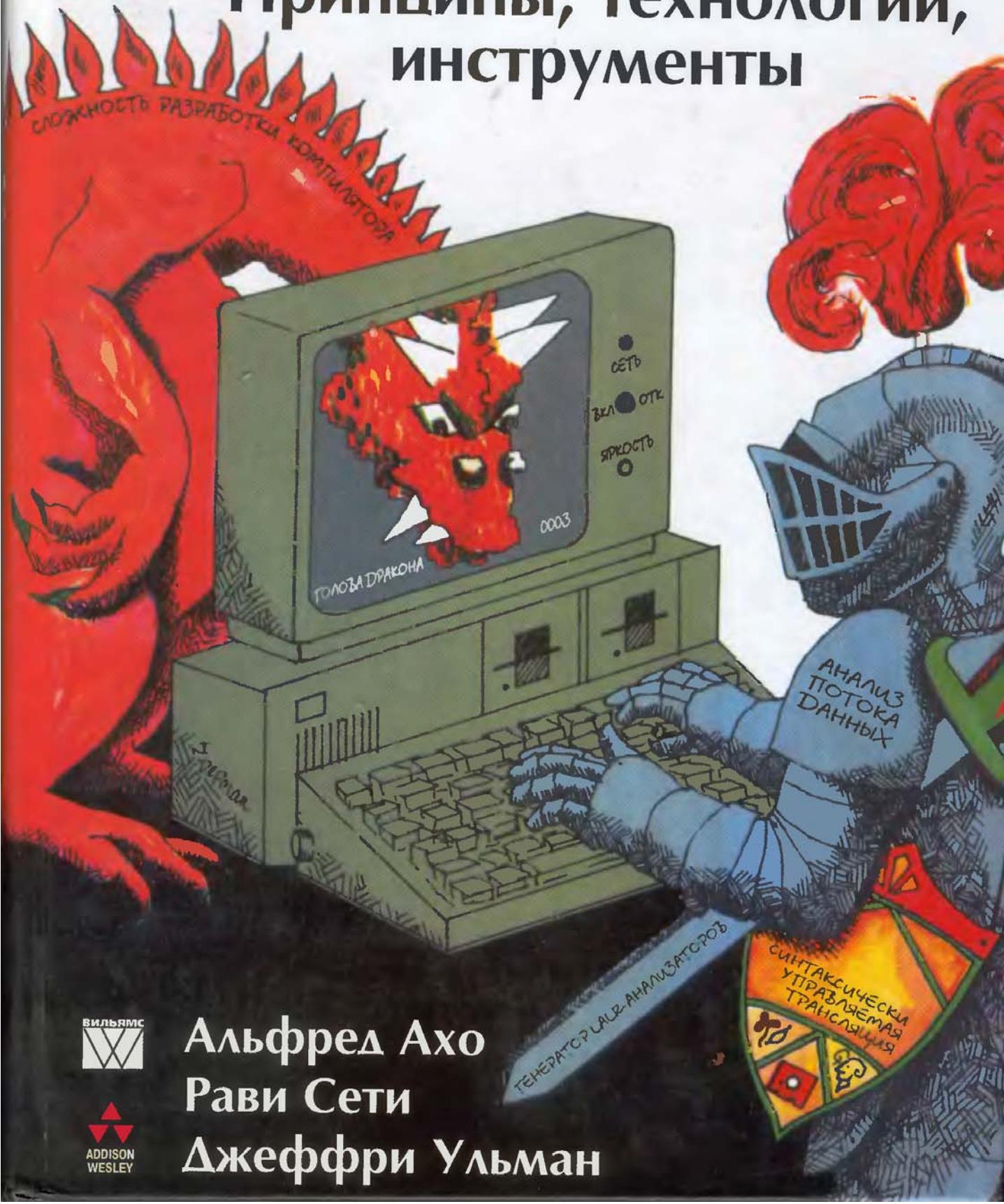


Компиляторы

Принципы, технологии, инструменты



ADDISON
WESLEY

Альфред Ахо
Рави Сети
Джеффри Ульман

Compilers

Principles, Techniques, and Tools

ALFRED V. AHO
*AT&T Bell Laboratories
Murray Hill, New Jersey*

RAVI SETHI
*AT&T Bell Laboratories
Murray Hill, New Jersey*

JEFFREY D. ULLMAN
*Stanford University
Stanford, California*



ADDISON-WESLEY PUBLISHING COMPANY

*Reading, Massachusetts • Menlo Park, California
Don Mills, Ontario • Wokingham, England • Amsterdam • Sydney
Singapore • Tokyo • Mexico City • Bogotá • Santiago • San Juan*

