, равное размеру машинного слова.

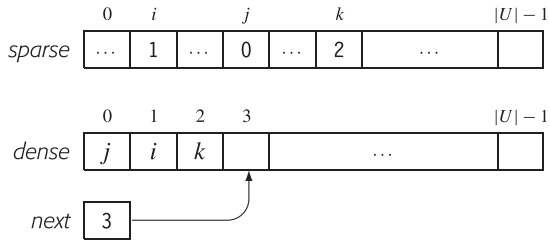
B.2.3. Представление разреженных множеств

Для фиксированного универсального множества *U* множество $S \subseteq U$ является разреженным, если |*S*| много меньше |*U*|. Некоторые множества, встречающиеся при компиляции, разрежены. Например, множества LiveOut, используемые при распределении регистров, обычно разрежены. Авторы компиляторов часто используют для представления таких множеств битовые векторы в силу их эффективности как по памяти, так и по времени. Но если степень разреженности достаточно велика, то возможны более эффективные по времени представления, особенно в ситуациях, когда значительную долю операций можно выполнить за время O(1) или O(|*S*|). С другой стороны, те же операции над множествами, представленными битовыми векторами, требуют времени O(1) или O(|*U*|). Если отношение |*U*| к |*S*| больше размера слова, то битовые векторы будут менее эффективны.

В одном из разреженных представлений множеств, обладающих этими свойствами, используются два вектора длины |*U*| и скаляр. Первый вектор, *sparse*, содержит разреженное представление множества, второй вектор, *dense*, – плотное представление множества. Скаляр, *next*, содержит индекс позиции в *dense*, в которую можно вставить следующий новый элемент. Разумеется, в *next* также хранится мощность множества.

В момент создания разреженного множества ни один вектор не нужно инициализировать; проверки на принадлежность множеству проверяют действительность каждой записи при доступе к ней. Операция *clear* просто сбрасывает *next* в ноль – его начальное значение. Чтобы добавить элемент $i \in U$ в *S*, код (1) сохраняет *i* в позиции *dense* с индексом *next*, (2) сохраняет значение *next* в *i*-й позиции *sparse* и (3) увеличивает *next*, так что теперь это индекс следующей позиции в *dense*, куда можно вставить элемент.

Если бы мы начали с пустого разреженного множества S и добавляли элементы j, i и k в порядке $i < j < k$, то множество перешло бы в следующее состояние:



Заметим, что для представления разреженного множества нужно достаточно места, чтобы представить всё U. Следовательно, его можно использовать только в офлайновых ситуациях, когда компилятору известен размер U.

Для разреженного множества существует краткий тест на принадлежность. Действительные записи для элемента i в *sparse* и *dense* должны указывать друг на друга, так что элемент i принадлежит множеству тогда и только тогда, когда

$$0 \le sparse[i] < next \quad \text{и} \quad dense[sparse[i]] = i.$$

В таблице на рис. В.1 приведены асимптотические сложности типичных операций над множествами, представленными таким образом. Поскольку эта схема включает и разреженное, и плотное представление множества, ей свойственны некоторые преимущества того и другого. К элементу множества можно обратиться за время O(1) через *sparse*, а те операции, которые должны обходить множества, могут использовать *dense* – сложность при этом составит $O(|S|)$.

При выборе между представлениями в виде битового вектора и разреженного множества нужно учитывать как пространственную, так и временную сложность. Для разреженного множества нужны два вектора длины $|U|$ и скаляр. Для битового вектора достаточно одного вектора длины $|U|$. Как показано на рис. В.1, разреженное множество превосходит битовый вектор с точки зрения асимптотической временной сложности. Однако благодаря эффективным реализациям операций над множествами, представленными битовыми векторами, это представление предпочтительнее в случаях, когда S не разрежено. Выбирая одно из двух представлений, важно иметь в виду как разреженность, так и относительную частоту выполнения различных операций.

В.2.4. Роль хеш-таблиц

Отображение текстовых имен, например имен в исходном коде, сгенерированных компилятором временных имен или виртуальных регистров, на внутренние имена, используемые в реализации множества, – один из самых

важных аспектов реализации. Один из эффективных способов – создать хеш-таблицу для этого отображения, в которой ключом является внешнее имя, а значением – внутреннее. Временная сложность такого отображения – искомое O(1).

Для имени *x* внутренним именем могла бы быть позиция бита, представляющего его в битовом векторе, или его индекс в векторе *dense* разреженного множества.

B.3. Реализация ПП

Созданное компилятором ПП – центральная структура данных. Наряду со вспомогательными структурами ПП представляет транслируемую программу. Компилятор многократно обходит эти структуры, анализирует их содержимое и переписывает его. Таким образом, реализация ПП играет важную роль в общей эффективности компилятора.

В главе 4 был представлен обзор различных промежуточных представлений, используемых в компиляторах. Здесь же мы остановимся на деталях реализации ПП. Часть материала основана на общеизвестной информации, а часть – на личном опыте авторов и их коллег. В разделах B.3.1 и B.3.2 рассматриваются вопросы реализации графовых и линейных ПП соответственно.

Собираясь вместе, авторы компиляторов часто обмениваются байками о том, что Д. Р. Чейз называл «тупыми компиляторными трюками».

Автор компилятора должен обращать внимание на общий размер ПП программы и на простоту его обхода компилятором. То и другое влияет на производительность. Использование большего объема памяти, чем необходимо, привносит дополнительные затраты, особенно в плане локальности и сборки мусора. В худшем случае это может ограничить размер входных программ, которые компилятор способен обработать. С трудностью обхода ПП связаны отдельные виды неэффективности.

B.3.1. Непосредственные графовые представления

Компиляторы используют разнообразные графовые ПП (см. раздел 4.3). Подстраивание реализации графа под специфические нужды компилятора может улучшить как удобство работы с ним, так и эффективность использования памяти. В этом разделе описываются некоторые вопросы, возникающие при реализации дерева и графа.

Представление деревьев

Очевидным представлением дерева в большинстве языков является множество узлов, соединенных указателями. В типичной реализации узлы выделяются по мере необходимости в процессе построения дерева. Автор компонента вправе использовать узлы разного размера, быть может, с разным

количеством сыновних узлов или с разными полями данных. Или же можно строить дерево, содержащее узлы только одного вида, рассчитанные на максимальный размер.

Альтернативное представление – массив узлов. Для связи между узлами можно использовать индексы или указатели (при условии что в языке имеется оператор взятия адреса *address-of*). При таком дизайне узлы по необходимости имеют одинаковый размер, но в остальном он похож на реализацию с указателями.

Решения, влияющие на размер ПП

Автор компилятора должен обращать внимание на общий размер ПП программы. Чем больше ПП, тем медленнее работает компилятор; в общем случае компилятор работает с каждым байтом выделенной им памяти. В худшем случае лишнее место в ПП может привести к проблемам с локальностью и ограничить размер программ, которые компилятор может без труда обработать.

Внимание к деталям при проектировании ПП может способствовать ограничению размера ПП:

- ***представлять все факты компактно.*** Имена следует преобразовать в небольшие целые числа в фазе анализа и транслировать обратно в строки для отладки и вывода. Структуры данных следует размещать так, чтобы избежать необходимости дополнения (см. раздел 5.6.5);
- ***представлять в ПП только общезначимые факты.*** Факты, которые нужны лишь на нескольких проходах или для нескольких операций, следует выносить во вспомогательные структуры. Само присутствие такой дополнительной информации можно отметить в отдельном поле, например в знаковом бите кода операции или типа узла;
- ***устранять ссылки на мертвые вспомогательные структуры.*** Если в среде с управляемой памятью компилятор выделил место для вспомогательных структур, то должен перезаписать все явные указатели на них, когда необходимость отпадет. Сохранение таких указателей помешает убрать эти структуры в мусор.

Чтобы оценить размер ПП, автор компилятора может встроить факультативный механизм отчетности в абстракцию ПП и периодически использовать его, чтобы получить представление об эффективности использования памяти. В своих проектах мы часто обнаруживали поля, которые используются редко или не используются вовсе.

У каждой схемы есть слабые и сильные стороны. Компромиссы следует оценивать в контексте конкретного проекта и языка его реализации.

- Схема на основе узлов и указателей способна обработать сколь угодно большие деревья естественным образом. В случае массива узлов необходим код для расширения массива, когда дерево становится больше первоначально отведенного для него размера.
- В схеме на основе узлов и указателей требуется выделять память для каждого узла. Механизм выделения на основе арены может уменьшить стоимость освобождения памяти (см. врезку на рис. 332).

- Для схемы на основе узлов и указателей локальность зависит исключительно от поведения применяемого распределителя памяти во время выполнения. В случае массива узлов используется большой блок непрерывной памяти, поэтому соседние узлы не конфликтуют друг с другом в кеше.
- Схему на основе узлов и указателей может быть сложнее отлаживать, чем на основе массива. Программисты обычно считают индексы массива интуитивно более понятными, чем адреса в памяти.

Автор компилятора легко может обобщить схему на основе узлов и указателей, смягчив проблемы, связанные со стоимостью выделения памяти и локальностью, если воспользуется механизмом блочно-непрерывного распределения памяти.

Блочно-непрерывное распределение упоминалось в главе 4 на стр. 207.

Блочно-непрерывный распределитель для дерева мог бы использовать главный и дополнительный распределители. Дополнительный распределитель хранит список свободных узлов, первоначально пустой, и, будучи вызван, возвращает узел из списка свободных, если тот не пуст. Типичный случай обрабатывается очень быстро. Когда список свободных пуст, вызывается главный распределитель, который получает блок памяти для хранения k узлов. Он строит в этой памяти список свободных блоков и возвращает один из них вызывающей стороне. Код построения дерева всегда вызывает дополнительный распределитель.

Увеличение k в пределах разумного уменьшает стоимость выделения памяти в расчете на один узел.

Отображение произвольных деревьев в двоичные

Прямолинейная реализация абстрактного синтаксического дерева могла бы поддерживать узлы с различным числом непосредственных потомков. Например, для типичного заголовка цикла

```
for i = 1 to n by 2
```

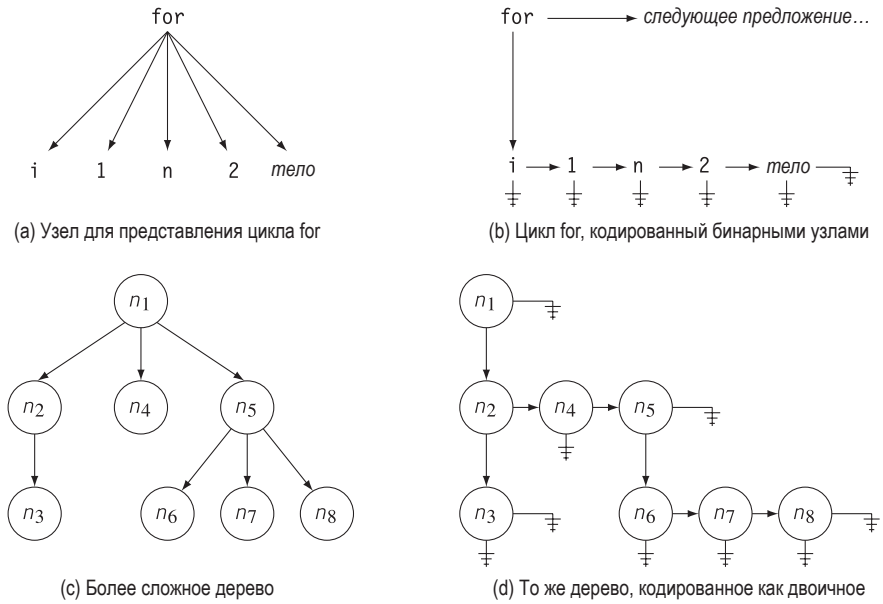
в АСД мог бы существовать узел с пятью сыновними узлами, как на рис. B.2(a). Узел с меткой `тело` представляет поддерево кода в теле цикла `for`.

Есть конструкции, для которых никакое фиксированное число потомков не подойдет. Чтобы представить вызов процедуры, АСД должно либо выделять узлы по числу параметров, либо использовать один сыновний узел, содержащий список параметров. Та же проблема возникает для φ-функции в ПП на основе формы SSA. Наличие узлов нескольких размеров усложняет код обхода АСД; в каждом узле должно быть указано, сколько у него непосредственных потомков, а код обхода должен посещать каждого потомка. Использование списка параметров добавляет накладные расходы на выделение памяти, построение списка и его обход.

Чтобы упростить реализацию деревьев, автор компилятора может отобразить естественную форму АСД – произвольное дерево с узлами разной

Идея отображать произвольные деревья в двоичные принадлежит Кнуту [240, том 1].

арности – в единообразное двоичное дерево, в котором каждый узел имеет ровно два сыновних узла. При таком отображении поле левого потомка в узле указывает на самый левый сыновний узел, а поле правого потомка – на следующего брата на том же уровне дерева. Правые сыновние узлы образуют списки братьев.



(a) Узел для представления цикла for

(b) Цикл for, кодированный бинарными узлами

(c) Более сложное дерево

(d) То же дерево, кодированное как двоичное

Рис. В.2 Отображение произвольных деревьев в двоичные

На рис. В.2(b) показан заголовок цикла `for` на рисунке (a), кодированный в виде двоичного дерева. Но у простоты двоичного дерева есть цена; например, в каждом левом узле имеется явный нулевой указатель. Заметим также, что в узле `for` имеется указатель на первое предложение, следующее за циклом; в оригинальной версии этот указатель, без сомнения, находился бы в родителе узла `for`. На рисунках (c) и (d) показан более сложный пример.

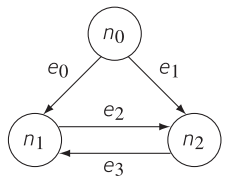
Использование двоичных деревьев упрощает реализацию в нескольких отношениях. Для единообразных узлов проще выделять память, они будут хорошо работать с распределителем на основе арены или блочно-непрерывным. Код обхода дерева простой и единообразный, в нем нет проблем из-за наличия нескольких форматов узлов. Но за простоту приходится расплачиваться дополнительными нулевыми указателями в листовых узлах.

Выделение памяти из арены обсуждается во врезке на стр. 332.

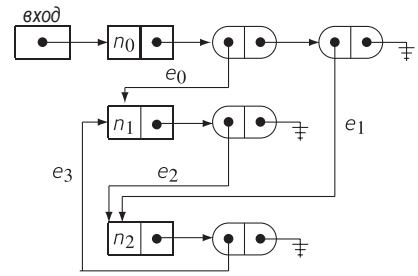
Представление произвольных графов

Компиляторы должны также представлять произвольные графы, например графы потока управления (ГПУ) и графы зависимостей. При реализации гра-

фов возникают те же проблемы, что для деревьев. В простых реализациях можно было бы использовать вершины, выделенные из кучи, в которых ребра представлены указателями. На рис. B.3(a) показан простой ГПУ. Ясно, что для него нужно три вершины. Трудности возникают из-за ребер: сколько входящих и исходящих ребер должно быть в каждой вершине? В каждой вершине можно было бы хранить список исходящих ребер; этот подход ведет к реализации, показанной на рис. B.3(b).



(a) Граф потока управления



(b) Графовое представление

Таблица вершин

Имя	Преемник	Предшественник
n_0	e_0	—
n_1	e_2	e_0
n_2	e_3	e_1

Таблица ребер

Имя	Начало	Конец	Следующий преемник	Следующий предшественник
e_0	n_0	n_1	e_1	e_3
e_1	n_0	n_2	—	e_2
e_2	n_1	n_2	—	—
e_3	n_2	n_1	—	—

(c) Табличное представление

Рис. B.3 Табличное представление ГПУ

На рис. B.3(b) прямоугольники представляют вершины, а овалы – ребра. При таком представлении легко обойти граф, следуя в направлении ребер. Если компилятору необходим произвольный доступ к вершинам, то можно добавить массив указателей на вершины, индексированный целочисленными именами вершин. При таком дополнении (на рисунке не показано) граф будет пригоден для решения прямых задач о потоке данных. Оно дает быстрый механизм нахождения всех преемников данной вершины.

Это представление никак не облегчает прохода по ребрам в обратном направлении, что необходимо для решения обратной задачи о потоке данных. Для обратного прохода нужен быстрый поиск предшественника. Автор компилятора мог бы добавить список предшественников в каждую вершину, заплатив за это дополнительным расходом времени и памяти.

Альтернатива – представить граф парой таблиц: вершин и ребер. Для данной вершины таблица вершин содержит имена ребер, ведущих к первому преемнику и к первому предшественнику. В таблице ребер имеется запись для каждого ребра, она содержит имена его начала и конца. Кроме того,

у каждого ребра имеется поле, содержащее имя следующего преемника его начала и следующего предшественника его конца. Благодаря этому компилятор для каждой вершины знает список ее преемников и список ее предшественников.

На рис. B.3(c) показан ГПУ на рисунке (a) в этой табличной форме. Хотя таблицы читаются не так легко, как представления на рисунках (a) и (b), они дают быстрый доступ к преемникам, предшественникам и отдельным вершинам и ребрам по их именам (в предположении, что имена кодируются небольшими целыми числами).

Табличное представление хорошо работает для приложений, где нужно обходить граф. На случай, когда компилятор часто использует другие операции, существуют и другие представления. Например, у распределителя регистров на основе раскрашивания графа есть две операции над графом доминирования: перебор соседей вершины и проверка наличия ребра между двумя вершинами (см. раздел 13.4.2). Эти операции предполагают другие представления; большинство распределителей на основе раскрашивания для быстрого перебора соседей пользуются списками смежности, а для ответа на вопрос о принадлежности ребра (*i*, *j*) графу – битовой матрицей.

Поскольку графы интерференции большие и разреженные, для векторов смежности может потребоваться слишком много места. В некоторых реализациях граф строится за два прохода – на первом вычисляется размер каждого вектора смежности, а на втором строятся сами векторы, минимального размера каждый. В других реализациях используется вариант спискового представления множеств из раздела B.2.1 – граф строится за один проход, а вектор смежности представляется неупорядоченным списком, в каждом узле которого хранится несколько ребер.

B.3.2. Линейные промежуточные формы

Концептуально линейная промежуточная форма напоминает таблицу или массив. Различные примеры ILOC-кода, приведенные в книге, изображены в виде таблиц. У линейных ПП есть неявный порядок; операции выполняются по порядку, пока не встретится условный переход, безусловный переход или вызов процедуры. Эту абстрактную модель легко визуализировать и анализировать. Однако на практике компиляторы пользуются реализациями более сложными, чем простая таблица.