Компиляторы

Принципы, технологии, инструменты

АЛЬФРЕД АХО
AT&T Bell Laboratories
Мюррей Хилл, штат Нью-Джерси

РАВИ СЕТИ
AT&T Bell Laboratories
Мюррей Хилл, штат Нью-Джерси

ДЖЕФФРИ УЛЬМАН
Станфордский университет
Станфорд, штат Калифорния



Москва • Санкт-Петербург • Киев
2003

ББК 32.973.26-018.2.75

A95

УДК 681.3.07

Издательский дом “Вильямс”

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

Под редакцией канд.техн.наук *И.В. Красикова* и канд.физ.-мат.наук *А.Б. Ставровского*

Под общей редакцией канд.техн.наук *И.В. Красикова*

По общим вопросам обращайтесь в Издательский дом “Вильямс”
по адресу: info@williamspublishing.com, <http://www.williamspublishing.com>

Ахо, Альфред, В., Сети, Рави, Ульман, Джейфри, Д.

A95 Компиляторы: принципы, технологии и инструменты. : Пер. с англ. — М. : Издательский дом “Вильямс”, 2003. — 768 с. : ил. — Парал. тит. англ.

ISBN 5-8459-0189-8 (рус.)

Каждый, кто интересовался разработкой компиляторов, несомненно, слышал о знаменитой “Книге Дракона” — “Dragon Book”, классическом труде Ахо и Ульмана “Принципы разработки компиляторов”. Бурное развитие технологий компиляции привело к рождению нового дракона — книги “Компиляторы: принципы, технологии, инструментарий” Альфреда Ахо, Рави Сети и Джейфри Ульмана.

Новая книга начинается с изложения принципов создания компиляторов, проиллюстрированного разработкой простейшего однопроходного компилятора. Оставшаяся часть книги посвящена развитию базовых идей и более прогрессивным и современным технологиям, включая такие вопросы, как синтаксический анализ, проверка типов, генерация и оптимизация кода.

Строгость изложения материала смягчается большим количеством практических примеров. Написание компиляторов охватывает языки программирования, архитектуру вычислительных систем, теорию языков, алгоритмы и технологию создания программного обеспечения. Помочь в освоении этих технологий и инструментария и призвана данная книга. Несмотря на учебную ориентацию, книга будет полезна всем, кто работает над созданием компиляторов или просто интересуется данной темой.

ББК 32.973.26-018.2.75

Все названия программных продуктов являются зарегистрированными торговыми марками соответствующих фирм.

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

Authorized translation from the English language edition published by Addison-Wesley Publishing Company, Inc., Copyright © 1985

All rights reserved. No part of this book may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording or by any information storage retrieval system, without permission from the Publisher.

Russian language edition published by Williams Publishing House according to the Agreement with R&I Enterprises International, Copyright © 2001

Издательство выражает признательность Дмитрию Владленовичу Виленскому за информационную помощь и поддержку.

ISBN5-8459-0189-8(рус.)
ISBN0-201-10088-6(англ.)

© Издательский дом “Вильямс”, 2001
© Addison-Wesley Publishing Company, Inc., 1985

Оглавление

ГЛАВА 1. Введение в компиляцию	22
ГЛАВА 2. Простой однопроходный компилятор	44
ГЛАВА 3. Лексический анализ	97
ГЛАВА 4. Синтаксический анализ	167
ГЛАВА 5. Синтаксически управляемая трансляция	279
ГЛАВА 6. Проверка типов	336
ГЛАВА 7. Среды времени исполнения	376
ГЛАВА 8. Генерация промежуточного кода	443
ГЛАВА 9. Генерация кода	487
ГЛАВА 10. Оптимизация кода	553
ГЛАВА 11. Создание компилятора	679
ГЛАВА 12. Некоторые компиляторы	688
ПРИЛОЖЕНИЕ А. Программный проект	698
ПРИЛОЖЕНИЕ Б. Спецификации языков программирования	704
БИБЛИОГРАФИЯ	742

Содержание

От редактора	18
Предисловие	19
Использование данной книги	19
Упражнения	20
Благодарности	21
ГЛАВА 1. Введение в компиляцию	22
1.1. Компиляторы	22
Модель анализа-синтеза компиляции	23
Контекст компилятора	25
1.2. Анализ исходной программы	25
Лексический анализ	26
Синтаксический анализ	26
Семантический анализ	28
Анализ в программах форматирования текста	29
1.3. Фазы компилятора	30
Управление таблицей символов	30
Обнаружение ошибок и сообщение о них	31
Фазы анализа	33
Генерация промежуточного кода	34
Оптимизация кода	34
Генерация кода	35
1.4. Родственники компилятора	35
Препроцессоры	35
Ассемблеры	37
Двухпроходный ассемблер	37
Загрузчики и редакторы связей	38
1.5. Группировка фаз	39
Предварительная и заключительная стадии	39
Проходы	40
Уменьшение количества проходов	40
1.6. Инструментарий для создания компиляторов	41
Библиографические примечания	43
ГЛАВА 2. Простой однопроходный компилятор	44
2.1. Обзор	44
2.2. Определение синтаксиса	45
Деревья разбора	47
Неоднозначность	48
Ассоциативность операторов	49
Приоритет операторов	50

2.3. Синтаксически управляемая трансляция	51
Постфиксная запись	51
Синтаксически управляемые определения	52
Синтезируемые атрибуты	52
Рекурсивный обход дерева	55
Схемы трансляции	55
Вывод результатов трансляции	56
2.4. Разбор	58
Нисходящий анализ	58
Предиктивный анализ	61
Использование ϵ -продукций	63
Создание предиктивного анализатора	63
Левая рекурсия	64
2.5. Транслятор для простых выражений	65
Абстрактный и конкретный синтаксис	65
Адаптация схемы трансляции	67
Процедуры для нетерминалов <i>expr</i> , <i>term</i> и <i>rest</i>	67
Оптимизация транслятора	69
Полная программа	70
2.6. Лексический анализ	71
Удаление пробелов и комментариев	71
Константы	72
Распознавание идентификаторов и ключевых слов	72
Интерфейс к лексическому анализатору	73
Лексический анализатор	74
2.7. Использование таблиц символов	76
Интерфейс таблицы символов	76
Обработка зарезервированных ключевых слов	77
Реализация таблицы символов	77
2.8. Абстрактная стековая машина	79
Арифметические инструкции	79
l-значения и r-значения	80
Работа со стеком	80
Трансляция выражений	80
Управление выполнением	81
Трансляция инструкций	82
Вывод результатов трансляции	82
2.9. Сборка транслятора	84
Описание транслятора	84
Модуль лексического анализа <i>lexer.c</i>	85
Модуль синтаксического анализатора <i>parser.c</i>	86
Модуль вывода <i>emitter.c</i>	87
Модули работы с таблицей символов <i>symbol.c</i> и <i>init.c</i>	87
Модуль обработки ошибок <i>error.c</i>	87
Создание компилятора	87
Листинг	88
Упражнения	92

Программные упражнения	95
Библиографические примечания	96
ГЛАВА 3. Лексический анализ	97
3.1. Роль лексических анализаторов	97
Задачи лексического анализа	98
Токены, шаблоны, лексемы	99
Атрибуты токенов	100
Лексические ошибки	101
3.2. Буферизация ввода	102
Пары буферов	102
Ограничители	104
3.3. Определение токенов	105
Строки и языки	105
Операции над языками	106
Регулярные выражения	107
Регулярные определения	109
Сокращения	110
Нерегулярные множества	110
3.4. Распознавание токенов	111
Диаграммы переходов	112
Реализация диаграммы переходов	116
3.5. Язык спецификации лексических анализаторов	119
Спецификации Lex	119
Прогностический оператор	123
3.6. Конечные автоматы	125
Недетерминированные конечные автоматы	125
Детерминированный конечный автомат	127
Преобразование НКА в ДКА	128
3.7. От регулярного выражения к НКА	132
Построение НКА по регулярному выражению	132
Двустековое моделирование НКА	136
Скорость работы	137
3.8. Построение генератора лексических анализаторов	139
Поиск по шаблону на основе НКА	140
ДКА для лексического анализа	142
Реализация прогностического оператора	143
3.9. Оптимизация поиска соответствия шаблону на базе ДКА	144
Важные состояния в НКА	144
От регулярного выражения к ДКА	145
Минимизация количества состояний ДКА	150
Минимизация состояний в лексических анализаторах	153
Методы сжатия таблиц	153
Упражнения	155
Программные упражнения	165
Библиографические примечания	165

ГЛАВА 4. Синтаксический анализ

167

4.1. Роль синтаксического анализатора	167
Обработка синтаксических ошибок	168
Стратегии восстановления после ошибок	172
4.2. Контекстно-свободные грамматики	173
Соглашения по обозначениям	174
Порождение	175
Деревья разбора и приведения	177
Неоднозначность	179
4.3. Разработка грамматики	179
Регулярные выражения и контекстно-свободные грамматики	180
Проверка языка, порожденного грамматикой	181
Устранение неоднозначности	181
Устранение левой рекурсии	183
Левая факторизация	185
Построение не-контекстно-свободных грамматик	186
4.4. Нисходящий анализ	188
Анализ методом рекурсивного спуска	188
Предиктивные анализаторы	189
Диаграммы переходов предиктивных синтаксических анализаторов	190
Нерекурсивный предиктивный анализ	192
FIRST и FOLLOW	195
Построение таблиц предиктивного анализа	197
LL(1)-грамматики	197
Восстановление после ошибок в предиктивном анализе	199
4.5. Восходящий синтаксический анализ	201
Основы	202
Обрезка основ	204
Стековая реализация ПС-анализа	205
Активные префиксы	207
Конфликты в процессе ПС-анализа	207
4.6. Синтаксический анализ приоритета операторов	209
Использование отношений приоритетов операторов	211
Нахождение отношений приоритетов операторов	213
Обработка унарных операторов	214
Функции приоритета	214
Восстановление после ошибок при синтаксическом анализе приоритета операторов	216
4.7. LR-анализаторы	220
Алгоритм LR-анализа	221
LR-грамматики	225
Построение таблиц SLR-анализа	226
Построение канонических таблиц LR-анализа	234
Построение таблиц LARL-анализа	240
Эффективное построение таблиц LALR-анализа	244
4.8. Использование неоднозначных грамматик	250
Использование приоритетов и ассоциативности для разрешения конфликтов	251