Компилятор манипулирует линейным ПП разными способами. Он может изменять значения полей в операции; например, распределитель регистров изменяет имена регистров в операциях. Во время прохода компилятор может добавлять, удалять или перемещать операции. Парсер добавляет код в конец ПП, распределитель регистров вставляет код выгрузки между существующими операциями. В процессе устранения мертвого кода операции удаляются. В процессе отложенного перемещения кода операции могут быть перемещены из середины одного блока в конец другого. Планировщик изменяет порядок операций.

Манипуляции объектом JAVA `ArrayList` могут характеризоваться сложностью $O(n^2)$, если вставки и удаления производятся часто.

В реализации, где используются простые структуры типа массива, некоторые из этих манипуляций могут сопровождаться неожиданными затратами. Вставки и удаления могут быть реализованы путем сдвига операций, чтобы освободить или повторно использовать место. Расширение массива может повлечь за собой копирование операций в новый объект в куче. Внимание к таким затратам оказывает прямое влияние на стоимость компиляции.

Проход анализа должен знать размер ПП, чтобы можно было заранее выделить всю необходимую память.

Автор компилятора может использовать различные стратегии реализации на разных проходах. В фазе анализа целью может быть минимизация накладных расходов на добавление операций в конец ПП. Проход, на котором просто анализируется поток данных, вполне может обойтись заранее выделенным массивом структур. Планировщику нужна возможность изменять порядок операций, а распределителю регистров – возможность вставлять и удалять их.

Реализация операций

Как и в случае узлов дерева, для линейного ПП существует две базовые операции: выделение памяти для отдельных структур и для массива структур. В обоих случаях структура представляет одну операцию ПП, но в первом отдельные структуры выделяются в куче и связываются указателями (или ссылками), а во втором – структуры выделяются блоками в одном или нескольких массивах, связать их можно с помощью либо индексов массива (если массив один), либо указателей. Плюсы и минусы такие же, как в предыдущем подразделе.

- Если отдельные структуры связаны, то компилятор легко может изменить порядок обхода. В случае реализации массива структур можно использовать обход в порядке индексов, что позволяет сэкономить на указателях, однако при такой схеме увеличиваются затраты на реорганизацию кода.
- В случае массива структур обход в порядке индексов, скорее всего, будет иметь дополнительные преимущества, связанные с пространственной локальностью и такими стандартными оптимизациями, улучшающими плотный код, как устранение избыточности, перемещение кода и снижение стоимости операций (см. главы 8 и 10).
- В случае структур, выделяемых индивидуально, стоимость выделения памяти выше, чем для массива. В управляемой среде выделение памяти в куче для нескольких небольших объектов может увеличить стоимость сборки мусора.

Линейные формы хорошо приспособлены для нескольких стратегий реализации. На проходе, где размер ПП известен и статичен, автор компилятора

может выделить один массив структур, достаточный для хранения всего ПП. Если обхода в порядке индексов достаточно, как в случае одного блока, то порядок может быть неявным. Если порядок операций может изменяться, то автор компилятора может добавить в каждую операцию указатели, чтобы создать однократно или дважды связанные списки и получить возможность обходить ПП в произвольном порядке.

Стратегии компенсации недостатков массива

Простота реализации ПП с использованием одного массива структур, который обходится в порядке индексов, весьма привлекательна. Если автор компонента выберет этот подход, то сможет использовать стратегии для компенсации его недостатков.

Вставка. Эффективную вставку можно реализовать с помощью оператора «объезда» `detour`. Чтобы выполнить вставку перед операцией *i*, компилятор создает сегмент, содержащий новые операции, за которым следуют копия операции *i* и оператор `detour`, возвращающий на выполнение операцию *i* + 1. Затем он заменяет операцию *i* оператором `detour`, переадресующим выполнение на новый сегмент кода.

Удаление. Эффективно выполнить удаление просто. Компилятор может использовать оператор `ignore`, который пропускает операцию, переходя на следующую. Чтобы удалить *i*-ю операцию, компилятор перезаписывает ее оператором `ignore`.

Если компилятор пользуется этими стратегиями компенсации, то должен выделить дополнительное место в массиве ПП, в котором будут размещаться вставленные сегменты кода.

В конце каждого прохода, когда компилятор записывает ПП на внешний носитель для следующего прохода, он может линеаризовать все операторы `detour` в нужном порядке и удалить все `ignore`. Разумеется, компилятор мог бы линеаризовать код и в середине прохода, если это почему-либо необходимо.

Если ожидается, что размер ПП будет изменяться, то автор компилятора может воспользоваться распределителем на основе арены или блочно-непрерывным. При такой реализации обычно необходимы указатели для связывания операций, поскольку они обеспечивают прозрачный способ связывания массивов структур, выделенных независимо.

Переменный размер узлов

Некоторые операции не помещаются в стандартную структуру, выделенную для операции. Например, количество исходных операндов φ-функции в форме SSA зависит от числа входящих ребер в графе потока управления; их может быть 2, а может быть и 32. В реализации, где для связывания отдельных операций используются указатели, можно просто выделить память для вариантной структуры и связать ее с ПП. В реализации с помощью массива структур можно использовать оператор `detour`, как предлагается во врезке выше. Конечно, код манипуляции ПП должен понимать все варианты и уметь их обрабатывать.

B.4. Реализация хеш-таблиц

Ввиду широкой доступности библиотек для таких языков, как JAVA и PYTHON, большинство студентов полагаются на библиотечные реализации хеш-таблиц, или отображений. Однако при определенных обстоятельствах необходима специальная реализация. В таком повсеместно используемом компиляторе, как GCC или LLVM, экономия, достигаемая благодаря специальной реализации, может быть значительной – особенно если подумать, сколько актов компиляции выполняется ежеминутно в мире. В других случаях специальная реализация может воспользоваться известными свойствами множества ожидаемых ключей (скажем, однолитерные строки или пары небольших целых чисел).

В этом разделе обсуждается несколько проблем, возникающих при реализации специальной хеш-таблицы. В разделе B.4.1 описываются две хеш-функции, которые на практике дают хорошие результаты. В разделах B.4.2 и B.4.3 представлены две популярные стратегии разрешения коллизий: *метод цепочек* и *метод открытой адресации*. В разделе B.4.4 рассматриваются вопросы управления памятью для хеш-таблиц.

B.4.1. Выбор хеш-функции

Пожалуй, самое важное решение при проектировании хеш-таблицы – выбор хеш-функции. Если хеш-функция порождает плохое распределение значений индекса, то средняя стоимость вставки и поиска увеличивается. По счастью, в литературе документировано несколько хороших хеш-функций, в т. ч. мультипликативные хеш-функции, описанные Кнутом, и универсальные хеш-функции, описанные Корменом и др.

Поведение отображения в худшем случае

Поиск в таблице символов может, а иногда и демонстрирует поведение в худшем случае. В начале 1980-х годов авторы работали над фазой анализа для векторизующего компилятора FORTRAN – PFC. Автор первой версии сканера PFC использовал для символьной таблицы несбалансированное двоичное дерево.

К сожалению, в нескольких важных приложениях, которые использовались для тестирования PFC, переменные в предложениях объявления перечислялись в алфавитном порядке. Поэтому сканер строил дерево, в котором большинство правых поддеревьев были пусты; поиск в таблице символов быстро вырождался в линейный, т. е. начинал занимать время $O(n)$ на одну операцию.

Чтобы исправить ситуацию, одному студенту было поручено построить хеш-таблицу. Для вычисления хеша реализация разбивала имя переменной на четырехбайтовые порции и комбинировала их операцией ИСКЛЮЧАЮЩЕЕ ИЛИ. Получающееся число по модулю размера таблицы использовалось в качестве индекса.

Увы, студент взял таблицу размера 2048, поэтому операция деления по модулю возвращала младшие 11 разрядов 32-разрядного слова. Хеш-функция дополняла короткие строки пробелами до полного слова. Следовательно, все одно- или двух-

буквенные имена порождали один и тот же ключ хеширования: младшие 11 разрядов «ЙЙ». Поскольку приложение содержало много переменных с однобуквенными именами, скорость поиска в таблице символов снова оказалась близка к O(*n*). После того как размер таблицы был сделан равным одному из чисел Мерсенна ($2^n - 1$), проблему удалось решить. В PFC много лет использовалась хеш-таблица с начальным размером 2047.

Мультипликативные хеш-функции

Мультипликативная хеш-функция обманчиво проста. Программист выбирает одну константу *C* и использует ее в следующей формуле:

$$h(key) = \lfloor TableSize \cdot ((C \cdot key) \bmod 1) \rfloor,$$

где *key* – целое число, используемое как ключ таблицы, а *TableSize* – текущий размер хеш-таблицы. Кнут предлагает следующее значение *C*:

$$0.6180339887 \approx \frac{\sqrt{5}-1}{2}.$$

Эта функция вычисляет $C \cdot key$, берет ее дробную часть с помощью функции mod и умножает результат на размер таблицы.

Универсальные хеш-функции

В одном прогоне программы, пользующейся хеш-функцией, константы одни и те же, но от прогона к прогону они могут изменяться.

Чтобы реализовать *универсальную хеш-функцию*, программист проектирует семейство функций, параметризованных небольшим набором констант. Во время выполнения значения констант выбираются случайным образом – для этого либо берутся случайные числа, либо случайный индекс в массив заранее протестированных констант. Поскольку при каждом выполнении программы хеш-функция изменяется, эта схема порождает разные распределения при каждом запуске. В контексте компилятора маловероятно, что если для одной отдельно взятой компиляции некоторой входной программы наблюдалось патологическое поведение, то такое же поведение будет наблюдаться и дальше. Чтобы реализовать универсальную версию мультипликативной хеш-функции, автор компилятора может случайным образом генерировать подходящее значение *C* в начале каждой компиляции.

B.4.2. Метод цепочек

В *методе цепочек* (*open hashing*) предполагается, что хеш-функция *h* порождает коллизии. Множество входных ключей разбивается функцией *h* на фиксированное число сегментов (*bucket*). Каждый сегмент содержит линейный список записей, по одной на имя. Функция *LookUp*(*n*) пробегает по линейному списку, хранящемуся в сегменте, соответствующему *h*(*n*), чтобы найти *n*.

Следовательно, *LookUp* требует одного вычисления $h(n)$ и обхода линейного списка. Вычисление $h(n)$ должно быть быстрым, а обход списка занимает время, пропорциональное длине списка. Для таблицы размера S, содержащей N имен, стоимость одного поиска должна быть приближенно равна $O(N/S)$. При условии что h распределяет имена приблизительно равномерно, а отношение числа имен к числу сегментов, N/S, мало, эта стоимость аппроксимирует нашу цель: время O(1) на каждый доступ.

На рис. B.4(a) показана небольшая хеш-таблица, реализованная по такой схеме. Предполагается, что h(a) = h(d) = 6, т. е. имеет место коллизия. Таким образом, a и d попадают в одну и ту же позицию таблицы и связываются в список. Функция вставки *Insert* должна добавлять новый элемент в начало списка – так быстрее.

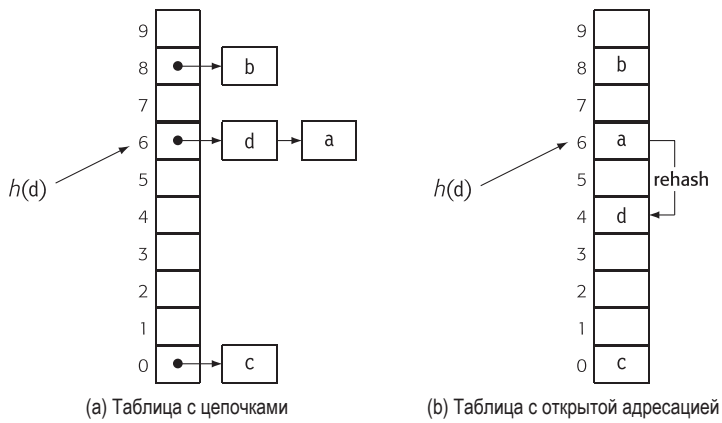


Рис. B.4 Организации хеш-таблиц

Метод цепочек имеет несколько достоинств. Поскольку при вставке каждого имени он создает новый узел в одном из связанных списков, он может обработать сколь угодно много имен без исчерпания памяти. Большое количество записей в одном сегменте не влияет на стоимость доступа к другим. Поскольку конкретной реализацией множества сегментов обычно является массив указателей, накладные расходы при увеличении S малы – один указатель для каждого добавленного сегмента. Поэтому поддерживать отношение N/S малым обходится дешево.

Основные недостатки метода цепочек неотделимы от этих достоинств. И с обоими можно справиться.

1. Метод цепочек может часто выделять память. При вставке каждого элемента создается новая запись. В системе с тяжеловесным механизмом выделения памяти затраты на реализацию могут быть заметны. Но использование облегченных механизмов, например арены или блочно-непрерывного распределителя, позволяет снизить затраты.
2. Если какой-либо список становится слишком большим, то *LookUp* вырождается в линейный поиск. При наличии хорошей функции хеши-

рования это происходит, только когда N много больше S. Реализация должна обнаруживать эту проблему и увеличивать массив сегментов. Как правило, это означает, что нужно выделить память для нового массива и заново вставить все записи из старой таблицы в новую.

Хорошо реализованная хеш-таблица с цепочками обеспечивает эффективный доступ с небольшими пространственными и временными издержками.

Чтобы улучшить поведение линейного поиска в одном сегменте, компилятор может динамически переупорядочивать цепочку. Ривест и др. [310, 329] описывает две эффективные стратегии: при каждом успешном поиске перемещать найденный узел вверх по цепочке на одну позицию или сразу в ее начало. Возможны и более сложные схемы организации сегментов. Однако автор компилятора должен оценить время, которое тратится на обход сегмента, прежде чем прилагать серьезные усилия к решению этой проблемы.

B.4.3. Метод открытой адресации

Метод *открытой адресации*, или *повторного хеширования*, обрабатывает коллизии, вычисляя другой индекс для имен, чья позиция, имеющая индекс $h(n)$, уже занята. В этом случае *LookUp*(n) вычисляет $h(n)$ и смотрит на позицию с этим индексом. Если она пуста, то *LookUp* завершается неудачно. Если *LookUp* видит n, значит, все хорошо. Если же она видит имя, отличное от n, то вызывает другую функцию $g(n)$, которая вычисляет инкремент поиска и далее осматривает позиции таблицы с индексами $(h(n) + g(n)) \bmod S$, $(h(n) + 2 \times g(n)) \bmod S$, $(h(n) + 3 \times g(n)) \bmod S$ и т. д., пока либо не найдет n, либо не натолкнется на пустую позицию, либо снова вернется к $h(n)$. (Позиции таблицы пронумерованы от 0 до $S - 1$, поэтому деление по модулю S всегда возвращает правильный индекс.) Если *LookUp* находит пустую позицию или повторно возвращается к $h(n)$, значит, поиск завершился неудачно.

На рис. B.4(b) показана небольшая хеш-таблица, реализованная по этой схеме. Данные в ней такие же, как на рис. B.4(a). Как и раньше, h(`a`) = h(`d`) = 6, тогда как h(`b`) = 8 и h(`c`) = 0. Во время вставки `d` произошла коллизия с `a`. Вторичная функция хеширования g(`d`) вернула 8, поэтому *Insert* поместила `d` в позицию таблицы с индексом 4 ((6 + 8) mod 10 = 4). По сути дела, метод открытой адресации строит цепочки элементов, как и метод цепочек, только цепочки эти хранятся прямо в таблице. Одна позиция таблицы может быть началом нескольких цепочек, для каждой из которых g вычисляет разные инкременты.

В этой схеме имеется тонкий компромисс между памятью и скоростью. Поскольку все ключи хранятся в таблице, S должно быть больше N. Если h и g порождают хорошие распределения, то коллизии случаются нечасто, цепочки повторного хеширования остаются короткими, а стоимость доступа – низкой. Поскольку затраты на пересчет g малы, в этой схеме нет необходимости хранить указатели для формирования цепочек повторного хеширования, так что мы экономим память под N указателей. Компилятор может распо-

рядиться этой памятью, чтобы увеличить таблицу и тем самым сократить количество коллизий. Основное достоинство открытой адресации понятно: более низкая стоимость доступа за счет более коротких цепочек.

Но у открытой адресации есть два важных недостатка. Оба проявляются, когда N приближается к S и таблица оказывается заполненной.

1. Поскольку цепочки повторного хеширования прошивают таблицу, коллизия между n и m может помешать последующей вставке какого-то третьего имени p. Если $h(n) = h(m)$ и $(h(m) + g(m)) \bmod S = h(p)$, то вставка n и последующая вставка m заполнят позицию, по праву принадлежащую p. Когда $|S| \gg |N|$, эта проблема незаметна. Но по мере того, как N приближается к S, она становится весьма существенной.
2. Поскольку S должно быть не меньше N, таблицу следует расширять, когда N становится слишком большим. (Реализация могла бы также расширять S, если какая-то цепочка оказывается слишком длинной.) В случае открытой адресации расширение необходимо, чтобы алгоритм оставался корректным; в случае метода цепочек расширение – лишь вопрос эффективности.

В некоторых реализациях вместо g используется константа. Хотя при таком подходе устраняются затраты на вычисление $g(n)$, есть и минус – для каждого значения h создается единственная цепочка повторного хеширования. Кроме того, цепочки повторного хеширования сливаются, когда при вторичной индексации обнаруживается уже занятая позиция таблицы. Эти два недостатка, пожалуй, перевешивают экономию на вычислении $g(n)$. Лучше использовать мультипликативную хеш-функцию с разными константами для h и g, которые выбираются на этапе инициализации из таблицы констант.

Размер таблицы S играет важную роль в методе открытой адресации. *LookUp* должна понимать, что дошла до уже посещенной ранее позиции, иначе она не остановится в случае неудачного поиска. Чтобы сделать это эффективно, реализация должна гарантировать, что рано или поздно вернется к $h(n)$. Если S – простое число, то при любом выборе $0 < g(n) < S$ генерируется последовательность проб, $p_1, p_2, \ldots, p_S$ такая, что $p_1 = p_S = h(n)$ и $p_i \neq h(n)$, $\forall\ 1 < i < S$. То есть *LookUp* проверит каждую позицию таблицы, прежде чем вернется к $h(n)$. Поскольку реализации, возможно, придется расширять таблицу, она должна включать таблицу простых чисел подходящего размера. На практике хватает небольшого набора простых чисел, обусловленного реалистичными ограничениями на размер программы и доступную компилятору память.

B.4.4. Хранение записей символов

Ни метод цепочек, ни открытая адресация не решают напрямую проблему выделения памяти для информации, ассоциированной с записями хеш-таблиц. Предположим, что компилятор использует структуру для хранения атрибутов каждого символа. В случае метода цепочек возникает искушение поместить эти структуры прямо в узлы цепочек, а случае открытой адреса-

ции – в индексированную таблицу. Но у обоих подходов есть недостатки. Отдельный стек структур позволяет достичь лучших результатов.

На рис. В.5 показана такая реализация. Указатели цепочек можно хранить в стеке, чтобы не выделять память для них по отдельности. В случае открытой адресации цепочки повторного хеширования по-прежнему неявно хранятся в индексированной таблице, благодаря чему экономится место – стремление, которое и привело к появлению этой идеи.

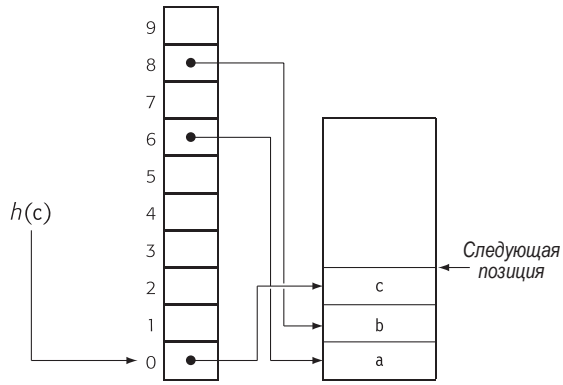


Рис. В.5 Выделение памяти для записей в стеке

Разумеется, автор компилятора может связать записи стека и воспользоваться либо распределителем на основе арены, либо блочно-непрерывным распределителем, чтобы решить проблему роста стека.

Когда фактические записи хранятся в стеке, они образуют плотную таблицу. В случае тяжеловесного механизма выделения памяти эта схема позволяет разнести стоимость выделения одного большого участка памяти на много записей. В среде со сборкой мусора это уменьшает количество объектов, подлежащих пометке и сборке. Если компилятор записывает таблицу на внешний носитель, то может воспользоваться блочным вводом-выводом. Наконец, плотная таблица может обеспечить лучшую локальность в процессе обхода таблицы символов компилятором – например, когда компилятор назначает каждому символу место в памяти.

И последнее преимущество – эта схема упрощает расширение множества индексов. Компилятор может отбросить старое множество, выделить память для нового, а затем повторно вставить записи в новую таблицу, просматривая стек, начиная снизу. Этот процесс устраняет необходимость одновременно хранить в памяти старое и новое множества индексов. В процессе обхода плотной таблицы компилятор не заходит в пустые позиции.

B.5. Гибкий дизайн таблицы символов

В большинстве компиляторов используется одна или несколько хеш-таблиц символов, которые служат центральным репозиторием сведений о различ-

ных именах, встречающихся в исходном коде, в ПП и в сгенерированном коде. По ходу разработки компилятора множество полей в таблице символов только растет. Добавляются поля для поддержки новых проходов и для передачи информации между проходами. Когда необходимость в поле отпадает, его можно удалить из определения таблицы символов или оставить там. По мере добавления полей размер таблицы символов увеличивается, и те части компилятора, которые имели к ней прямой доступ, необходимо перекомпилировать.

Мы столкнулись с этой проблемой при реализации сред программирования $\mathcal{R}^n$ и ParaScope. Вследствие экспериментальной природы этих систем поля добавлялись и удалялись очень часто. Чтобы решить эту проблему, мы спроектировали и реализовали более сложную, но и более гибкую таблицу символов – *двумерную хеш-таблицу*. Она позволила устранить большинство изменений в определении таблицы символов и ее реализации.

На рис. B.6 приведено концептуальное изображение такой таблицы, заполненной для блока на языке C, показанного во врезке. В реализации используются две разные индексированные хеш-таблицы: одна для первичного ключа, другая – для имени поля, являющегося вторичным ключом. Вторичный ключ используется, чтобы найти вектор данных для нужного поля. Первичный ключ нужен для выбора правильной позиции внутри вектора. Конечно, рисунок скрывает часть сложности. Первичный и вторичный индексы должны быть реализованы как полноценные хеш-таблицы, в которых коллизии разрешаются одним из двух описанных выше способов.

```
{ char w[12];
  real x;
  double y;
  int z;
  ...
}
```

Объявления для рис. B.6

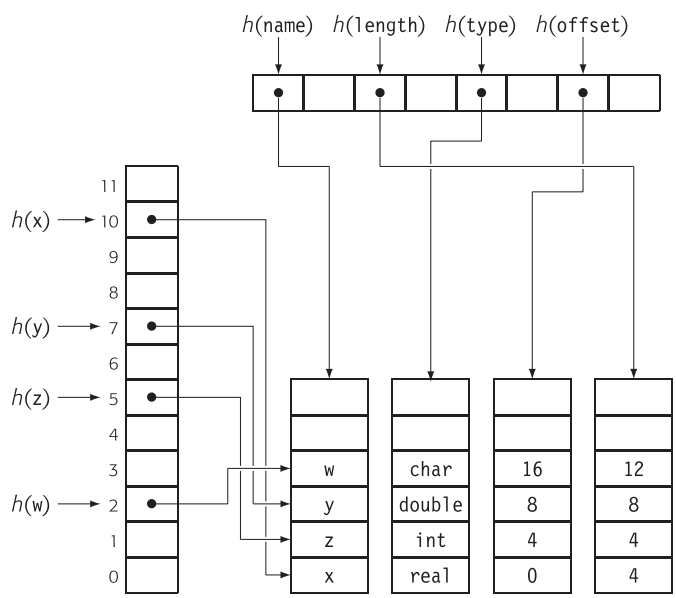


Рис. B.6 Двумерная хеш-таблица символов

Когда состав полей стабилизируется, автор компилятора может перейти на таблицу с фиксированными полями.

Несмотря на кажущуюся сложность, эта схема обходится не слишком дорого. Каждый доступ к таблице требует двух вычислений хеш-кода вместо одного. Реализации не нужно выделять память для поля, пока в нем не сохраняется какое-то значение, и это позволяет избежать накладных расходов на неиспользуемые поля. Каждый разработчик может создавать и удалять поля в таблице символов, не мешая другим программистам.

Реализация предоставляла точки входа для задания начальных значений поля (по имени), для удаления поля (по имени) и для формирования статистического отчета об использовании поля. Она позволяла программистам управлять собственной таблицей символов ответственно и независимо, не мешая коллегам.

И наконец, реализация была абстрагирована от конкретных экземпляров таблицы символов. То есть одна и та же реализация использовалась для управления таблицами в парсере и других частях компилятора.

Замечания

Многие алгоритмы компилятора предназначены для работы с множествами, отображениями, таблицами и графами. От реализации напрямую зависит пространственная и временная сложность этих алгоритмов и, в конечном счете, удобство пользования самим компилятором [63]. В учебниках рассматриваются многие вопросы, собранные в этом приложении [5, 45, 119, 207, 240].

В наших исследовательских компиляторах использовались почти все структуры данных, описанные в этом приложении. Несколько раз мы сталкивались с проблемами производительности, вызванными ростом структуры данных.

- Абстрактные синтаксические деревья, упомянутые во врезке в главе 4, могут вырастать до неприемлемых размеров. Техника отображения произвольного дерева в двоичное упрощает реализацию и, похоже, сдерживает накладные расходы [240].
- Табличное представление графа с помощью списков преемников и предшественников многократно изобреталось заново. Особенно хорошо оно работает для ГПУ, поскольку компилятору приходится много раз обходить как преемников, так и предшественников. Впервые мы воспользовались этой структурой данных в системе PFC в 1980 году.
- Множества, используемые в процессе анализа потока данных, бывают очень большими. Поскольку при таком масштабе выделение и освобождение памяти влияют на производительность, мы обычно использовали распределитель Хансона на основе арены [189] для битовых векторов.

- Размер и разреженность графов интерференции делают их еще одной областью, заслуживающей тщательного рассмотрения. Мы используем упорядоченный список с несколькими элементами множества в одном узле, чтобы удерживать стоимость построения графа в разумных пределах и одновременно управлять накладными расходами на выделение памяти [111].

Таблицы символов играют центральную роль в том, как компиляторы хранят информацию и обращаются к ней. Много внимания было уделено организации этих таблиц. Реорганизация списков [310, 329], сбалансированные деревья поиска [45, 119] и хеширование [240, том 3] важны для эффективного доступа к этим таблицам. Книги Кнута [240, том 3] и Кормена [119] – стандартные справочники по высококачественным хеш-функциям.

Литература

[1] Philip S. Abrams, An APL Machine, PhD thesis, Stanford University, Stanford, CA, February 1970, Technical Report SLAC-R-114, Stanford Linear Accelerator Center, Stanford University, February 1970.

[2] Aravind Acharya, Uday Bondhugula, Albert Cohen, Polyhedral autotransformation with no integer linear programming, in: Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation, ACM SIGPLAN Notices 53 (4) (June 2018) 529–542.

[3] Alfred V. Aho, Mahadevan Ganapathi, Steven W.K. Tjiang, Code generation using tree matching and dynamic programming, ACM Transactions on Programming Languages and Systems (TOPLAS) 11 (4) (October 1989) 491–516.

[4] Alfred V. Aho, John E. Hopcroft, Jeffrey D. Ullman, On finding lowest common ancestors in trees, in: Conference Record of the Fifth Annual ACM Symposium on Theory of Computing (STOC '73), Austin, TX, May 1973, pp. 253–265.

[5] Alfred V. Aho, John E. Hopcroft, Jeffrey D. Ullman, The Design and Analysis of Computer Algorithms, Addison-Wesley, Reading, MA, 1974.

[6] Alfred V. Aho, Stephen C. Johnson, Optimal code generation for expression trees, Journal of the ACM 23 (3) (July 1976) 488–501.

[7] Alfred V. Aho, Stephen C. Johnson, Jeffrey D. Ullman, Code generation for expressions with common subexpressions, in: Conference Record of the Third ACM Symposium on Principles of Programming Languages, Atlanta, GA, January 1976, pp. 19–31.

[8] Alfred V. Aho, Ravi Sethi, Jeffrey D. Ullman, Compilers: Principles, Techniques, and Tools, Addison-Wesley, Reading, MA, 1986[1].

[9] Alfred V. Aho, Jeffrey D. Ullman, The Theory of Parsing, Translation, and Compiling, Prentice-Hall, Englewood Cliffs, NJ, 1973.

[10] Philippe Aigrain, Susan L. Graham, Robert R. Henry, Marshall Kirk McKusick, Eduardo Pelegrí-Llopart, Experience with a Graham-Glanville style code generator, in: Proceedings of the ACM SIGPLAN '84 Symposium on Compiler Construction, ACM SIGPLAN Notices 19 (6) (June 1984) 13–24.

[11] Alexander Aiken, Alexandru Nicolau, Optimal loop parallelization, in: Proceedings of the ACM SIGPLAN '88 Conference on Programming Language Design and Implementation, ACM SIGPLAN Notices 23 (7) (July 1988) 308–317.

[1] *Альфред В. Ахо, Моника С. Лам, Рави Сети, Джеффри Д. Ульман*. Компиляторы. Принципы, технологии и инструментарий. Диалектика-Вильямс, 2017.

[12] Frances E. Allen, Program optimization, in: Mark I. Halprin, Christopher J. Shaw (Eds.), Annual Review in Automatic Programming, vol. 5, Pergamon Press, Oxford, England, 1969, pp. 239–307.
[13] Frances E. Allen, Control flow analysis, in: Proceedings of a Symposium on Compiler Optimization, ACM SIGPLAN Notices 5 (7) (July 1970) 1–19.
[14] Frances E. Allen, A basis for program optimization, in: Foundations and Systems, in: Information Processing: Proceedings of IFIP Congress 1971, vol. 1, Ljubljana, Yugoslavia, August 1971, North-Holland Publishing Company, Amsterdam, 1972, pp. 385–390.
[15] Frances E. Allen, The history of language processor technology in IBM, IBM Journal of Research and Development 25 (5) (September 1981) 535–548.
[16] Frances E. Allen, Private communication. Dr. Allen noted that Beatty described live analysis in a September 1968 document titled “Optimization Methods for Highly Parallel, Multiregister Machines,” April 2009.
[17] Frances E. Allen, John Cocke, A catalogue of optimizing transformations, in: R. Rustin (Ed.), Design and Optimization of Compilers, Prentice-Hall, Englewood Cliffs, NJ, June 1972, pp. 1–30.
[18] Frances E. Allen, John Cocke, Graph-theoretic constructs for program flow analysis, Technical Report RC 3923 (17789), IBM Thomas J. Watson Research Center, Yorktown Heights, NY, July 1972.
[19] Frances E. Allen, John Cocke, A program data flow analysis procedure, Communications of the ACM 19 (3) (March 1976) 137–147.
[20] Frances E. Allen, John Cocke, Ken Kennedy, Reduction of operator strength, in: Steven S. Muchnick, Neil D. Jones (Eds.), Program Flow Analysis: Theory and Applications, Prentice-Hall, Englewood Cliffs, NJ, 1981, chapter 3, pp. 79–101.
[21] John R. Allen, Ken Kennedy, Optimizing Compilers for Modern Architectures, Morgan Kaufmann, San Francisco, CA, October 2001.
[22] Bowen Alpern, Fred B. Schneider, Verifying temporal properties without temporal logic, ACM Transactions on Programming Languages and Systems (TOPLAS) 11 (1) (January 1989) 147–167.
[23] Bowen Alpern, Mark N. Wegman, F. Kenneth Zadeck, Detecting equality of variables in programs, in: Proceedings of the Fifteenth Annual ACM Symposium on Principles of Programming Languages, San Diego, CA, January 1988, pp. 1–11.
[24] Stephen Alstrup, Dov Harel, Peter W. Lauridsen, Mikkel Thorup, Dominators in linear time, SIAM Journal on Computing 28 (6) (June 1999) 2117–2132.
[25] Marc A. Auslander, Martin E. Hopkins, An overview of the PL.8 compiler, in: Proceedings of the ACM SIGPLAN ’82 Symposium on Compiler Construction, ACM SIGPLAN Notices 17 (6) (June 1982) 22–31.
[26] Andrew Ayers, Robert Gottlieb, Richard Schooler, Aggressive inlining, in: Proceedings of the ACM SIGPLAN ’97 Conference on Programming Language Design and Implementation, ACM SIGPLAN Notices 32 (5) (May 1997) 134–145.

[27] John W. Backus, The history of FORTRAN I, II, and III, in: Richard L. Wexelblat (Ed.), History of Programming Languages, Academic Press, New York, NY, 1981, pp. 25–45.
[28] John W. Backus, R. J. Beeber, S. Best, R. Goldberg, L. M. Haibt, H. L. Herrick, R. A. Nelson, D. Sayre, P. B. Sheridan, H. Stern, I. Ziller, R. A. Hughes, R. Nutt, The FORTRAN automatic coding system, in: Proceedings of the Western Joint Computer Conference, Institute of Radio Engineers, New York, NY, February 1957, pp. 188–198.
[29] David F. Bacon, Susan L. Graham, Oliver J. Sharp, Compiler transformations for high-performance computing, ACM Computing Surveys 26 (4) (1994) 345–420.
[30] Jean-Loup Baer, D. P. Bovet, Compilation of arithmetic expressions for parallel computations, in: Mathematics, Software, Volume 1 of Information Processing 68: Proceedings of IFIP Congress 1968, Edinburgh, UK, August 1968, North-Holland Publishing Company, Amsterdam, 1968, pp. 340–346.
[31] John T. Bagwell Jr., Local optimizations, in: Proceedings of a Symposium on Compiler Optimization, ACM SIGPLAN Notices 5 (7) (July 1970) 52–66.
[32] Vasanth Bala, Evelyn Duesterwald, Sanjeev Banerjia, Dynamo: A transparent dynamic optimization system, in: Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation, ACM SIGPLAN Notices 35 (5) (May 2000) 1–12.
[33] Vasanth Bala, Norman Rubin, Efficient instruction scheduling using finite state automata, International Journal of Parallel Programming 25 (2) (April 1997) 53–82.
[34] J. Eugene Ball, Predicting the effects of optimization on a procedure body, in: Proceedings of the ACM SIGPLAN '79 Symposium on Compiler Construction, ACM SIGPLAN Notices 14 (8) (August 1979) 214–220.
[35] John Banning, An efficient way to find side effects of procedure calls and aliases of variables, in: Conference Record of the Sixth Annual ACM Symposium on Principles of Programming Languages, San Antonio, TX, January 1979, pp. 29–41.
[36] William A. Barrett, John D. Couch, Compiler Construction: Theory and Practice, Science Research Associates, Inc., Chicago, IL, 1979.
[37] Jeffrey M. Barth, An interprocedural data flow analysis algorithm, in: Conference Record of the Fourth ACM Symposium on Principles of Programming Languages, Los Angeles, CA, January 1977, pp. 119–131.
[38] Alan M. Bauer, Harry J. Saal, Does APL really need run-time checking?, Software–Practice and Experience 4 (2) (1974) 129–138.
[39] Laszlo A. Belady, A study of replacement algorithms for a virtual storage computer, IBM Systems Journal 5 (2) (1966) 78–101.
[40] Peter Bergner, Peter Dahl, David Engebretsen, Matthew T. O'Keefe, Spill code minimization via interference region spilling, in: Proceedings of the ACM SIGPLAN '97 Conference on Programming Language Design and Implementation, ACM SIGPLAN Notices 32 (5) (May 1997) 287–295.

[41] David Bernstein, Dina Q. Goldin, Martin Charles Golumbic, Hugo Krawczyk, Yishay Mansour, Itai Nahshon, Ron Y. Pinter, Spill code minimization techniques for optimizing compilers, in: Proceedings of the ACM SIGPLAN ’89 Conference on Programming Language Design and Implementation, ACM SIGPLAN Notices 24 (7) (July 1989) 258–263.

[42] David Bernstein, Michael Rodeh, Global instruction scheduling for superscalar machines, in: Proceedings of the ACM SIGPLAN ’91 Conference on Programming Language Design and Implementation, ACMSIGPLAN Notices 26 (6) (June 1991) 241–255.