Неоднозначность “кочующего” else	253
Неоднозначности особых продуктов частных случаев	255
Восстановление после ошибок при LR-анализе	257
4.9. Генераторы синтаксических анализаторов	259
Генератор синтаксических анализаторов Yacc	260
Использование Yacc с неоднозначной грамматикой	263
Создание лексического анализатора в Yacc с помощью Lex	265
Восстановление после ошибок в Yacc	266
Упражнения	268
Библиографические примечания	277
ГЛАВА 5. Синтаксически управляемая трансляция	279
5.1. Синтаксически управляемые определения	280
Вид синтаксически управляемого определения	280
Синтезируемые атрибуты	281
Наследуемые атрибуты	282
Графы зависимости	283
Порядок выполнения	285
5.2. Построение синтаксических деревьев	286
Синтаксические деревья	286
Построение синтаксических деревьев для выражений	287
Синтаксически управляемое определение для построения синтаксических деревьев	288
Направленные ациклические графы выражений	289
5.3. Восходящее выполнение S-атрибутных определений	292
Синтезируемые атрибуты в стеке синтаксического анализатора	292
5.4. L-атрибутные определения	295
L-атрибутные определения	295
Схемы трансляции	296
5.5. Нисходящая трансляция	299
Устранение левой рекурсии из схемы трансляции	299
Разработка предиктивного транслятора	303
5.6. Восходящее вычисление наследуемых атрибутов	305
Удаление внедренных действий из схемы трансляции	305
Наследование атрибутов в стеке синтаксического анализатора	306
Моделирование вычисления наследуемых атрибутов	308
Замена наследуемых атрибутов синтезируемыми	312
Сложное синтаксически управляемое определение	312
5.7. Рекурсивные вычислители	312
Обход слева направо	313
Другие обходы	314
5.8. Память для значений атрибутов во время компиляции	315
Назначение памяти атрибутам во время компиляции	317
Устранение копий	318
5.9. Назначение памяти в процессе создания компилятора	319
Вычисление времени жизни по грамматике	319
Неперекрывающиеся времена жизни	323

5.10. Анализ синтаксически управляемых определений	324
Рекурсивное вычисление атрибутов	325
Строго нециклические синтаксически управляемые определения	327
Проверка цикличности	328
Упражнения	330
Библиографические примечания	334
ГЛАВА 6. Проверка типов	336
6.1. Системы типов	337
Выражения типов	338
Системы типов	339
Статическая и динамическая проверка типов	340
Восстановление после ошибки	340
6.2. Спецификация простой программы проверки типов	341
Простой язык	341
Проверка типов выражений	342
Проверка типов инструкций	343
Проверка типов функций	343
6.3. Эквивалентность выражений типа	344
Структурная эквивалентность выражений типа	344
Имена выражений типа	347
Циклы в представлениях типов	349
6.4. Преобразования типов	350
Неявное преобразование типов	350
6.5. Перегрузка функций и операторов	352
Множество возможных типов подвыражения	352
Сужение множества возможных типов	354
6.6. Полиморфные функции	355
Почему используются полиморфные функции	355
Переменные типа	356
Язык с полиморфными функциями	358
Подстановки, примеры и унификация	360
Проверка полиморфных функций	361
6.7. Алгоритм унификации	365
Упражнения	369
Библиографические примечания	374
ГЛАВА 7. Среды времени исполнения	376
7.1. Вопросы исходного языка	376
Процедуры	376
Деревья активации	377
Стеки управления	379
Область видимости объявления	380
Организация памяти и связывание имен	382
7.2. Организация памяти	382
Классификация памяти времени выполнения	382
Записи активаций	384

Размещение локальных данных в процессе компиляции	385
7.3. Стратегии выделения памяти	386
Статическое распределение памяти	387
Стековое распределение памяти	389
Висячие ссылки	394
Распределение памяти в куче	395
7.4. Доступ к нелокальным именам	396
Блоки	396
Лексическая область видимости без вложенных процедур	398
Лексическая область видимости при наличии вложенных процедур	400
Дисплеи	403
Динамическая область видимости	406
7.5. Передача параметров	407
Передача по значению	408
Передача по ссылке	409
Копирование-восстановление	410
Передача по имени	411
7.6. Таблицы символов	412
Записи таблицы символов	413
Символы имени	414
Информация о выделенной памяти	415
Использование списков для представления таблицы символов	415
Хеш-таблицы	416
Представление информации об области видимости	420
7.7. Возможности языков по динамическому выделению памяти	422
Мусор	424
Висячие ссылки	424
7.8. Технологии динамического распределения памяти	425
Явное выделение блоков фиксированного размера	425
Явное выделение блоков переменного размера	426
Неявное освобождение	426
7.9. Распределение памяти в Fortran	428
Данные в областях COMMON	429
Простой алгоритм эквивалентности	430
Алгоритм эквивалентности для языка программирования Fortran	433
Отображение областей данных	435
Упражнения	436
Библиографические примечания	442
ГЛАВА 8. Генерация промежуточного кода	443
8.1. Языки промежуточных представлений	443
Графическое представление	444
Трехадресный код	445
Типы трехадресных инструкций	446
Синтаксически управляемая трансляция в трехадресный код	447
Реализация трехадресных инструкций	449
Сравнение представлений: использование косвенного обращения	451

8.2. Объявления	452
Объявления в процедуре	452
Отслеживание информации об области видимости	453
Имена полей в записях	455
8.3. Инструкции присвоения	456
Имена в таблице символов	456
Повторное использование временных имен	458
Адресация элементов массива	459
Схема трансляции для адресации элементов массива	461
Преобразования типов в присвоениях	463
Доступ к полям записей	465
8.4. Логические выражения	465
Методы трансляции логических выражений	466
Числовое представление	466
Сокращенные вычисления	467
Инструкции потока управления	468
Трансляция логических выражений с помощью потока управления	470
Смешанные логические выражения	471
8.5. Инструкции case	473
Синтаксически управляемая трансляция инструкции case	474
8.6. Технология обратных поправок	476
Логические выражения	476
Инструкции потока управления	479
Схема реализации трансляции	479
Метки и безусловные переходы	481
8.7. Вызовы процедур	481
Вызывающие последовательности	481
Простой пример	482
Упражнения	483
Библиографические примечания	485
ГЛАВА 9. Генерация кода	487
9.1. Вопросы создания генератора кода	487
Вход генератора кода	487
Целевые программы	488
Управление памятью	489
Выбор инструкций	489
Распределение регистров	490
Выбор порядка вычислений	491
Подходы к генерации кода	491
9.2. Целевая машина	492
Стоимость инструкций	493
9.3. Управление памятью во время исполнения	494
Статическое распределение	495
Стековое распределение	497
Адресация имен во время исполнения	499
9.4. Базовые блоки и графы потоков	500

Базовые блоки	500
Преобразования в базовых блоках	502
Преобразования, сохраняющие структуру	502
Алгебраические преобразования	503
Графы потоков	503
Представление базовых блоков	504
Циклы	505
9.5. Информация о последующем использовании	505
Вычисление последующих использований	505
Память для временных имен	506
9.6. Простой генератор кода	507
Дескрипторы регистров и адресов	508
Алгоритм генерации кода	508
Функция <i>getreg</i>	509
Генерация кода для других типов инструкций	511
Условные инструкции	512
9.7. Распределение и назначение регистров	512
Глобальное распределение регистров	513
Счетчики использований	513
Назначение регистров для внешних циклов	516
Распределение регистров путем раскраски графа	516
9.8. Представление базовых блоков в виде дага	517
Построение дага	518
Применение дагов	520
Массивы, указатели и вызовы процедур	522
9.9. Локальная оптимизация	524
Излишние загрузки и сохранения	524
Недостижимый код	525
Оптимизация потока управления	526
Алгебраические упрощения	527
Снижение стоимости	527
Использование машинных идиом	527
9.10. Генерация кода на основе дагов	527
Переупорядочение	528
Эвристическое упорядочение дагов	529
Оптимальное упорядочение для деревьев	530
Алгоритм назначения меток	531
Генерация кода на основе помеченного дерева	532
Многорегистровые операции	535
Алгебраические свойства	535
Общие подвыражения	536
9.11. Алгоритм динамического программирования для генерации кода	537
Класс регистровых машин	538
Принцип динамического программирования	538
Последовательное вычисление	538
Алгоритм динамического программирования	539
9.12. Генераторы генераторов кода	541