[43] Robert L. Bernstein, Producing good code for the case statement, Software–Practice and Experience 15 (10) (October 1985) 1021–1024.

[44] David A. Berson, Rajiv Gupta, Mary Lou Soffa, Integrated instruction scheduling and register allocation techniques, in: Proceedings of the 11th International Workshop on Languages and Compilers for Parallel Computing, in: Lecture Notes in Computer Science (LNCS), vol. 1656, Springer-Verlag, Berlin-Heidelberg, Germany, 1998, pp. 247–262.

[45] Andrew Binstock, John Rex, Practical Algorithms for Programmers, Addison-Wesley, Reading, MA, 1995.

[46] Peter L. Bird, An implementation of a code generator specification language for table driven code generators, in: Proceedings of the ACM SIGPLAN ’82 Symposium on Compiler Construction, ACM SIGPLAN Notices 17 (6) (June 1982) 44–55.

[47] Rastislav Bodík, Rajiv Gupta, Mary Lou Soffa, Complete removal of redundant expressions, in: Proceedings of the ACM SIGPLAN ’98 Conference on Programming Language Design and Implementation, ACM SIGPLAN Notices 33 (5) (May 1998) 1–14.

[48] Hans-Juergen Boehm, Space efficient conservative garbage collection, in: Proceedings of the ACMSIGPLAN ’93 Conference on Programming Language Design and Implementation, ACM SIGPLAN Notices 28 (6) (June 1993) 197–206.

[49] Hans-Juergen Boehm, Alan Demers, Implementing Russell, in: Proceedings of the ACM SIGPLAN ’86 Symposium on Compiler Construction, ACM SIGPLAN Notices 21 (7) (July 1986) 186–195.

[50] Hans-Juergen Boehm, Mark Weiser, Garbage collection in an uncooperative environment, Software–Practice and Experience 18 (9) (September 1988) 807–820.

[51] Benoit Boissinot, Alain Darte, Fabrice Rastello, Benoit Dupont de Dinechin, Christophe Guillon, Revisiting out-of-SSA translation for correctness, code quality and efficiency, in: Proceedings of the 7th Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO ’09), Seattle, WA, March 2009, pp. 114–125.

[52] Uday Bondhugula, Albert Hartono, J. Ramanujam, P. Sadayappan, A practical automatic polyhedral parallelizer and locality optimizer, in: Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation, ACM SIGPLAN Notices 43 (6) (June 2008) 101–113.

[53] Florent Bouchez, A Study of Spilling and Coalescing in Register Allocation as Two Separate Phases, PhD thesis, École Normale Supérieur de Lyon, Lyon, France, April 2009.

[54] David G. Bradlee, Susan J. Eggers, Robert R. Henry, Integrating register allocation and instruction scheduling for RISCs, in: Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-IV), ACM SIGPLAN Notices 26 (4) (April 1991) 122–131.

[55] Preston Briggs, Register Allocation via Graph Coloring, PhD thesis, Department of Computer Science, Rice University, Houston, TX, April 1992, Technical Report TR92-183, Computer Science Department, Rice University, 1992.

[56] Preston Briggs, Keith D. Cooper, Timothy J. Harvey, L. Taylor Simpson, Practical improvements to the construction and destruction of static single assignment form, Software–Practice and Experience 28 (8) (July 1998) 859–881.

[57] Preston Briggs, Keith D. Cooper, Ken Kennedy, Linda Torczon, Coloring heuristics for register allocation, in: Proceedings of the ACM SIGPLAN '89 Conference on Programming Language Design and Implementation, ACM SIGPLAN Notices 24 (7) (July 1989) 275–284.

[58] Preston Briggs, Keith D. Cooper, Ken Kennedy, Linda Torczon, Digital computer register allocation and code spilling using interference graph coloring, United States Patent 5, 249, 295, March 1993.

[59] Preston Briggs, Keith D. Cooper, L. Taylor Simpson, Value numbering, Software–Practice and Experience 27 (6) (June 1997) 701–724.

[60] Preston Briggs, Keith D. Cooper, Linda Torczon, Coloring register pairs, ACM Letters on Programming Languages and Systems (LOPLAS) 1 (1) (March 1992) 3–13.

[61] Preston Briggs, Keith D. Cooper, Linda Torczon, Rematerialization, in: Proceedings of the ACM SIGPLAN '92 Conference on Programming Language Design and Implementation, ACM SIGPLAN Notices 27 (7) (July 1992) 311–321.

[62] Preston Briggs, Keith D. Cooper, Linda Torczon, Improvements to graph coloring register allocation, ACM Transactions on Programming Languages and Systems (TOPLAS) 16 (3) (May 1994) 428–455.

[63] Preston Briggs, Linda Torczon, An efficient representation for sparse sets, ACM Letters on Programming Languages and Systems (LOPLAS) 2 (1–4) (March–December 1993) 59–69.

[64] Philip Brisk, Foad Dabiri, Jamie Macbeth, Majid Sarrafzadeh, Polynomial time graph coloring register allocation, in: Fourteenth International Workshop on Logic and Synthesis, Lake Arrowhead, CA, June 2005, pp. 447–454.

[65] Klaus Brouwer, Wolfgang Gellerich, Erhard Ploedereder, Myths and facts about the efficient implementation of finite automata and lexical analysis, in: Proceedings of the Seventh International Conference on Compiler Construction (CC '98), in: Lecture Notes in Computer Science (LNCS), vol. 1383, Springer-Verlag, Heidelberg, Germany, 1998, pp. 1–15.

[66] Janusz A. Brzozowski, Canonical regular expressions and minimal state graphs for definite events, in: Mathematical Theory of Automata, in: MRI Symposia Series, vol. 12, Polytechnic Press, Polytechnic Institute of Brooklyn, New York, NY, 1962, pp. 529–561.

[67] Adam L. Buchsbaum, Haim Kaplan, Anne Rogers, Jeffery R. Westbrook, Lineartime pointer-machine algorithms for least common ancestors, MST verification, and dominators, in: Proceedings of the Thirtieth Annual ACM Symposium on Theory of Computing (STOC), Dallas, TX, 1998, pp. 279–288.

[68] Michael Burke, An interval-based approach to exhaustive and incremental interprocedural data-flow analysis, ACM Transactions on Programming Languages and Systems (TOPLAS) 12 (3) (July 1990) 341–395.

[69] Michael Burke, Jong-Deok Choi, Stephen Fink, David Grove, Michael Hind, Vivek Sarkar, Mauricio Serrano, V.C. Sreedhar, Harini Srinivasan, The Jalapeño dynamic optimizing compiler for Java, in: Proceedings of the ACM 1999 Conference on Java Grande, San Francisco, CA, June 1999, pp. 129–141.

[70] Michael Burke, Linda Torczon, Interprocedural optimization: Eliminating unnecessary recompilation, ACMTransactions on Programming Languages and Systems (TOPLAS) 15 (3) (July 1993) 367–399.

[71] Jiazhen Cai, Robert Paige, Using multiset discrimination to solve language processing problems without hashing, Theoretical Computer Science 145 (1–2) (1995) 189–228.

[72] Brad Calder, Dirk Grunwald, Reducing branch costs via branch alignment, in: Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VI), ACM SIGPLAN Notices 29 (11) (November 1994) 242–251.

[73] David Callahan, Steve Carr, Ken Kennedy, Improving register allocation for subscripted variables, in: Proceedings of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation, ACM SIGPLAN Notices 25 (6) (June 1990) 53–65.

[74] David Callahan, Keith D. Cooper, Ken Kennedy, Linda Torczon, Interprocedural constant propagation, in: Proceedings of the ACM SIGPLAN '86 Symposium on Compiler Construction, ACM SIGPLAN Notices 21 (7) (July 1986) 152–161.

[75] David Callahan, Brian Koblenz, Register allocation via hierarchical graph coloring, in: Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation, ACMSIGPLAN Notices 26 (6) (June 1991) 192–203.

[76] Luca Cardelli, Type systems, in: Allen B. Tucker Jr. (Ed.), The Computer Science and Engineering Handbook, CRC Press, Boca Raton, FL, December 1996, chapter 103, pp. 2208–2236.

[77] Steve Carr, Ken Kennedy, Scalar replacement in the presence of conditional control flow, Software–Practice and Experience 24 (1) (January 1994) 51–77.

[78] Roderic G. G. Cattell, Automatic derivation of code generators from machine descriptions, ACM Transactions on Programming Languages and Systems (TOPLAS) 2 (2) (April 1980) 173–190.

[79] Roderic G. G. Cattell, Joseph M. Newcomer, Bruce W. Leverett, Code generation in a machine-independent compiler, in: Proceedings of the ACM SIGPLAN '79 Symposium on Compiler Construction, ACM SIGPLAN Notices 14 (8) (August 1979) 65–75.

[80] Gregory J. Chaitin, Register allocation and spilling via graph coloring, in: Proceedings of the ACM SIGPLAN '82 Symposium on Compiler Construction, ACM SIGPLAN Notices 17 (6) (June 1982) 98–105.

[81] Gregory J. Chaitin, Register allocation and spilling via graph coloring, United States Patent 4, 571, 678, February 1986.

[82] Gregory J. Chaitin, Marc A. Auslander, Ashok K. Chandra, John Cocke, Martin E. Hopkins, Peter W. Markstein, Register allocation via coloring, Computer Languages 6 (1) (January 1981) 47–57.

[83] David R. Chase, An improvement to bottom-up tree pattern matching, in: Proceedings of the Fourteenth Annual ACM Symposium on Principles of Programming Languages, Munich, Germany, January 1987, pp. 168–177.

[84] David R. Chase, MarkWegman, F. Kenneth Zadeck, Analysis of pointers and structures, in: Proceedings of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation, ACMSIGPLAN Notices 25 (6) (June 1990) 296–310.

[85] J. Bradley Chen, Bradley D. D. Leupen, Improving instruction locality with just-intime code layout, in: Proceedings of the First USENIX Windows NT Workshop, Seattle, WA, August 1997, pp. 25–32.

[86] C. J. Cheney, A nonrecursive list compacting algorithm, Communications of the ACM 13 (11) (November 1970) 677–678.

[87] Jong-Deok Choi, Michael Burke, Paul R. Carini, Efficient flow-sensitive interprocedural computation of pointer-induced aliases and side effects, in: Proceedings of the Twentieth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Charleston, SC, January 1993, pp. 232–245.

[88] Frederick C. Chow, A Portable Machine-Independent Global Optimizer–Designand Measurements, PhD thesis, Department of Electrical Engineering, Stanford University, December 1983, Technical Report CSL-TR-83-254, Computer Systems Laboratory, Stanford University, Palo Alto, CA, USA, December 1983.

[89] Cliff Click, Combining Analyses, Combining Optimizations, PhD thesis, Department of Computer Science, Rice University, Houston, TX, February 1995, Technical Report TR95-252, Computer Science Department, Rice University, 1995.

[90] Cliff Click, Global code motion/global value numbering, in: Proceedings of the ACM SIGPLAN '95 Conference on Programming Language Design and Implementation, ACM SIGPLAN Notices 30 (6) (June 1995) 246–257.

[91] Cliff Click, Keith D. Cooper, Combining analyses, combining optimizations, ACM Transactions on Programming Languages and Systems (TOPLAS) 17 (2) (March 1995) 181–196.

[92] Cliff Click, Michael Paleczny, A simple graph-based intermediate representation, in: Papers from the 1995 ACM SIGPLAN Workshop on Intermediate Representations, ACM SIGPLAN Notices 30 (3) (March 1995) 35–49.
[93] Cliff Click, Michael Paleczny, Christopher Vick, Interference graph trimming (unpublished), Technical report available on ResearchGate, May 2004.
[94] John Cocke, Global common subexpression elimination, in: Proceedings of a Symposium on Compiler Optimization, ACM SIGPLAN Notices 5 (7) (July 1970) 20–24.
[95] John Cocke, Ken Kennedy, An algorithm for reduction of operator strength, Communications of the ACM 20 (11) (November 1977) 850–856.
[96] John Cocke, Peter W. Markstein, Measurement of program improvement algorithms, in: Simon H. Lavington (Ed.), Information Processing 80, Proceedings of IFIP Congress 80, North Holland, Amsterdam, Netherlands, 1980, pp. 221–228.
[97] John Cocke, Peter W. Markstein, Strength reduction for division and modulo with application to accessing a multilevel store, IBM Journal of Research and Development 24 (6) (1980) 692–694.
[98] John Cocke, Jacob T. Schwartz, Programming languages and their compilers: Preliminary notes, Technical Report, Courant Institute of Mathematical Sciences, New York University, New York, NY, 1970.
[99] Jacques Cohen, Garbage collection of linked structures, ACM Computing Surveys 13 (3) (September 1981) 341–367.
[100] Robert Cohn, P. Geoffrey Lowney, Hot cold optimization of largeWindows/NT applications, in: Proceedings of the Twenty-Ninth IEEE/ACM Annual International Symposium on Microarchitecture (MICRO-29), Paris, France, December 1996, pp. 80–89.
[101] Stephanie Coleman, Kathryn S. McKinley, Tile size selection using cache organization and data layout, in: Proceedings of the ACM SIGPLAN '95 Conference on Programming Language Design and Implementation, ACM SIGPLAN Notices 30 (6) (June 1995) 279–290.
[102] George E. Collins, A method for overlapping and erasure of lists, Communications of the ACM 3 (12) (December 1960) 655–657.
[103] Melvin E. Conway, Design of a separable transition diagram compiler, Communications of the ACM 6 (7) (July 1963) 396–408.
[104] Richard W. Conway, Thomas R. Wilcox, Design and implementation of a diagnostic compiler for PL/I, Communications of the ACM 16 (3) (March 1973) 169–179.
[105] Katherine Coons, Warren Hunt, Bertrand A. Maher, Doug Burger, Kathryn S. McKinley, Optimal Huffman tree-height reduction for instruction-level parallelism, Technical Report TR-08-34, Department of Computer Science, The University of Texas at Austin, Austin, TX, August 2008.
[106] Keith D. Cooper, Anshuman Dasgupta, Jason Eckhardt, Revisiting graph coloring register allocation: A study of the Chaitin-Briggs and Callahan-Koblenz algorithms, in: Proceedings of the Eighteenth International Con-

ference on Languages and Compilers for Parallel Computing (LCPC '05), in: Lecture Notes in Computer Science (LNCS), vol. 4339, Springer-Verlag, Berlin, Heidelberg, Germany, 2006, pp. 1–16.

[107] Keith D. Cooper, Jason Eckhardt, Improved passive splitting, in: Proceedings of the 2005 International Conference on Programming Languages and Compilers, CSREA Press, Las Vegas, NV, June 2005, pp. 115–122.

[108] Keith D. Cooper, Mary W. Hall, Linda Torczon, An experiment with inline substitution, Software–Practice and Experience 21 (6) (June 1991) 581–601.

[109] Keith D. Cooper, Timothy J. Harvey, Ken Kennedy, An empirical study of iterative data-flow analysis, in: Proceedings of the Fifteenth International Conference on Computing (CIC'06), IEEE Computer Society, Washington, D. C., 2006, pp. 266–276.

[110] Keith D. Cooper, Timothy J. Harvey, Ken Kennedy, A simple, fast dominance algorithm, Technical Report TR-06-38870, Department of Computer Science, Rice University, Houston, TX, January 2006.

[111] Keith D. Cooper, Timothy J. Harvey, Linda Torczon, How to build an interference graph, Software–Practice and Experience 28 (4) (April 1998) 425–444.

[112] Keith D. Cooper, Ken Kennedy, Interprocedural side-effect analysis in linear time, in: Proceedings of the ACM SIGPLAN '88 Conference on Programming Language Design and Implementation, ACM SIGPLAN Notices 23 (7) (July 1988) 57–66.

[113] Keith D. Cooper, Ken Kennedy, Fast interprocedural alias analysis, in: Proceedings of the Sixteenth Annual ACM Symposium on Principles of Programming Languages, Austin, TX, January 1989, pp. 49–59.

[114] Keith D. Cooper, Ken Kennedy, Linda Torczon, The impact of interprocedural analysis and optimization in the $\mathcal{R}^n$ programming environment, ACM Transactions on Programming Languages and Systems (TOPLAS) 8 (4) (October 1986) 491–523.

[115] Keith D. Cooper, Philip J. Schielke, Non-local instruction scheduling with limited code growth, in: F. Mueller, A. Bestavros (Eds.), Proceedings of the 1998 ACM SIGPLAN Workshop on Languages, Compilers, and Tools for Embedded Systems (LCTES), in: Lecture Notes in Computer Science (LNCS), vol. 1474, Springer-Verlag, Heidelberg, Germany, 1998, pp. 193–207.

[116] Keith D. Cooper, L. Taylor Simpson, Live range splitting in a graph coloring register allocator, in: Proceedings of the Seventh International Compiler Construction Conference (CC '98), in: Lecture Notes in Computer Science (LNCS), vol. 1383, Springer-Verlag, Heidelberg, Germany, 1998, pp. 174–187.

[117] Keith D. Cooper, L. Taylor Simpson, Christopher A. Vick, Operator strength reduction, ACM Transactions on Programming Languages and Systems (TOPLAS) 23 (5) (September 2001) 603–625.

[118] Keith D. Cooper, Todd Waterman, Understanding energy consumption on the C62x, in: Proceedings of the 2002Workshop on Compilers and Operating Systems for Low Power, Charlottesville, VA, September 2002, pp. 4-1–4-8.

[119] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Introduction to Algorithms, MIT Press, Cambridge, MA, 1992[1].

[120] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, F. Kenneth Zadeck, Efficiently computing static single assignment form and the control dependence graph, ACM Transactions on Programming Languages and Systems (TOPLAS) 13 (4) (October 1991) 451–490.

[121] Ron Cytron, Andy Lowry, F. Kenneth Zadeck, Code motion of control structures in high-level languages, in: Conference Record of the Thirteenth Annual ACM Symposium on Principles of Programming Languages, St. Petersburg Beach, FL, January 1986, pp. 70–85.

[122] Jan Daciuk, Comparison of construction algorithms for minimal, acyclic, deterministic, finite-state automata from sets of strings, in: Proceedings of the Seventh International Conference on Implementation and Application of Automata (CIAA 2002), in: Lecture Notes in Computer Science (LNCS), vol. 2608, Springer, Berlin, Heidelberg, 2003, pp. 255–261.