Генерация кода путем преобразования дерева	542
Проверка соответствия шаблону путем синтаксического анализа	547
Программы семантической проверки	548
Упражнения	548
Библиографические примечания	551
ГЛАВА 10. Оптимизация кода	553
10.1. Введение	554
Критерии для преобразований, улучшающих код	554
Повышение производительности	555
Организация оптимизирующего компилятора	556
10.2. Основные источники оптимизации	559
Преобразования, сохраняющие функции	559
Общие подвыражения	559
Распространение копий	561
Удаление бесполезного кода	562
Оптимизации циклов	562
Перемещение кода	563
Переменные индукции и снижение стоимости	563
10.3. Оптимизация базовых блоков	565
Использование алгебраических тождеств	567
10.4. Циклы в графах потока	568
Доминаторы	568
Естественные циклы	570
Внутренние циклы	571
Предзаголовки	571
Приводимые графы потоков	572
10.5. Введение в глобальный анализ потоков данных	574
Точки и пути	575
Достигающие определения	575
Анализ потока данных структурированной программы	577
Консервативная оценка информации потока данных	579
Вычисление <i>in</i> и <i>out</i>	580
Работа с циклами	581
Представление множеств	583
Локальные достигающие определения	584
Цепочки определений использований	585
Порядок вычислений	585
Общий поток управления	586
10.6. Итеративное решение уравнений потока данных	588
Итеративный алгоритм для достигающих определений	588
Доступные выражения	591
Анализ активных переменных	594
Цепочки использований определений	596
10.7. Преобразования, улучшающие код	596
Устранение глобальных общих подвыражений	596
Распространение копирований	598

Поиск вычислений, инвариантных относительно цикла	601
Выполнение перемещения кода	602
Альтернативные стратегии перемещения кода	604
Поддержание информации о потоке данных после перемещения кода	605
Устранение переменных индукции	605
Переменные индукции и выражения, инвариантные относительно цикла	610
10.8. Работа с альтернативными именами	610
Простой язык указателей	611
Воздействие присвоений указателей	611
Использование информации об указателях	614
Анализ межпроцедурного потока данных	615
Модель кода с вызовами процедур	616
Вычисление псевдонимов	617
Анализ потока данных при наличии вызовов процедур	618
Использование информации об изменениях	621
10.9. Анализ потока данных структурированных графов потока	622
Поиск вглубь	622
Дуги представления графа потока вглубь	625
Глубина графа потока	626
Интервалы	626
Разбиение на интервалы	627
Графы интервалов	628
Разделение узлов	629
T_1-T_2 -анализ	629
Области	630
Поиск доминаторов	631
10.10. Эффективные алгоритмы потоков данных	633
Упорядочение вглубь в итеративных алгоритмах	633
Анализ потока данных, основанный на структуре	635
Некоторые ускорения структурного алгоритма	639
Обработка неприводимых графов потока	639
10.11. Средства для анализа потока данных	640
Схема анализа потока данных	641
Аксиомы схем анализа потока данных	643
Монотонность и дистрибутивность	644
Решения типа “слияние всех путей” в задачах о потоках данных	648
Консервативные решения задач о потоках	649
Итеративный алгоритм для обобщенных схем	650
Инструмент анализа потока данных	650
Свойства алгоритма 10.18	651
Сходимость алгоритма 10.18	652
Исправление инициализации	653
10.12. Оценка типов	653
Работа с бесконечными множествами типов	654
Простая система типов	656
Прямая схема	657
Обратная схема	658

10.13. Символьная отладка оптимизированного кода	661
Отслеживание значений переменных в базовых блоках	662
Влияние глобальной оптимизации	667
Устранение переменных индукции	667
Устранение глобальных общих подвыражений	667
Перемещение кода	668
Упражнения	669
Библиографические примечания	675
ГЛАВА 11. Создание компилятора	679
11.1. Планирование компилятора	679
Вопросы исходного языка	679
Вопросы целевого языка	680
Критерии производительности	680
11.2. Подходы к разработке компилятора	680
Раскрутка	681
11.3. Среда разработки компилятора	684
11.4. Тестирование и сопровождение	686
ГЛАВА 12. Некоторые компиляторы	688
12.1. Препроцессор математических формул EQN	688
12.2. Компиляторы Pascal	689
12.3. Компиляторы C	690
12.4. Компиляторы Fortran H	691
Оптимизация кода в Fortran H	693
Алгебраическая оптимизация	693
Оптимизация регистров	694
12.5. Компилятор Bliss/11	694
12.6. Оптимизирующий компилятор Modula-2	696
ПРИЛОЖЕНИЕ А. Программный проект	698
A.1. Введение	698
A.2. Структура программы	698
A.3. Синтаксис подмножества Pascal	699
A.4. Лексические соглашения	701
A.5. Предлагаемые упражнения	701
A.6. Эволюция интерпретатора	703
A.7. Расширения	703
ПРИЛОЖЕНИЕ Б. Спецификации языков программирования	704
Б.1. Язык программирования C++	704
Б.2. Язык программирования C#	721
БИБЛИОГРАФИЯ	742
Предметный указатель	764

От редактора

Перед вами перевод известной книги знаменитых авторов, перу которых принадлежит ряд бестселлеров в области компьютерных наук, причем диапазон их плодотворной работы чрезвычайно широк — от фундаментальных научных статей до великолепных учебников для студентов. Именно таким учебником и является данная книга, которая доступна широкому кругу программистов и представляет собой завершенное изложение технологий построения компиляторов.