[123] Manuvir Das, Unification-based pointer analysis with directional assignments, in: Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation, ACM SIGPLAN Notices 35 (5) (May 2000) 35–46.

[124] Anshuman Dasgupta, Tailoring Traditional Optimizations for Runtime Compilation, PhD thesis, Department of Computer Science, Rice University, Houston, TX, September 2006.

[125] Jack W. Davidson, Christopher W. Fraser, The design and application of a retargetable peephole optimizer, ACM Transactions on Programming Languages and Systems (TOPLAS) 2 (2) (April 1980) 191–202.

[126] Jack W. Davidson, Christopher W. Fraser, Automatic generation of peephole optimizations, in: Proceedings of the ACM SIGPLAN '84 Symposium on Compiler Construction, ACM SIGPLAN Notices 19 (6) (June 1984) 111–116.

[127] Jack W. Davidson, Christopher W. Fraser, Register allocation and exhaustive peephole optimization, Software–Practice and Experience 14 (9) (September 1984) 857–865.

[128] Jack W. Davidson, Christopher W. Fraser, Automatic inference and fast interpretation of peephole optimization rules, Software–Practice and Experience 17 (11) (November 1987) 801–812.

[129] Jack W. Davidson, Ann M. Holler, A study of a C function inliner, Software–Practice and Experience 18 (8) (August 1988) 775–790.

[130] Jack W. Davidson, Sanjay Jinturkar, Aggressive loop unrolling in a retargetable, optimizing compiler, in: Proceedings of the Sixth International Conference on Compiler Construction (CC '96), in: Lecture Notes in Computer Science (LNCS), vol. 1060, Springer, Berlin, Heidelberg, April 1996, pp. 59–73.

---

[1] *Кормен Томас Х., Лейзерсон Чарльз И., Ривест Рональд Л.* Алгоритмы: построение и анализ. МЦНМО, 2002.

[131] Alan J. Demers, Mark Weiser, Barry Hayes, Hans Boehm, Daniel Bobrow, Scott Shenker, Combining generational and conservative garbage collection: Framework and implementations, in: Proceedings of the Seventeenth Annual ACM Symposium on Principles of Programming Languages, San Francisco, CA, January 1990, pp. 261–269.

[132] Frank DeRemer, Simple LR(k) grammars, Communications of the ACM 14 (7) (July 1971) 453–460.

[133] Frank DeRemer, Thomas J. Pennello, Efficient computation of LALR(1) lookahead sets, in: Proceedings of the ACM SIGPLAN '79 Symposium on Compiler Construction, ACM SIGPLAN Notices 14 (8) (August 1979) 176–187.

[134] Alain Deutsch, Interprocedural may-alias analysis for pointers: Beyond *k*-limiting, in: Proceedings of the ACM SIGPLAN '94 Conference on Programming Language Design and Implementation, ACMSIGPLAN Notices 29 (6) (June 1994) 230–241.

[135] L. Peter Deutsch, An Interactive Program Verifier, PhD thesis, Computer Science Department, University of California, Berkeley, Berkeley, CA, 1973, Technical Report CSL-73-1, Xerox Palo Alto Research, May 1973.

[136] L. Peter Deutsch, Daniel G. Bobrow, An efficient, incremental, automatic, garbage collector, Communications of the ACM 19 (9) (September 1976) 522–526.

[137] L. Peter Deutsch, Allan M. Schiffman, Efficient implementation of the Smalltalk-80 system, in: Conference Record of the Eleventh Annual ACM Symposium on Principles of Programming Languages, Salt Lake City, UT, January 1984, pp. 297–302.

[138] Dhananjay M. Dhamdhere, On algorithms for operator strength reduction, Communications of the ACM 22 (5) (May 1979) 311–312.

[139] Dhananjay M. Dhamdhere, A fast algorithm for code movement optimisation, ACM SIGPLAN Notices 23 (10) (October 1988) 172–180.

[140] Dhananjay M. Dhamdhere, A new algorithm for composite hoisting and strength reduction, International Journal of Computer Mathematics 27 (1) (1989) 1–14.

[141] Dhananjay M. Dhamdhere, Practical adaptation of the global optimization algorithm of Morel and Renvoise, ACM Transactions on Programming Languages and Systems (TOPLAS) 13 (2) (April 1991) 291–294.

[142] Dhananjay M. Dhamdhere, J. R. Isaac, A composite algorithm for strength reduction and code movement optimization, International Journal of Computer and Information Sciences 9 (3) (June 1980) 243–273.

[143] Michael K. Donegan, Robert E. Noonan, Stefan Feyock, A code generator generator language, in: Proceedings of the ACM SIGPLAN '79 Symposium on Compiler Construction, ACM SIGPLAN Notices 14 (8) (August 1979) 58–64.

[144] Karl-Heinz Drechsler, Manfred P. Stadel, A solution to a problem with Morel and Renvoise's "Global optimization by suppression of partial redundancies," ACM Transactions on Programming Languages and Systems (TOPLAS) 10 (4) (October 1988) 635–640.

[145] Karl-Heinz Drechsler, Manfred P. Stadel, A variation of Knoop, Rüthing, and Steffen's "Lazy code motion," ACM SIGPLAN Notices 28 (5) (May 1993) 29–38.
[146] Jay Earley, An efficient context-free parsing algorithm, Communications of the ACM 13 (2) (February 1970) 94–102.
[147] Kemal Ebcioğlu, Toshio Nakatani, A new compilation technique for parallelizing loops with unpredictable branches on a VLIW architecture, in: Selected Papers of the SecondWorkshop on Languages and Compilers for Parallel Computing (LCPC '89), Pitman Publishing, London, UK, 1990, pp. 213–229.
[148] John R. Ellis, Bulldog: A Compiler for VLIW Architectures, The MIT Press, Cambridge, MA, 1986.
[149] Maryam Emami, Rakesh Ghiya, Laurie J. Hendren, Context-sensitive interprocedural points-to analysis in the presence of function pointers, in: Proceedings of the ACM SIGPLAN '94 Conference on Programming Language Design and Implementation, ACM SIGPLAN Notices 29 (6) (June 1994) 242–256.
[150] Andrei P. Ershov, On programming of arithmetic expressions, Communications of the ACM 1 (8) (August 1958) 3–6 (The figures appear in volume 1, number 9, page 16).
[151] *Ершов А. П.* Сведение задачи распределения памяти при составлении программ к задаче раскраски вершин графов // Докл. АН СССР, 142:4 (1962), 785–787.
[152] Andrei P. Ershov, Alpha–An automatic programming system of high efficiency, Journal of the ACM 13 (1) (January 1966) 17–24.
[153] M. Anton Ertl, Optimal code selection in DAGs, in: Proceedings of the Twenty-Sixth ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Antonio, TX, January 1999, pp. 242–249.
[154] Robert R. Fenichel, Jerome C. Yochelson, A LISP garbage-collector for virtual memory computer systems, Communications of the ACM 12 (11) (November 1969) 611–612.
[155] Jeanne Ferrante, Karl J. Ottenstein, Joe D.Warren, The program dependence graph and its use in optimization, ACM Transactions on Programming Languages and Systems (TOPLAS) 9 (3) (July 1987) 319–349.
[156] Charles N. Fischer, Richard J. LeBlanc Jr., The implementation of run-time diagnostics in Pascal, IEEE Transactions on Software Engineering SE-6 (4) (1980) 313–319.
[157] Charles N. Fischer, Richard J. LeBlanc Jr., Crafting a Compiler with C, Benjamin/Cummings, Redwood City, CA, 1991.
[158] Joseph A. Fisher, Trace scheduling: A technique for global microcode compaction, IEEE Transactions on Computers C-30 (7) (July 1981) 478–490.
[159] Joseph A. Fisher, John R. Ellis, John C. Ruttenberg, Alexandru Nicolau, Parallel processing: A smart compiler and a dumb machine, in: Proceedings of the ACM SIGPLAN '84 Symposium on Compiler Construction, ACM SIGPLAN Notices 19 (6) (June 1984) 37–47.

[160] Robert W. Floyd, An algorithm for coding efficient arithmetic expressions, Communications of the ACM 4 (1) (January 1961) 42–51.
[161] J. M. Foster, A syntax improving program, Computer Journal 11 (1) (May 1968) 31–34.
[162] Christopher W. Fraser, David R. Hanson, Todd A. Proebsting, Engineering a simple, efficient code generator generator, ACM Letters on Programming Languages and Systems (LOPLAS) 1 (3) (September 1992) 213–226.
[163] Christopher W. Fraser, Robert R. Henry, Hard-coding bottom-up code generation tables to save time and space, Software–Practice and Experience 21 (1) (January 1991) 1–12.
[164] Christopher W. Fraser, Alan L. Wendt, Integrating code generation and optimization, in: Proceedings of the ACM SIGPLAN '86 Symposium on Compiler Construction, ACM SIGPLAN Notices 21 (7) (July 1986) 242–248.
[165] Christopher W. Fraser, Alan L. Wendt, Automatic generation of fast optimizing code generators, in: Proceedings of the ACM SIGPLAN '88 Conference on Programming Language Design and Implementation, ACM SIGPLAN Notices 23 (7) (July 1988) 79–84.
[166] Mahadevan Ganapathi, Charles N. Fischer, Description-driven code generation using attribute grammars, in: Conference Record of the Ninth Annual ACM Symposium on Principles of Programming Languages, Albuquerque, NM, January 1982, pp. 108–119.
[167] Karthik Gargi, A sparse algorithm for predicated global value numbering, in: Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation, ACM SIGPLAN Notices 37 (5) (May 2002) 45–56.
[168] George Lal, Andrew W. Appel, Iterated register coalescing, in: Proceedings of the Twenty-Third ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, St. Petersburg Beach, FL, January 1996, pp. 208–218.
[169] Phillip B. Gibbons, Steven S. Muchnick, Efficient instruction scheduling for a pipelined architecture, in: Proceedings of the ACM SIGPLAN '86 Symposium on Compiler Construction, ACM SIGPLAN Notices 21 (7) (July 1986) 11–16.
[170] R. Steven Glanville, Susan L. Graham, A new method for compiler code generation, in: Conference Record of the Fifth Annual ACM Symposium on Principles of Programming Languages, Tucson, AZ, January 1978, pp. 231–240.
[171] Nikolas Gloy, Michael D. Smith, Procedure placement using temporal-ordering information, ACM Transactions on Programming Languages and Systems (TOPLAS) 21 (5) (September 1999) 977–1027.
[172] Adele Goldberg, David Robson, Smalltalk-80: The Language and Its Implementation, Addison-Wesley, Reading, MA, 1983.
[173] James R. Goodman, Wei-Chung Hsu, Code scheduling and register allocation in large basic blocks, in: Proceedings of the Second International Conference on Supercomputing, Saint Malo, France, July 1988, pp. 442–452.

[174] Eiichi Goto, Monocopy and associative operations in extended Lisp, Technical Report 74-03, University of Tokyo, Tokyo, Japan, May 1974.
[175] Susan L. Graham, Table-driven code generation, IEEE Computer 13 (8) (August 1980) 25–34.
[176] Susan L. Graham, Michael A. Harrison, Walter L. Ruzzo, An improved contextfree recognizer, ACM Transactions on Programming Languages and Systems (TOPLAS) 2 (3) (July 1980) 415–462.
[177] Susan L. Graham, Robert R. Henry, Robert A. Schulman, An experiment in table driven code generation, in: Proceedings of the ACM SIGPLAN '82 Symposium on Compiler Construction, ACM SIGPLAN Notices 17 (6) (June 1982) 32–43.
[178] Susan L. Graham, Mark Wegman, A fast and usually linear algorithm for global flow analysis, in: Conference Record of the Second ACM Symposium on Principles of Programming Languages, Palo Alto, CA, January 1975, pp. 22–34.
[179] Susan L. Graham, Mark Wegman, A fast and usually linear algorithm for global flow analysis, Journal of the ACM 23 (1) (1976) 172–202.
[180] Torbjörn Granlund, Richard Kenner, Eliminating branches using a superoptimizer and the GNU C compiler, in: Proceedings of the ACM SIGPLAN '92 Conference on Programming Language Design and Implementation, ACM SIGPLAN Notices 27 (7) (July 1992) 341–352.
[181] David Gries, Compiler Construction for Digital Computers, John Wiley and Sons, New York, NY, 1971.
[182] Daniel Grove, Linda Torczon, Interprocedural constant propagation: A study of jump function implementations, in: Proceedings of the ACM SIGPLAN '93 Conference on Programming Language Design and Implementation, ACM SIGPLAN Notices 28 (6) (June 1993) 90–99.
[183] Rajiv Gupta, Mary Lou Soffa, Region scheduling: An approach for detecting and redistributing parallelism, IEEE Transactions on Software Engineering SE-16 (4) (April 1990) 421–431.
[184] Rajiv Gupta, Mary Lou Soffa, Tim Steele, Register allocation via clique separators, in: Proceedings of the ACM SIGPLAN '89 Conference on Programming Language Design and Implementation, ACM SIGPLAN Notices 24 (7) (July 1989) 264–274.
[185] Sebastian Hack, Register Allocation for Programs in SSA Form, PhD thesis, Universität Karlsruhe, Karlsruhe, Germany, October 2006.
[186] Sebastian Hack, Gerhard Goos, Optimal register allocation for SSA-form programs in polynomial time, Information Processing Letters 98 (4) (May 2006) 150–155.
[187] Max Hailperin, Cost-optimal code motion, ACM Transactions on Programming Languages and Systems (TOPLAS) 20 (6) (November 1998) 1297–1322.
[188] Gilbert Joseph Hansen, Adaptive Systems for the Dynamic Run-Time Optimization of Programs, PhD thesis, Carnegie Mellon University, Pittsburgh, PA, 1974.

[189] David R. Hanson, Fast allocation and deallocation of memory based on object lifetimes, Software–Practice and Experience 20 (1) (January 1990) 5–12.
[190] Dov Harel, A linear time algorithm for finding dominators in flow graphs and related problems, in: Proceedings of the Seventeenth Annual ACM Symposium on Theory of Computing (STOC), Providence, RI, May 1985, pp. 185–194.
[191] William H. Harrison, A class of register allocation algorithms, Technical Report RC-5342, IBM Thomas J. Watson Research Center, Yorktown Heights, NY, 1975.
[192] William H. Harrison, A new strategy for code generation–The general purpose optimizing compiler, IEEE Transactions on Software Engineering SE-5 (4) (July 1979) 367–373.
[193] Timothy J. Harvey, Reducing the Impact of Spill Code, MS thesis, Department of Computer Science, Rice University, Houston, TX, May 1998.
[194] Amir H. Hashemi, David R. Kaeli, Brad Calder, Efficient procedure mapping using cache line coloring, in: Proceedings of the ACM SIGPLAN '97 Conference on Programming Language Design and Implementation, ACM SIGPLAN Notices 32 (5) (May 1997) 171–182.
[195] Philip J. Hatcher, Thomas W. Christopher, High-quality code generation via bottom-up tree pattern matching, in: Conference Record of the Thirteenth Annual ACMSymposium on Principles of Programming Languages, St. Petersburg Beach, FL, January 1986, pp. 119–130.
[196] Matthew S. Hecht, Jeffrey D. Ullman, Characterizations of reducible flow graphs, Journal of the ACM 21 (3) (July 1974) 367–375.
[197] Matthew S. Hecht, Jeffrey D. Ullman, A simple algorithm for global data flow analysis problems, SIAM Journal on Computing 4 (4) (1975) 519–532.
[198] J. Heller, Sequencing aspects of multiprogramming, Journal of the ACM8 (3) (July 1961) 426–439.
[199] John L. Hennessy, Thomas Gross, Postpass code optimization of pipeline constraints, ACM Transactions on Programming Languages and Systems (TOPLAS) 5 (3) (July 1983) 422–448.
[200] John L. Hennessy, David A. Patterson, Computer Architecture, Sixth Edition: A Quantitative Approach, Morgan Kaufmann Publishers Inc., San Francisco, CA, 2017.
[201] Vincent P. Heuring, The automatic generation of fast lexical analysers, Software–Practice and Experience 16 (9) (September 1986) 801–808.
[202] Michael Hind, Michael Burke, Paul Carini, Jong-Deok Choi, Interprocedural pointer alias analysis, ACMTransactions on Programming Languages and Systems (TOPLAS) 21 (4) (July 1999) 848–894.
[203] Michael Hind, Anthony Pioli, Which pointer analysis should I use?, in: Proceedings of the International Symposium on Software Testing and Analysis, ACM SIGSOFT Software Engineering Notes 25 (5) (September 2000) 113–123.
[204] Christoph M. Hoffmann, Michael J. O'Donnell, Pattern matching in trees, Journal of the ACM 29 (1) (January 1982) 68–95.