Многие работы этих авторов переведены на русский язык и неизменно пользуются повышенным спросом у серьезного читателя. Достаточно вспомнить такие книги, как *Построение и анализ вычислительных алгоритмов* [7], *Структуры данных и алгоритмы* [8] или *Теория синтаксического анализа, перевода и компиляции* [18–19].

Предлагаемая книга выгодно отличается от большинства своих предшественниц тем, что здесь охвачен весь процесс создания компилятора — от лексического анализа до генерации и оптимизации целевого кода. Использование абстрактных математических понятий доводится в ней до конкретных алгоритмов, пригодных для непосредственной реализации. Технологии, изложенные в этой книге, уже нашли свое применение не только в компиляции, но и в системах подготовки текстов, управления базами данных, генерации схем электронных устройств и других областях.

Оригинал этой книги опубликован в 1988 году. С тех пор были созданы новые парадигмы программирования, десятки новых языков и компиляторов для них. Однако осмелимся утверждать, что в их реализации не было ничего принципиально нового по сравнению с тем, что изложено в этой книге. Данное издание может служить практическим руководством по технологиям создания компиляторов. Описанные в ней технологии были созданы в основном еще в 60-е–80-е годы и существенно уже не изменятся. Книга по праву может считаться классической, потому что она не устарела и не устареет, пока люди будут создавать языки и системы программирования.

Предисловие

Данное издание является развитием книги Альфреда Ахо и Джейфри Ульмана *Principles of Computer Design*. Подобно своей предшественнице, она задумана как начальный учебный курс по проектированию компиляторов. Основное внимание уделяется решению общих проблем, возникающих при создании трансляторов языков программирования, независимо от исходного языка или целевой машины.

Вероятно, лишь немногие из вас будут заниматься построением или поддержкой компиляторов для основных языков программирования, однако идеи и технологии, описанные в ней, можно с успехом применять при разработке другого программного обеспечения. Например, технологии поиска соответствия строк шаблонам, используемые при построении лексических анализаторов, применяются и в текстовых редакторах, и в системах выборки информации, и в распознавании образов. Контекстно-свободные грамматики и синтаксически управляемые определения используются для создания многих небольших языков, например типографских систем для подготовки макетов книг. Технологии оптимизации кода применяются для проверки корректности программ и создания "структурированных" программ из неструктурированных.

Использование данной книги

В книге глубоко раскрыты основные аспекты создания компиляторов. Глава 1 описывает базовую структуру компилятора и представляет собой фундамент для остального материала книги.

В главе 2 рассматривается транслятор, переводящий инфиксные выражения в постфиксные. Он построен с использованием базовых технологий, описываемых в этой книге. В следующих главах развивается изложенный здесь материал.

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

Глава 4 пополняет наши знания о технологиях синтаксического анализа. Здесь рассматриваются как методы рекурсивного спуска, применяемые при ручной реализации трансляторов, так и более сложные и универсальные LR-технологии, используемые в генераторах синтаксических анализаторов.

Глава 5 знакомит нас с принципиальными идеями синтаксически управляемой трансляции. Материал этой главы используется в оставшейся части книги как для описания, так и реализации трансляторов.

В главе 6 выдвигаются основные идеи выполнения статической семантической проверки и детально рассматриваются вопросы проверки типов.

В главе 7 обсуждаются вопросы организации памяти для поддержки сред времени исполнения программ.

Глава 8 начинается с обсуждения языков промежуточного представления программ, а затем показывает нам, каким образом основные конструкции языка программирования могут транслироваться в промежуточный код.

Глава 9 посвящена генерации целевого кода, включая вопросы генерации кода "на лету" и оптимальные методы генерации кода для выражений. Здесь также рассмотрены локальная оптимизация и генераторы генераторов кода.

Глава 10 представляет исчерпывающую информацию об оптимизации кода. В ней детально изложены методы анализа потока данных, а также основные методы глобальной оптимизации.

В главе 11 обсуждается ряд практических вопросов, возникающих при реализации компиляторов, в частности важность вопросов разработки программного обеспечения и тестирования.

В главе 12 проведено исследование компиляторов, построенных с использованием представленных в книге технологий.

Приложение А описывает простой язык — “подмножество” Pascal, который может использоваться в качестве основы для реальных проектов.

На основе этой книги авторы преподавали как вводный, так и основной курсы для студентов и аспирантов AT&T Bell Laboratories, Колумбийского, Принстонского и Стэнфордского университетов.

Вводный курс, посвященный компиляторам, может основываться на материале следующих разделов книги.

Введение	Глава 1 и разделы 2.1–2.5
Лексический анализ	2.6, 3.1–3.4
Таблицы символов	2.7, 7.6
Синтаксический анализ	2.4, 4.1–4.4
Синтаксически управляемая трансляция	2.5, 5.1–5.5
Проверка типов	6.1, 6.2
Организация времени исполнения	7.1–7.3
Генерация промежуточного кода	8.1–8.3
Генерация кода	9.1–9.4
Оптимизация кода	10.1, 10.2

Информация, необходимая для программного проекта, подобного приведенному в приложении А, имеется в главе 2.