[205] John E. Hopcroft, An *n* log *n* algorithm for minimizing states in a finite automaton, in: Zvi Kohavi, Azaria Paz (Eds.), Theory of Machines and Computations: Proceedings, Academic Press, New York, NY, 1971, pp. 189–196.
[206] John E. Hopcroft, Jeffrey D. Ullman, Introduction to Automata Theory, Languages, and Computation, Addison-Wesley, Reading, MA, 1979.
[207] Ellis Horowitz, Sartaj Sahni, Fundamentals of Computer Algorithms, Computer Science Press, Inc., Potomac, MD, 1978.
[208] Lawrence P. Horwitz, Richard M. Karp, Raymond E. Miller, Shmuel Winograd, Index register allocation, Journal of the ACM 13 (1) (January 1966) 43–61.
[209] Susan Horwitz, Phil Pfeiffer, Thomas Reps, Dependence analysis for pointer variables, in: Proceedings of the ACM SIGPLAN '89 Conference on Programming Language Design and Implementation, ACMSIGPLAN Notices 24 (7) (July 1989) 28–40.
[210] Brett L. Huber, Path-selection heuristics for dominator-path scheduling, Master's thesis, Computer Science Department, Michigan Technological University, Houghton, MI, October 1995.
[211] Wen-Mei W. Hwu, Scott A. Mahlke, William Y. Chen, Pohua P. Chang, Nancy J. Warter, Roger A. Bringmann, Roland G. Ouellette, Richard E. Hank, Tokuzo Kiyohara, Grant E. Haab, John G. Holm, Daniel M. Lavery, The superblock: An effective technique for VLIW and superscalar compilation, Journal of Supercomputing–Special Issue on Instruction Level Parallelism 7 (1–2) (May 1993) 229–248.
[212] ISO IEC, Programming Languages–C++, Standard 14882:2017, International Organization for Standardization, Geneva, Switzerland, 2017.
[213] Peter Zilahy Ingerman, Thunks: A way of compiling procedure statements with some comments on procedure declarations, Communications of the ACM 4 (1) (January 1961) 55–58.
[214] Edgar T. Irons, A syntax directed compiler for Algol 60, Communications of the ACM 4 (1) (January 1961) 51–55.
[215] Mark Scott Johnson, Terrence C. Miller, Effectiveness of a machine-level, global optimizer, in: Proceedings of the ACM SIGPLAN '86 Symposium on Compiler Construction, ACM SIGPLAN Notices 21 (7) (July 1986) 99–108.
[216] Stephen C. Johnson, Yacc: Yet another compiler-compiler, Technical Report 32 (Computing Science), AT&T Bell Laboratories, Murray Hill, NJ, 1975.
[217] Stephen C. Johnson, A tour through the portable C compiler, in: Unix Programmer's Manual, 7th Edition, vol. 2b, AT&T Bell Laboratories, Murray Hill, NJ, January 1979.
[218] Walter L. Johnson, James H. Porter, Stephanie I. Ackley, Douglas T. Ross, Automatic generation of efficient lexical processors using finite state techniques, Communications of the ACM 11 (12) (December 1968) 805–813.
[219] Douglas W. Jones, How (not) to code a finite state machine, ACM SIGPLAN Notices 23 (8) (August 1988) 19–22.
[220] S. M. Joshi, Dhananjay M. Dhamdhere, A composite hoisting-strength reduction transformation for global program optimization, International

Journal of Computer Mathematics 11 (1) (1982) 21–44 (part I), International Journal of Computer Mathematics 11 (2) (1982) 111–126 (part II).
[221] John B. Kam, Jeffrey D. Ullman, Global data flow analysis and iterative algorithms, Journal of the ACM 23 (1) (January 1976) 158–171.
[222] John B. Kam, Jeffrey D. Ullman, Monotone data flow analysis frameworks, Acta Informatica 7 (3) (September 1977) 305–317.
[223] Tadao Kasami, An efficient recognition and syntax analysis algorithm for context-free languages, Scientific Report AFCRL-65-758, Air Force Cambridge Research Laboratory, Bedford, MA, 1965.
[224] Ken Kennedy, A global flow analysis algorithm, International Journal of Computer Mathematics 3 (Section A) (December 1971) 5–15.
[225] Ken Kennedy, Global Flow Analysis and Register Allocation for Simple Code Structures, PhD thesis, Courant Institute of Mathematical Sciences, New York University, New York, NY, October 1971.
[226] Ken Kennedy, Global dead computation elimination, SETL Newsletter 111, Courant Institute of Mathematical Sciences, New York University, New York, NY, August 1973.
[227] Ken Kennedy, Reduction in strength using hashed temporaries, in: SETL Newsletter, vol. 102, Courant Institute of Mathematical Sciences, New York University, New York, NY, March 1973.
[228] Ken Kennedy, Use-definition chains with applications, Computer Languages 3 (3) (1978) 163–179.
[229] Ken Kennedy, A survey of data flow analysis techniques, in: Neil D. Jones, Steven S. Muchnik (Eds.), Program Flow Analysis: Theory and Applications, Prentice-Hall, Englewood Cliffs, NJ, 1981.
[230] Ken Kennedy, Linda Zucconi, Applications of graph grammar for program control flow analysis, in: Conference Record of the Fourth ACMSymposium on Principles of Programming Languages, Los Angeles, CA, January 1977, pp. 72–85.
[231] Robert Kennedy, Fred C. Chow, Peter Dahl, Shin-Ming Liu, Raymond Lo, Mark Streich, Strength reduction via SSAPRE, in: Proceedings of the Seventh International Conference on Compiler Construction (CC '98), in: Lecture Notes in Computer Science (LNCS), vol. 1383, Springer-Verlag, Heidelberg, Germany, March 1998, pp. 144–158.
[232] Daniel R. Kerns, Susan J. Eggers, Balanced scheduling: Instruction scheduling when memory latency is uncertain, in: Proceedings of the ACM SIGPLAN '93 Conference on Programming Language Design and Implementation, ACM SIGPLAN Notices 28 (6) (June 1993) 278–289.
[233] Robert R. Kessler, Peep–An architectural description driven peephole optimizer, in: Proceedings of the ACM SIGPLAN '84 Symposium on Compiler Construction, ACM SIGPLAN Notices 19 (6) (June 1984) 106–110.
[234] Gary A. Kildall, A unified approach to global program optimization, in: Conference Record of the ACM Symposium on Principles of Programming Languages, Boston, MA, October 1973, pp. 194–206.

[235] Stephen C. Kleene, Representation of events in nerve nets and finite automata, in: Claude E. Shannon, John McCarthy (Eds.), Automata Studies, in: Annals of Mathematics Studies, vol. 34, Princeton University Press, Princeton, NJ, 1956, pp. 3–41.
[236] Jens Knoop, Oliver Rüthing, Bernhard Steffen, Lazy code motion, in: Proceedings of the ACM SIGPLAN '92 Conference on Programming Language Design and Implementation, ACM SIGPLAN Notices 27 (7) (July 1992) 224–234.
[237] Jens Knoop, Oliver Rüthing, Bernhard Steffen, Lazy strength reduction, International Journal of Programming Languages 1 (1) (March 1993) 71–91.
[238] Donald E. Knuth, A history of writing compilers, Computers and Automation 11 (12) (December 1962) 8–18, Reprinted in: Bary W. Pollack (Ed.), Compiler Techniques, Auerbach, Princeton, NJ, 1972, pp. 38–56.
[239] Donald E. Knuth, On the translation of languages from left to right, Information and Control 8 (6) (December 1965) 607–639.
[240] Donald E. Knuth, The Art of Computer Programming, Addison-Wesley, Reading, MA, 1973[1].
[241] Dexter C. Kozen, Automata and Computability, Springer-Verlag, New York, NY, 1997.
[242] Glenn Krasner (Ed.), Smalltalk-80: Bits of History, Words of Advice, Addison-Wesley, Reading, MA, August 1983.
[243] Sanjay M. Krishnamurthy, A brief survey of papers on scheduling for pipelined processors, SIGPLAN Notices 25 (7) (July 1990) 97–106.
[244] Steven M. Kurlander, Charles N. Fischer, Zero-cost range splitting, in: Proceedings of the ACM SIGPLAN '94 Conference on Programming Language Design and Implementation, ACM SIGPLAN Notices 29 (6) (June 1994) 257–265.
[245] Monica Lam, Software pipelining: An effective scheduling technique for VLIW machines, in: Proceedings of the ACM SIGPLAN '88 Conference on Programming Language Design and Implementation, ACMSIGPLAN Notices 23 (7) (July 1988) 318–328.
[246] David Alex Lamb, Construction of a peephole optimizer, Software–Practice and Experience 11 (6) (June 1981) 639–647.
[247] William Landi, Barbara G. Ryder, Pointer-induced aliasing: A problem taxonomy, in: Proceedings of the Eighteenth Annual ACM Symposium on Principles of Programming Languages, Orlando, FL, January 1991, pp. 93–103.
[248] David Landskov, Scott Davidson, Bruce Shriver, Patrick W. Mallett, Local microcode compaction techniques, ACM Computing Surveys 12 (3) (September 1980) 261–294.
[249] Rudolf Landwehr, Hans-Stephan Jansohn, Gerhard Goos, Experience with an automatic code generator generator, in: Proceedings of the ACM SIG-

[1] *Дональд Э. Кнут*. Искусство программирования. Вильямс, 2019.

PLAN ’82 Symposium on Compiler Construction, ACM SIGPLAN Notices 17 (6) (June 1982) 56–66.

[250] *Лавров С. С.* Об экономии памяти в замкнутых операторных схемах // Журнал вычисл. матем. и матем. физ., 1:4 (1961), 687–701; U.S.S.R. Comput. Math. Math. Phys., 1:3 (1962), 810–828.

[251] Vincent Lefévre, Multiplication by an integer constant, Research Report 4192, INRIA, France, May 2001.

[252] Thomas Lengauer, Robert Endre Tarjan, A fast algorithm for finding dominators in a flowgraph, ACM Transactions on Programming Languages and Systems (TOPLAS) 1 (1) (July 1979) 121–141.

[253] Philip M. Lewis, Richard E. Stearns, Syntax-directed transduction, Journal of the ACM 15 (3) (July 1968) 465–488.

[254] Vincenzo Liberatore, Martin Farach-Colton, Ulrich Kremer, Evaluation of algorithms for local register allocation, in: Proceedings of the Eighth International Conference on Compiler Construction (CC ’99), in: Lecture Notes in Computer Science (LNCS), vol. 1575, Springer-Verlag, Heidelberg, Germany, 1999, pp. 137–152.

[255] Henry Lieberman, Carl Hewitt, A real-time garbage collector based on the lifetimes of objects, Communications of the ACM 26 (6) (June 1983) 419–429.

[256] Barbara Liskov, Russell Atkinson, Toby Bloom, Eliot Moss, J. Craig Schaffert, Robert Scheifler, Alan Snyder, CLU Reference Manual, Lecture Notes in Computer Science (LNCS), vol. 114, Springer-Verlag, Heidelberg, Germany, 1981.

[257] Jack L. Lo, Susan J. Eggers, Improving balanced scheduling with compiler optimizations that increase instruction-level parallelism, in: Proceedings of the ACM SIGPLAN ’95 Conference on Programming Language Design and Implementation, ACM SIGPLAN Notices 30 (6) (June 1995) 151–162.

[258] Raymond Lo, Fred Chow, Robert Kennedy, Shin-Ming Liu, Peng Tu, Register promotion by sparse partial redundancy elimination of loads and stores, in: Proceedings of the ACM SIGPLAN ’98 Conference on Programming Language Design and Implementation, ACM SIGPLAN Notices 33 (5) (May 1998) 26–37.

[259] P. Geoffrey Lowney, Stefan M. Freudenberger, Thomas J. Karzes, W.D. Lichtenstein, Robert P. Nix, John S. O’Donnell, John C. Ruttenburg, The multiflow trace scheduling compiler, The Journal of Supercomputing–Special Issue 7 (1–2) (March 1993) 51–142.

[260] Edward S. Lowry, C.W. Medlock, Object code optimization, Communications of the ACM 12 (1) (January 1969) 13–22.

[261] John Lu, Keith D. Cooper, Register promotion in C programs, in: Proceedings of the ACM SIGPLAN ’97 Conference on Programming Language Design and Implementation, ACM SIGPLAN Notices 32 (5) (May 1997) 308–319.

[262] John Lu, Rob Shillingsburg, Clean: Removing useless control flow, Unpublished, Department of Computer Science, Rice University, Houston, TX, June 1994.

[263] Peter Lucas, Die Strukturanalyse von Formelübersetzern, Elektronische Rechenanlagen 3 (4) (August 1961) 159–167.
[264] Peter W. Markstein, Victoria Markstein, F. Kenneth Zadeck, Reassociation and strength reduction, Unpublished book chapter.
[265] Henry Massalin, Superoptimizer–A look at the smallest program, in: Proceedings of the Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-II), ACM SIGPLAN Notices 22 (10) (October 1987) 122–126.
[266] John McCarthy, Recursive functions of symbolic expressions and their computation by machine, Part I, Communications of the ACM 3 (4) (April 1960) 184–195.
[267] John McCarthy, Lisp–Notes on its past and future, in: Proceedings of the 1980 ACM Conference on Lisp and Functional Programming, Stanford University, Stanford, CA, 1980, pp. v–viii.
[268] William M. McKeeman, Peephole optimization, Communications of the ACM8 (7) (July 1965) 443–444.
[269] Kathryn S. McKinley, Steve Carr, Chau-Wen Tseng, Improving data locality with loop transformations, ACM Transactions on Programming Languages and Systems (TOPLAS) 18 (4) (July 1996) 424–453.
[270] Robert McNaughton, H. Yamada, Regular expressions and state graphs for automata, IRE Transactions on Electronic Computers EC-9 (1) (March 1960) 39–47.
[271] Robert Metzger, Sean Stroud, Interprocedural constant propagation: An empirical study, ACM Letters on Programming Languages and Systems (LOPLAS) 2 (1–4) (March–December 1993) 213–232.
[272] Ana Milanova, Atanas Rountev, Barbara G. Ryder, Precise call graphs for C programs with function pointers, Automated Software Engineering 11 (1) (January 2004) 7–26.
[273] Terrence C. Miller, Tentative Compilation: A Design for an APL Compiler, PhD thesis, Yale University, New Haven, CT, May 1978. See also the paper of the same title in: Proceedings of the International Conference on APL: Part 1, New York, NY, 1979, pp. 88–95.
[274] Robin Milner, Mads Tofte, Robert Harper, David MacQueen, The Definition of Standard ML–Revised, MIT Press, Cambridge, MA, 1997.
[275] James Strother Moore, The Interlisp Virtual Machine specification, Technical Report CSL 76-5, Xerox Palo Alto Research Center, Palo Alto, CA, September 1976.
[276] Etienne Morel, Claude Renvoise, Global optimization by suppression of partial redundancies, Communications of the ACM 22 (2) (February 1979) 96–103.
[277] Robert Morgan, Building an Optimizing Compiler, Digital Press (an imprint of Butterworth–Heineman), Boston, MA, February 1998.
[278] Rajeev Motwani, Krishna V. Palem, Vivek Sarkar, Salem Reyen, Combining register allocation and instruction scheduling, Technical Report 698, Cou-

rant Institute of Mathematical Sciences, New York University, New York, NY, July 1995.
[279] Steven S. Muchnick, Advanced Compiler Design & Implementation, Morgan Kaufmann, San Francisco, CA, 1997.
[280] Frank Mueller, David B. Whalley, Avoiding unconditional jumps by code replication, in: Proceedings of the ACM SIGPLAN '92 Conference on Programming Language Design and Implementation, ACMSIGPLAN Notices 27 (7) (July 1992) 322–330.
[281] Thomas P. Murtagh, An improved storage management scheme for block structured languages, ACM Transactions on Programming Languages and Systems (TOPLAS) 13 (3) (July 1991) 372–398.
[282] Peter Naur (editor), J. W. Backus, F. L. Bauer, J. Green, C. Katz, J. McCarthy, A. J. Perlis, H. Rutishauser, K. Samelson, B. Vauquois, J. H. Wegstein, A. van Wijngaarden, M. Woodger, Revised report on the algorithmic language Algol 60, Communications of the ACM 6 (1) (January 1963) 1–17.
[283] E. Ketcha Ngassam, Bruce W. Watson, Derrick G. Kourie, Hardcoding finite state automata processing, in: Proceedings of the 2003 Annual Research Conference of the South African Institute of Computer Scientists and Information Technologists on Enablement through Technology (SAICSIT '03), September 2003, pp. 111–121.
[284] Brian R. Nickerson, Graph coloring register allocation for processors with multiregister operands, in: Proceedings of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation, ACM SIGPLAN Notices 25 (6) (June 1990) 40–52.
[285] Cindy Norris, Lori L. Pollock, A scheduler-sensitive global register allocator, in: Proceedings of Supercompting '93, Portland, OR, November 1993, pp. 804–813.
[286] Cindy Norris, Lori L. Pollock, An experimental study of several cooperative register allocation and instruction scheduling strategies, in: Proceedings of the Twenty-Eighth Annual International Symposium on Microarchitecture (MICRO-28), Ann Arbor, MI, December 1995, pp. 169–179.
[287] Kristen Nygaard, Ole-Johan Dahl, The development of the SIMULA languages, in: Proceedings of the First ACM SIGPLAN Conference on the History of Programming Languages, Los Angeles, CA, USA, ACM SIGPLAN Notices 13 (8) (August 1978) 245–272.
[288] Michael Paleczny, Christopher A. Vick, Cliff Click, The Java HotSpot Server Compiler, in: JVM '01: Proceedings of the 2001 Java Virtual Machine Research and Technology Symposium, vol. 1, Monterey, CA, April 2001, pp. 1–12.
[289] Jinpyo Park, Soo-Mook Moon, Optimistic register coalescing, in: Proceedings of the 1998 International Conference on Parallel Architecture and Compilation Techniques (PACT), October 1998, pp. 196–204.
[290] Eduardo Pelegrí-Llopart, Susan L. Graham, Optimal code generation for expression trees: An application of BURS theory, in: Proceedings of the Fifteenth Annual ACM Symposium on Principles of Programming Languages, San Diego, CA, January 1988, pp. 294–308.

[291] Thomas J. Pennello, Very fast LR parsing, in: Proceedings of the ACM SIGPLAN '86 Symposium on Compiler Construction, ACM SIGPLAN Notices 21 (7) (July 1986) 145–151.
[292] Fernando Magno Quintão Pereira, Jens Palsberg, Register allocation via coloring of chordal graphs, in: Proceedings of the Asian Symposium on Programming Languages and Systems (ASPLAS '05), in: Lecture Notes in Computer Science (LNCS), vol. 3780, Springer, Berlin, Heidelberg, November 2005, pp. 315–329.
[293] Karl Pettis, Robert C. Hansen, Profile guided code positioning, in: Proceedings of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation, ACM SIGPLAN Notices 25 (6) (June 1990) 16–27.
[294] Shlomit S. Pinter, Register allocation with instruction scheduling: A new approach, in: Proceedings of the ACM SIGPLAN '93 Conference on Programming Language Design and Implementation, ACMSIGPLAN Notices 28 (6) (June 1993) 248–257.
[295] Gordon D. Plotkin, Call-by-name, call-by-value and the λ-calculus, Theoretical Computer Science 1 (2) (December 1975) 125–159.
[296] Massimiliano Poletto, Vivek Sarkar, Linear scan register allocation, ACM Transactions on Programming Languages and Systems (TOPLAS) 21 (5) (September 1999) 895–913.
[297] Todd A. Proebsting, Simple and efficient BURS table generation, in: Proceedings of the ACM SIGPLAN '92 Conference on Programming Language Design and Implementation, ACM SIGPLAN Notices 27 (7) (July 1992) 331–340.
[298] Todd A. Proebsting, Optimizing an ANSI C interpreter with superoperators, in: Proceedings of the Twenty-Second ACMSIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Francisco, CA, January 1995, pp. 322–332.
[299] Todd A. Proebsting, Charles N. Fischer, Linear-time, optimal code scheduling for delayed-load architectures, in: Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation, ACM SIGPLAN Notices 26 (6) (June 1991) 256–267.
[300] Reese T. Prosser, Applications of boolean matrices to the analysis of flow diagrams, in: Proceedings of the Eastern Joint Computer Conference, Institute of Radio Engineers, New York, NY, December 1959, pp. 133–138.
[301] Paul W. Purdom Jr., Edward F. Moore, Immediate predominators in a directed graph [H], Communications of the ACM 15 (8) (August 1972) 777–778.
[302] Michael O. Rabin, Dana Scott, Finite automata and their decision problems, IBM Journal of Research and Development 3 (2) (April 1959) 114–125.
[303] Brian Randell, L. J. Russell, Algol 60 Implementation: The Translation and Use of Algol 60 Programs on a Computer, Academic Press, London, UK, January 1964.
[304] Bob R. Rau, C. D. Glaeser, Some scheduling techniques and an easily schedulable horizontal architecture for high performance scientific computing, in: Proceedings of the Fourteenth Annual Workshop on Microprogramming (MICRO-14), Chatham, MA, December 1981, pp. 183–198.

[305] John H. Reif, Symbolic programming analysis in almost linear time, in: Conference Record of the Fifth Annual ACM Symposium on Principles of Programming Languages, Tucson, AZ, January 1978, pp. 76–83.
[306] John H. Reif, Harry R. Lewis, Symbolic evaluation and the global value graph, in: Conference Record of the Fourth ACM Symposium on Principles of Programming Languages, Los Angeles, CA, January 1977, pp. 104–118.
[307] Thomas Reps, “Maximal-munch” tokenization in linear time, ACM Transactions on Programming Languages and Systems (TOPLAS) 20 (2) (March 1998) 259–273.
[308] Martin Richards, The portability of the BCPL compiler, Software–Practice and Experience 1 (2) (April–June 1971) 135–146.
[309] Steve Richardson, Mahadevan Ganapathi, Interprocedural analysis versus procedure integration, Information Processing Letters 32 (3) (August 1989) 137–142.
[310] Ronald Rivest, On self-organizing sequential search heuristics, Communications of the ACM 19 (2) (February 1976) 63–67.
[311] Anne Rogers, Kai Li, Software support for speculative loads, in: Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-V), ACM SIGPLAN Notices 27 (9) (September 1992) 38–50.
[312] Barry K. Rosen, Mark N. Wegman, F. Kenneth Zadeck, Global value numbers and redundant computations, in: Proceedings of the Fifteenth Annual ACM Symposium on Principles of Programming Languages, San Diego, CA, January 1988, pp. 12–27.
[313] Daniel J. Rosenkrantz, Richard Edwin Stearns, Properties of deterministic topdown grammars, Information and Control 17 (3) (October 1970) 226–256.
[314] Barbara G. Ryder, Constructing the call graph of a program, IEEE Transactions on Software Engineering SE-5 (3) (May 1979) 216–226.
[315] A. V. S. Sastry, Roy D. C. Ju, A new algorithm for scalar register promotion based on SSA form, in: Proceedings of the ACM SIGPLAN ’98 Conference on Programming Language Design and Implementation, ACM SIGPLAN Notices 33 (5) (May 1998) 15–25.
[316] Kirk Sattley, Allocation of storage for arrays in Algol 60, Communications of the ACM 4 (1) (January 1961) 60–65.
[317] Randolph G. Scarborough, Harwood G. Kolsky, Improved optimization of FORTRAN object programs, IBM Journal of Research and Development 24 (6) (November 1980) 660–676.
[318] Philip J. Schielke, Stochastic Instruction Scheduling, PhD thesis, Department of Computer Science, Rice University, Houston, TX, May 2000, Technical Report TR00-370, Computer Science Department, Rice University, 2000.
[319] Herb Schorr, William M. Waite, An efficient machine-independent procedure for garbage collection in various list structures, Communications of the ACM 10 (8) (August 1967) 501–506.

[320] Jacob T. Schwartz, On programming: An interim report on the SETL project. Installment II: The SETL language and examples of its use, Technical Report, Courant Institute of Mathematical Sciences, New York University, New York, NY, October 1973.
[321] Ravi Sethi, Jeffrey D. Ullman, The generation of optimal code for arithmetic expressions, Journal of the ACM 17 (4) (October 1970) 715–728.
[322] Marc Shapiro, Susan Horwitz, Fast and accurate flow-insensitive points-to analysis, in: Proceedings of the Twenty-Fourth ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Paris, France, January 1997, pp. 1–14.
[323] Robert M. Shapiro, Harry Saint, The representation of algorithms, Technical Report CA-7002-1432, Massachusetts Computer Associates, February 1970.
[324] Peter B. Sheridan, The arithmetic translator-compiler of the IBM FORTRAN automatic coding system, Communications of the ACM 2 (2) (February 1959) 9–21.
[325] Olin Shivers, Control flow analysis in Scheme, in: Proceedings of the ACM SIGPLAN '88 Conference on Programming Language Design and Implementation, ACM SIGPLAN Notices 23 (7) (July 1988) 164–174.
[326] L. Taylor Simpson, Value-Driven Redundancy Elimination, PhD thesis, Department of Computer Science, Rice University, Houston, TX, 1996, Technical Report TR96-308, Computer Science Department, Rice University, 1996.
[327] Michael Sipser, Introduction to the Theory of Computation, PWS Publishing Co., Boston, MA, December 1996.
[328] Richard L. Sites, Daniel R. Perkins, Universal P-code definition, version 0.2, Technical Report 78-CS-C29, Department of Applied Physics and Information Sciences, University of California, San Diego, San Diego, CA, January 1979.
[329] Daniel Dominic Sleator, Robert Endre Tarjan, Amortized efficiency of list update and paging rules, Communications of the ACM 28 (2) (February 1985) 202–208.
[330] Michael D. Smith, Mark Horowitz, Monica S. Lam, Efficient superscalar performance through boosting, in: Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-V), ACM SIGPLAN Notices 27 (9) (September 1992) 248–259.
[331] Michael D. Smith, Norman Ramsey, Glenn Holloway, A generalized algorithm for graph-coloring register allocation, in: Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation, ACM SIGPLAN Notices 39 (6) (May 2004) 277–288.
[332] Mark Smotherman, Sanjay M. Krishnamurthy, P. S. Aravind, David Hunnicutt, Efficient DAG construction and heuristic calculation for instruction scheduling, in: Proceedings of the Twenty-Fourth Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-24), Albuquerque, NM, August 1991, pp. 93–102.

[333] Arthur Sorkin, Some comments on “A solution to a problem with Morel and Renvoise’s ‘Global optimization by suppression of partial redundancies,”’ ACM Transactions on Programming Languages and Systems (TOPLAS) 11 (4) (October 1989) 666–668.
[334] Thomas C. Spillman, Exposing side-effects in a PL/1 optimizing compiler, in: Foundations and Systems, in: Information Processing: Proceedings of IFIP Congress 1971, vol. 1, Ljubljana, Yugoslavia, August 1971, North-Holland Publishing Company, Amsterdam, 1972, pp. 376–381.
[335] Guy L. Steele Jr., Rabbit: A compiler for Scheme, Technical Report AI-TR-474, MIT Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Cambridge, MA, May 1978.
[336] Guy L. Steele Jr., Richard P. Gabriel, The evolution of LISP, in: Thomas J. Bergin, Richard G. Gibson (Eds.), History of Programming Languages–II, ACM Press, New York, NY, January 1996, pp. 233–330.
[337] Mark Stephenson, Saman Amarasinghe, Predicting unroll factors using supervised classification, in: CGO ’05: Proceedings of the International Symposium on Code Generation and Optimization, San Jose, CA, IEEE Computer Society,Washington, DC, March 2005, pp. 123–134.
[338] Philip H. Sweany, Steven J. Beaty, Post-compaction register assignment in a retargetable compiler, in: Proceedings of the Twenty-Third Annual International Symposium andWorkshop on Microprogramming and Micro-architecture (MICRO-23), Orlando, FL, November 1990, pp. 107–116.
[339] Philip H. Sweany, Steven J. Beaty, Dominator-path scheduling–A global scheduling method, in: Proceedings of the Twenty-Fifth Annual International Symposium on Microarchitecture (MICRO-25), ACM SIGMICRO Newsletter 23 (1–2) (December 1992) 260–263.
[340] Deian Tabakov, Moshe Y. Vardi, Experimental evaluation of classical automata constructions, in: Proceedings of the Twelfth International Conference on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR ’05), in: Lecture Notes in Computer Science (LNCS), vol. 3835, Springer, Berlin, Heidelberg, December 2005, pp. 396–411.
[341] Robert Endre Tarjan, Testing flow graph reducibility, Journal of Computer and System Sciences 9 (3) (December 1974) 355–365.
[342] Robert Endre Tarjan, Fast algorithms for solving path problems, Journal of the ACM 28 (3) (July 1981) 594–614.
[343] Robert Endre Tarjan, A unified approach to path problems, Journal of the ACM 28 (3) (July 1981) 577–593.
[344] Robert Endre Tarjan, John H. Reif, Symbolic program analysis in almost-linear time, SIAM Journal on Computing 11 (1) (February 1982) 81–93.
[345] Ken Thompson, Programming techniques: Regular expression search algorithm, Communications of the ACM 11 (6) (1968) 419–422.
[346] Steven W. K. Tjiang, Twig reference manual, Technical Report CSTR 120, Computing Sciences, AT&T Bell Laboratories, Murray Hill, NJ, January 1986.
[347] Linda Torczon, Compilation Dependences in an Ambitious Optimizing Compiler, PhD thesis, Department of Computer Science, Rice University, Houston, TX, 1985.

[348] Jeffrey D. Ullman, Fast algorithms for the elimination of common subexpressions, Acta Informatica 2 (3) (July 1973) 191–213.
[349] David Ungar, Generation scavenging: A non-disruptive high performance storage reclamation algorithm, in: Proceedings of the First ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments, ACM SIGSOFT Software Engineering Notes 9 (3) (May 1984) 157–167.
[350] Anand Venkat, Mary Hall, Michelle Strout, Loop and data transformations for sparse matrix code, in: Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, ACM SIGPLAN Notices 50 (6) (June 2015) 521–532.
[351] Victor Vyssotsky, Peter Wegner, A graph theoretical FORTRAN source language analyzer, Manuscript, AT&T Bell Laboratories, Murray Hill, NJ, 1963.
[352] WilliamWaite, Gerhard Goos, Compiler Construction, Springer-Verlag, New York, NY, 1984.
[353] William M. Waite, The cost of lexical analysis, Software–Practice and Experience 16 (5) (May 1986) 473–488.
[354] David W. Wall, Global register allocation at link time, in: Proceedings of the ACM SIGPLAN '86 Symposium on Compiler Construction, ACM SIGPLAN Notices 21 (7) (July 1986) 264–275.
[355] Bruce W. Watson, A taxonomy of deterministic finite automata minimization algorithms, Computing Science Report 93/44, Department of Mathematics and Computing Science, Eindhoven University of Technology, Eindhoven, The Netherlands, December 1993.
[356] Bruce W. Watson, A fast new semi-incremental algorithm for the construction of minimal acyclic DFAs, in: Proceedings of the Third International Workshop on Implementing Automata (WIA 1998), in: Lecture Notes in Computer Science (LNCS), vol. 1660, Springer, Berlin, Heidelberg, 1999, pp. 121–132.
[357] Bruce W. Watson, A fast and simple algorithm for constructing minimal acyclic deterministic finite automata, Journal of Universal Computer Science 8 (2) (February 2002) 363–367.
[358] Mark N. Wegman, F. Kenneth Zadeck, Constant propagation with conditional branches, in: Conference Record of the Twelfth Annual ACM Symposium on Principles of Programming Languages, New Orleans, LA, January 1985, pp. 291–299.
[359] Mark N. Wegman, F. Kenneth Zadeck, Constant propagation with conditional branches, ACM Transactions on Programming Languages and Systems (TOPLAS) 13 (2) (April 1991) 181–210.
[360] William E. Weihl, Interprocedural data flow analysis in the presence of pointers, procedure variables, and label variables, in: Conference Record of the Seventh Annual ACM Symposium on Principles of Programming Languages, Las Vegas, NV, January 1980, pp. 83–94.
[361] Clark Wiedmann, Steps toward an APL compiler, in: Proceedings of the International Conference on APL, ACM SIGAPL APL Quote Quad 9 (4) (June 1979) 321–328.

[362] Paul R. Wilson, Uniprocessor garbage collection techniques, in: Proceedings of the International Workshop on Memory Management, in: Lecture Notes in Computer Science (LNCS), vol. 637, Springer-Verlag, Heidelberg, Germany, 1992, pp. 1–42.

[363] Robert P. Wilson, Monica S. Lam, Efficient context-sensitive pointer analysis for C programs, in: Proceedings of the ACM SIGPLAN '95 Conference on Programming Language Design and Implementation, ACMSIGPLAN Notices 30 (6) (June 1995) 1–12.

[364] Michael Wolfe, High Performance Compilers for Parallel Computing, Addison Wesley, Redwood City, CA, 1996.

[365] Derick Wood, The theory of left-factored languages, part 1, The Computer Journal 12 (4) (November 1969) 349–356.

[366] Derick Wood, The theory of left-factored languages, part 2, The Computer Journal 13 (1) (February 1970) 55–62.

[367] Derick Wood, A further note on top-down deterministic languages, The Computer Journal 14 (4) (November 1971) 396–403.

[368] William Wulf, Richard K. Johnsson, Charles B. Weinstock, Steven O. Hobbs, Charles M. Geschke, The Design of an Optimizing Compiler, Programming Languages Series, American Elsevier Publishing Company, Inc., New York, NY, 1975.

[369] Cliff Young, David S. Johnson, David R. Karger, Michael D. Smith, Near-optimal intraprocedural branch alignment, in: Proceedings of the ACM SIGPLAN '97 Conference on Programming Language Design and Implementation, ACM SIGPLAN Notices 32 (5) (May 1997) 183–193.

[370] Daniel H. Younger, Recognition and parsing of context-free languages in time n^3, Information and Control 10 (2) (1967) 189–208.

[371] F. Kenneth Zadeck, Incremental data flow analysis in a structured program editor, in: Proceedings of the ACM SIGPLAN '84 Symposium on Compiler Construction, ACM SIGPLAN Notices 19 (6) (June 1984) 132–143.

[372] Weilei Zhang, Barbara G. Ryder, Automatic construction of accurate application call graph with library call abstraction for Java, Journal of Software Maintenance and Evolution: Research and Practice 19 (4) (July 2007) 231–252.

Предметный указатель

Символы

ϵ-переход, 67, 82, 84, 98
ϵ-продукция, 123
φ-функция, 212, 443, 489
  алгоритм вставки, 490, 492, 497, 499, 535
  аргументы, 214, 490
  вставка, 213, 490, 566
  избыточные, 213

A

Action таблица, 140
Algol, идентификатор, 65, 108
Algol-подобные языки, 57, 61, 114, 161, 204, 251, 294, 297, 300, 302, 389, 450

B

break предложение, 381, 529

F

FIRST множество, 127
FOLLOW множество, 129

G

Goto таблица, 140

I

if–then–else
  неоднозначность, 114
  LR(1)-разбор, 157
ILOC, 783

J

JAVA, 24, 25, 47, 51, 57, 58, 61, 62, 118, 178, 194, 203, 204, 220, 252, 262, 306, 366, 381, 398, 520, 524, 680, 737, 738, 794, 809
  выражение, 259
  HotSpot Server компилятор, 217, 222, 762, 764

L

L-значение, 354
LALR(1), 171
LL(1)
  парсеры, 107
  условие, 131
LR(1)
  алгоритм разбора, 140
  канонический набор, 151
  ошибка shift-reduce, 158
  парсеры, 107
  элементы, 147
  closure функция, 149
  goto функция, 150

M

MAXLIVE, 700

N

NaN, 418

R

R-значение, 353
rarp, 40

A

Абстрактное синтаксическое дерево (АСД), 180
  близкое к исходному коду, 230
  построение, 184, 236, 242
Автоматическая переменная, 220, 270
Адрес возврата, 298
Адресное пространство
  виртуальное, 273
  работающего процесса, 753

физическое, 274, 289
Адресуемость, 302, 318, 324, 339
дисплей, 322
имен, 297
поле, 303, 323
связи доступа, 320
статическая связь, 320
структуры данных, 325
Активация, 298
Активное значение, 216
Алгебраические тождества, 415, 418, 469, 542, 565, 593, 609, 616, 635, 758
Альтернация, 59
Анализ зависимостей, 38
Анализ потока данных, 38, 438, 468
возвратная скачковая функция, 524
встреча на всех путях, 474
доступные выражения, 483
задача о потенциальной модификации, 486
задача о потенциальной ссылке, 487
использование, 189
итеративный алгоритм, 439
неприводимый граф, 526
обратная задача, 471
определение, 189
оптимизм, 515
пессимизм, 516
потоконезависимость, 486
приводимый граф, 526
прямая задача, 471
уравнения доминирования, 472
DEDef, 484
DefKill, 477, 484
ExprKill, 477, 486
LiveOut, 438, 477
UEExpr, 477, 486
UEVar, 439, 477
VarKill, 439
Антизависимость, 650
Арена, выделение памяти, 332, 802, 804
Арифметические операторы, 33, 116, 349
предшествование, 117, 425
Арифметические операции, 118, 225, 263, 360, 370, 380, 435, 602
Архитектура системы команд (ISA), 24, 184, 193, 280, 318, 358, 369, 447, 596, 597, 649

Б

Базовое имя, 490
Базовый адрес, 318
Байт-код, 25, 194
Балансировка дерева по высоте, 414, 421
Безопасность оптимизации, 400, 407
Бжозовского алгоритм, 98
Бинарная операция, 355, 416
Бинарные операторы, 162, 175
Бинарный узел, 236, 625, 637
Битовые векторы, 798
Бит пометки, 336
Блочно-непрерывное распределение памяти, 207, 803
Булево выражение, 368
Булево дополнение, 162
Булево значение, 368
Булево сравнение, 37
Булевый оператор, 349, 368
Бэкуса–Наура форма, 109, 119

В

Вектор, 275
предварительной информации, 365
Ветвь
выбранная, 193, 199
неявная, 193, 199
явная, 95
Видимость, 256
Виртуальная машина JAVA (JVM), 25, 194, 217, 330, 520
Виртуальное адресное пространство, 273
Виртуальное имя, 210
Виртуальный регистр, 41, 210, 686
Внутрипроцедурная оптимизация, 412
Внутрипроцедурная техника, 191
Восходящий обход, 266, 617, 731
Восходящий парсер, 106, 118, 138, 140
Восходящий поток информации, 234
Восходящий проход распределения, 731
Время жизни, 213, 219, 415
Встраивание, 450
Вывод, 35, 111
Выделение памяти
блочно-непрерывное, 207, 803
в куче, 306

для записей объектов, 310
для записи активации (ЗА), 305
затраты, 307
Вызываемая сторона, 295
Вызывающая сторона, 295
Выравнивания ограничение, 280
Вычисление
адреса, 89, 211, 318, 321, 361, 402, 695
адреса в массиве, 275
многомерном, 362, 402, 407
одномерном, 360
являющемся параметром, 365
неподвижной точки, 75

Г

Генератор кодогенераторов, 599
Генерирование кода с отбрасыванием, 749
Глобальная оптимизация, 412
Глобальное имя, 495
Глобальное устранение общих подвыражений, 484
Глобальные методы, 412
Глобальный анализ потока данных, 438
Глобальный дисплей, 320, 322, 327
Грамматика, 33
контекстно-свободная, 105
неоднозначная, 114, 119, 157, 158, 159, 631
однозначная, 138
свободная от возвратов, 126, 131
Граф
вызовов, 191
построение, 518
зависимостей, 642
зависимостей по данным, 189
интерференции, 691, 708
потока управления, 180, 187, 197, 411, 469, 543, 666, 752
построение, 198
представление, 804
Грязное значение, 692

Д

Двоичное дерево, 349, 803, 809, 816
Двоичный поиск, 208, 348, 382, 383, 393
Двойная буферизация, 92
Деоптимизация, 768
Дерево-образец, 616
Дерево разбора, 112, 116, 183
Деревья, связанные с синтаксисом, 183
Деструктивная операция, 193
Детерминированный конечный автомат (ДКА), 68, 109, 156, 206, 630
механизмы допускания, 68
минимизация, 76, 98, 780
Диспетчеризация, 311
Дисплей, 322
Длина плана, 640, 647
Доминирование, 412, 471
граница, 492
дерево, 493
Дополнение, 61
Достигающие определения, 477, 484
Достижимость, 484
Доступность, 483
Доступные выражения, 477, 483

Е

Единица компиляции, 459

Ж

Живая переменная, 438

З

Заключительная последовательность, 326
Закороченное вычисление, 370
Замена теста линейной функции (LFTR), 575, 589
Замещение в стеке, 773
Замкнутая структура классов, 253
Замыкание, 60, 301
свойства, 64
Запись активации (ЗА), 270, 274, 293, 302, 307, 313, 335, 450, 599, 702, 744, 749, 774, 785
выделение памяти, 305, 327
в куче, 306
в стеке, 305
статическое, 307
объединение, 307
Зарезервированные слова, 51

И

Избыточность, 414, 550
Имена
  декорирование, 318
  разрешение, 202, 245
  эквивалентность по, 265
Индуктивная переменная, 582
Интервал
  жизни, 443, 687, 698
    восстановление, 218, 685, 697
    выгрузка, 685, 697
  инициации, 670
Интервальный граф, 729
Интерпретатор, 25
Интерференция, 690
Использование, 189
История выполнения, 298

К

Квалифицированное имя, 254
Кеш, 288
  встроенный, 312
  методов, 312
  производительность, 287
  частота попаданий, 289
Класс, 251
  анализ иерархии, 192
  метод, 251
  повышающее приведение, 342
  понижающее приведение, 342
  экземпляр, 251
Классическая грамматика выражений, 116
Классы памяти, 270
Клини замыкание, 60, 64, 66, 96
Клонирование суперблока, 571
Ключевое слово, 51
Код условия, 372
Компенсационный код, 664
Компилятор, 23
Конвейер, 643
Конечное замыкание, 60, 66
Конечный автомат, 54
Консолидация, 414, 710
  консервативная, 723
  повторяющаяся, 724
Константа
  пул, 202, 205, 220
  свертка, 415, 462
Контекстно-свободная грамматика, 105, 109
  бесполезная продукция, 166
  Бэкуса–Наура форма, 109
  вывод, 111
  левая рекурсия, 122
  левая факторизация, 131
  левосторонний вывод, 113
  неоднозначность, 114
  нетерминальный символ, 110
  правая рекурсия, 123
  правосторонний вывод, 113
  предиктивная, 126
  предложение, 110
  продукция, 110
  свободная от возвратов, 126
  сентенциальная форма, 111
  терминальный символ, 110
  ϵ-продукция, 123
Копирование, 337
Критический путь, 649
Критическое ребро, 506, 557
Куча, 272, 329
  выделение памяти, 306
  выделение первого подходящего блока, 330
  затраты на управление, 310
  подсчет ссылок, 334
  распределители с несколькими пулами, 332
  сборка мусора, 335

Л

Левая рекурсия, 122
  косвенная, 124
  прямая, 122
  устранение, 122
Левая факторизация, 131
Лексема, 50
Лексические иерархии, 246
Лексические области видимости, 203, 246
Линейная промежуточная форма, 806
Листовая процедура, 307, 562
Локальная нумерация значений (ЛНЗ), 415, 517, 540, 551, 748, 750
Локальная оптимальность, 629
Локальная оптимизация, 410

М

Массив
  размещение по столбцам, 402
  совместимость, 263
Машинная зависимость, 538
Машинная независимость, 538
Межпроцедурная оптимизация, 412
Межпроцедурный анализ данных, 191
Мертвая вспомогательная структура, 802
Мертвая операция, 549, 610
Мертвая последовательность, 561
Мертвая циклическая структура, 338
Мертвая φ-функция, 491, 495, 498, 535
Мертвое значение, 612
Мертвый код, 542, 549, 589, 759
  устранение, 363, 463, 542, 609, 616, 748, 806
Мертвый объект, 334
Мертвый эффект, 610
Метод, 251
  вызов, 310
  кеш, 312
Микросинтаксис, 51
Множественное наследование, 256
Монотонная функция, 75

Н

Наследование, 252
  иерархии, 203, 252
Неблокирующие операции, 642
Недетерминированный конечный автомат (НКА), 67, 742
  конфигурация, 69
Неоднозначная грамматика, 114, 157, 630
Неоднозначная ссылка, 421
Неоднозначное значение, 216
Неоднозначность, 114
Неоднозначный условный переход, 192
Непосредственный доминатор, 472
Нерегулярная переменная, 220
Нетерминальный символ, 110
Нумерация
  в обратном порядке, 474
  значений, 414

О

Область
  видимости, 202
  данных, 219, 270
  оптимизации, 410
Обратная задача о потоке данных, 471
Обратное ребро, 570, 746
Общепрограммная оптимизация, 412
Объект, 251
Однозначное значение, 216
Однооператорные блоки, 189
Оконная оптимизация, 242, 597, 608
Окончание, 138
Оператор встречи, 474
Операция
  бесполезная, 542
  деревья, 616
  недостижимая, 542
  планирование, 645
  потока управления, 788
Опережающий символ, 122
Определение, 189
  до использования, 164
Определяющее вхождение, 239
Оптимизатор, 30, 37
Оптимизация, 32, 398, 537, 684
  балансировка дерева по высоте, 421
  внутрипроцедурная, 412
  встраивание, 450
  глобальная, 412
  глобальное размещение кода, 443
  замена теста линейной функции, 589
  клонирование суперблока, 570
  локальная, 410
  локальная нумерация значений, 414
  межпроцедурная, 412
  нахождение неинициализированных переменных, 438
  общепрограммная, 412
  отложенное перемещение кода, 550
  переименование, 573
  поднятие кода, 558
  последовательность, 590
  разворачивание цикла, 434
  размещение процедуры, 454
  распространение констант, 513
  расчленение цикла, 572
  региональная, 411
  снижение стоимости операций, 580
  суперлокальная нумерация значений, 431
  устранение бесполезного кода, 543

устранение бесполезного потока управления, 545
устранение мертвого кода, 542
устранение недостижимого кода, 549
Оптимистические алгоритмы, 516
Ориентированный ациклический граф (ОАГ), 183, 185, 187, 631, 635, 646, 680
Освобождение памяти
неявное, 307, 333
ограниченное по времени, 335
стоимость, 338
явное, 307
Отклонение, 298
Открытая структура классов, 254
Отложенное перемещение кода (ОПК), 550
Ошибка
reduce-reduce, 160
shift-reduce, 158

П

Пакетные сборщики мусора, 335
Память
выделение, 298, 305, 330
модель, 215
ограниченность, 405
режимы адресации, 783
Параллелизм на уровне команд (ILP), 423, 429, 436, 643, 644, 648
Параметрический полиморфизм, 269
Парсер, 35, 105
Передача
параметров
по значению, 314
по значению с модификацией, 314
по имени, 317
по ссылке, 315
по ссылке, 324
Перенос, 563
Планирование
команд, 39, 42, 638
выбор, 39, 596
корректное, 647
обратное, 660
программный конвейер, 670, 673
сбалансированное, 659, 680
по модулю, 674
План ядра, 677
Подкласс, 203, 252
Подсчет ссылок, 334
Полиморфизм, 252
подтипа, 252
Полное время выполнения, 38, 42
Полный КА, 66
Положительное замыкание, 60
Полурешетка, 513
Получатель, 252
Последовательность вызова, 296
Постдоминирование, 544
Построение
подмножеств, 72, 98
трассы, 665, 748, 753
По строкам (размещение массива), 275
Потоконезависимость, 486
Правая рекурсия, 123
Предварительная последовательность, 325
Предикатное выполнение, 372
Предложение, 110
выбора, 381
Предсказуемость, 485
Предсказуемые выражения, 477, 485, 551, 554
Преобразование в булево значение, 267
Приводимый граф, 526
Приемка, действие, 143, 155, 233
Проверка диапазона, 366
Программный конвейер, 670
Продукция, 110
Пролог, 325
Промежуточное представление (ПП), 29, 176, 783
определяющее, 181
производное, 181
реализация, 801
Простой блок, 187, 198, 410
Прямое вычисление адреса, 382
Псевдоинтерференция, 691
Псевдоним, 315
Псевдооперация, 200

Р

Разбиение множества, 76
Разбор синтаксический, 105
восходящий парсер, 118
генератор парсеров, 134

нисходящий парсер, 118
рекурсивного спуска, 132
LL(1)-грамматика, 134
LL(1)-парсер, 133
LR(1)-парсер, 140
Разворачивание со слиянием, 435
Разделение мультимножества, 208
Разделители, 83
Раскрашивание графа, 687
Распознаватель, 50
Распределение регистров, 39.
См. также Распределитель регистров
восстановление, 702
выгрузка, 702
модели памяти, 215
хроматическое число, 707
Распределитель, 207
на основе линейного сканирования, 749
первый подходящий, 330
регистров, 210, 218, 221, 354, 357, 405, 415, 453, 506, 558, 562, 574, 597, 618, 648, 650, 658, 684, 748, 795, 797
с несколькими пулами, 332
Распространение констант
разреженный простой алгоритм, 513
разреженный условный алгоритм, 574
Расширенный простой блок (РПБ), 411, 431, 663
древовидная структура, 432
нетривиальный, 431
путь, 431, 566
Расщепление нулевой стоимости, 726
Ребро
замыкающее цикл, 570
сериализации, 654
Реверсивный обратный порядок (РОП), 474, 477
Региональная константа, 581
Регистр
общего назначения (РОН), 693, 695
сохраняемый вызываемой стороной, 305
сохраняемый вызывающей стороной, 305
Регистры
классы, 693
прочистка, 725
спрос на, 354, 691
Регулярные выражения (РВ), 58, 62, 64, 108, 740, 750, 780
ациклические, 206
конечное замыкание, 60
определение, 60
поиск, 742
положительное замыкание, 60
примеры, 61
приоритет, 61
свободные от замыканий, 97, 223
сопоставление с образцом, 62
Регулярный язык, 58, 64, 112
Режимы адресации, 601
Рекурренция, 674
Реляционное выражение, 368

С

Сборка мусора, 334, 335
копирующие сборщики, 337
поколенческие сборщики, 337
сборщики с пометкой и очисткой, 337
сборщики типа «остановка и копирование», 337
Сборщик типа «остановка и копирование», 337
Свертка операций копирования, 506
Сверхдлинное командное слово (VLIW)
компилятор для ЦПОС, 677
процессор, 645
Свободная переменная, 249
Своевременный (JIT) компилятор, 25, 737
Свойство конечной убывающей цепочки, 479
Связи доступа, 320
использование, 321
сравнение с глобальным дисплеем, 322
цепочка, 323
Сдвинутый нуль массива, 361
Сентенциальная форма, 111
Сигнатура типа, 260
Синтаксис, 32
Синтаксическая категория, 49
Синтаксически управляемая трансляция, 227
Синтаксическое дерево, 112

Система восходящего переписывания (BURS), 617, 630, 635
Скалярная оптимизация, 538
Сканер, 33, 49, 104
  с максимальным совпадением, 87
Скачковая функция, 522
Слоты запуска, 639
Смещенное раскрашивание, 724
Снижение стоимости операций, 402, 580
Соглашение
  об именовании, 785
  о связях, 296, 324, 388
    заключительная последовательность, 326
    предварительная последовательность, 325
    пролог, 325
    эпилог, 326
Состояние ошибки, 53
Спагетти, 90
Статическая диспетчеризация, 311
Статическая координата, 247
Статическая переменная, 220
Статическая привязка, 245
Статическая связь, 320
Статический анализ, 469
Статическое выделение памяти, 307
Статическое расстояние, 321
Степенное множество, 69
Страница, 274
Строгое доминирование, 492
Строка ссылок, 703
Суперкласс, 203, 252
Суперлокальная нумерация значений, 431
Суперскалярный процессор, 644

Т

Таблица символов, 82, 201
Теневая переменная цикла, 379
Терминальный символ, 110
Тип, 258, 285
  перечисления, 263
Токен, 50
Томпсона построение, 70
Точка соединения, 469
Трехфазный компилятор, 30

У

Указатель на запись активации (УЗА), 271, 303, 350, 354, 561, 606, 607, 620, 693, 702, 722
Универсальная хеш-функция, 810
Уничтоженное значение, 420
Упорядоченные списки, 796
Управляющая последовательность, 62
Упреждающий (AOT) компилятор, 25, 737, 743, 761
Условный переход, 37, 193, 243, 370
  неоднозначный, 192
  относительный, 200
  цели, 198
Устранение косвенной левой рекурсии, 124

Ф

Фаза
  анализа, 28, 36
  синтеза, 28, 39, 45
    генератор, 599
Фактический параметр, 191, 268, 297, 313, 358
Физический регистр, 41, 686
Формальный параметр, 297, 313
Форма с единственным статическим присваиванием (SSA), 208, 212, 488
  вставка операции копирования, 505
  вставка φ-функций, 495
  имена, 490
  максимальная SSA-форма, 491
  переименование переменных, 499
  полуусеченная SSA-форма, 491
Функция-батут, 279

Х

Хвостовой вызов, 380, 560, 562
Хеш-таблица
  альтернативы, 208
  коллизия, 206
  метод открытой адресации, 812
  метод цепочек, 810
  реализация, 809
  роль, 800
Хеш-функция, 809
Хопкрофта алгоритм, 76, 83, 100, 104

Хордальный граф, 728
Хроматическое число, 707

Ц

Целесообразность, 400
Цикл
  инвариант, 38
  разворачивание , 403
  слияние, 435
  ядро, 670
Циклически порожденная зависимость, 674

Ч

Частичная избыточность, 551
Частично мертвая операция, 759
Чистая выгрузка, 725
Чистое значение, 692

Э

Экспонирование сверху, 424
Элемент массива, 217, 275, 363, 421, 727
  адресное выражение, 403
  ссылка, 131, 164
Эпилог, 326

Книги издательства «ДМК Пресс»
можно купить оптом и в розницу на складе издательства по адресу:
Москва, ул. Электродная, д. 2, стр. 12, офис 7, тел. **+7 (499) 322-19-38**,
а также заказать на сайте **www.dmkpress.com**
с доставкой в любой регион РФ

Кит Д. Купер, Линда Торкзон

Создание компиляторов: инженерный подход

Главный редактор *Яценков В. С.*
editor@dmkpress.com
Перевод *Слинкин А. А.*
Корректор *Синяева Г. И.*
Верстка *Чаннова А. А.*
Дизайн обложки *Мовчан А. Г.*

Гарнитура PT Serif. Печать цифровая.
Усл. печ. л. 69,39. Тираж 200 экз.

Веб-сайт издательства: **www.dmkpress.com**

Перед вами перевод культовой книги Engineering a Compiler. Она фундаментально и всеобъемлюще раскрывает сложный процесс создания компиляторов. Вы поймете, как работают «под капотом» языки программирования, преобразуя человекочитаемый код в машинный.

В новое издание добавлены главы, посвященные семантической детализации, поддержке именования и адресуемости на этапе выполнения и форме кода для выражений присваивания и управляющих конструкций.

Основные особенности:

- углубленное изучение алгоритмов и методов, применяемых в фазе анализа современного компилятора;
- упор на оптимизацию и генерирование кода, основные области недавних исследований и разработок;
- новые главы, посвященные тому, как компиляторы (и интерпретаторы) реализуют абстракции.

Издание рекомендуется не только разработчикам, но и инженерам, которые хотят глубоко понять оптимизацию, анализ кода и другие этапы компиляции. Также книга будет полезна студентам и преподавателям.



Кит Д. Купер – профессор вычислительной техники Университета Райса. В разные годы занимал должности заведующего кафедрой информатики, заведующего кафедрой вычислительной и прикладной математики, а также заместителя декана по исследованиям инженерной школы Университета Райса. Является членом Ассоциации вычислительной техники (ACM).



Линда Торкзон длительное время работала старшим научным сотрудником кафедры информатики Университета Райса, затем занимала различные руководящие должности в сфере исследований и разработки новых технологий. Была главным исследователем проекта по созданию адаптируемых сред компиляции, финансируемого агентством DARPA.

ДМК ПРЕСС издательство
www.дмк.рф

ISBN 978-5-93700-428-4
9 785937 004284 >