Курс, посвященный инструментарию построения компиляторов, может включать обсуждение генераторов лексических анализаторов из раздела 3.5, генераторов синтаксических анализаторов из разделов 4.8 и 4.9, генераторов генераторов кода из раздела 9.12, а также материал о технологиях построения компиляторов из главы 11.

Основной курс может делать упор на алгоритмы, используемые в генераторах лексических и синтаксических анализаторов (главы 3 и 4); материал об эквивалентности типов, перегрузке, полиморфизме и унификации (глава 6), организации памяти времени исполнения (глава 7); методы генерации кода на основе шаблонов из главы 9; и материал об оптимизации кода из главы 10.

Упражнения

Как и ранее, сложность упражнений указана звездочками. Упражнения без звездочек проверяют понимание определений, упражнения с одной звездочкой относятся к более сложному курсу, а “двухзвездные” служат “пищей” для серьезных размышлений.

Благодарности

На различных этапах написания этой книги мы получили множество неоценимых комментариев и советов от большого числа специалистов. В связи с этим мы выражаем свою признательность Биллу Эппельбу (Bill Appelbe), Нельсону Бибу (Nelson Beebe), Джону Бентли (Jon Bentley), Луи Богесу (Lois Bogess), Родни Фэрроу (Rodney Farrow), Стю Фельдман (Stu Feldman), Чарльзу Фишеру (Charles Fischer), Крису Фрэзеру (Chris Fraser), Арту Життельману (Art Gittelman), Эрику Гроссе (Eric Grosse), Дэйву Хансону (Dave Hanson), Фрицу Хенглейну (Fritz Henglein), Роберту Генри (Robert Henry), Жерару Хольцману (Gerard Holzmann), Стиву Джонсону (Steve Johnson), Брайену Кернигану (Brian Kernighan), Кену Кубота (Ken Kubota), Дэниэлу Леману (Daniel Lehmann), Дэйву Мак-Квину (Dave MacQueen), Дианне Маки (Dianne Maki), Алану Мартину (Alan Martin), Дугу Мак-Илрою (Doug McIlroy), Чарльзу Мак-Лафлину (Charles McLaughlin), Джону Митчеллу (John Mitchell), Эллиоту Органику (Elliot Organick), Роберту Пейджу (Robert Paige), Филу Пфайфферу (Phil Pfeiffer), Робу Пайку (Rob Pike), Кари-Йоко Ряйхя (Kari-Jouko Räihä), Дэнису Ритчи (Dennis Ritchie), Срираму Санкар (Sriram Sankar), Полу Стокеру (Paul Stoecker), Бьорну Страуструпу (Bjarne Stroustrup), Тому Шимански (Tom Szymanski), Киму Трейси (Kim Tracy), Петеру Вайнбергеру (Peter Weinberger), Дженифер Видом (Jennifer Widom) и Рейнгарду Вильгельму (Reinhard Wilhelm).

Оригинал этой книги создан с использованием великолепного программного обеспечения, доступного в UNIX. Команда типографского набора выглядела следующим образом.

```
pic files | tbl | eqn | troff -ms
```

Pic — это язык Брайена Кернигана (Brian Kernighan) для набора рисунков; мы особо признательны Брайену за успешное удовлетворение наших обширных потребностей в иллюстрациях. tbl — это язык Майка Леска (Mike Lesk), предназначенный для макетирования таблиц; eqn — язык Брайена Кернигана (Brian Kernighan) и Лоринды Черри (Lorinda Cherry) для печати математических формул; troff — программа Джо Оссана (Joe Ossana) для форматирования текста при фотонаборе (в нашем случае — для Mergenthaler Linotron 202/N). Пакет макросов ms разработан Майком Леском (Mike Lesk). И наконец, с текстом мы работали с помощью make Стю Фельдман (Stu Feldman); перекрестные ссылки в тексте обрабатывались посредством awk, созданного Альфредом Ахо (Alfred Aho), Брайеном Керниганом (Brian Kernighan) и Петером Вайнбергером (Peter Weinberger), и sed Ли Мак-Махона (Lee McMahon).

Авторы должны отдельно поблагодарить Патрицию Соломон (Patricia Solomon) за помощь в подготовке книги для фотокомпозиции. Во время работы над книгой поддержку Джейфри Ульману (Jeffrey Ullman) оказало Эйнштейновское общество Академии искусств и науки Израиля. И наконец, авторы признательны AT&T Bell Laboratories за поддержку во время подготовки рукописи.

A.V.A., R.S., J.D.U.

ГЛАВА 1

Введение в компиляцию

Принципы и технологии написания компиляторов столь распространены, что идеи, описанные в этой книге, используются в самых разных областях информационных технологий. Написание компиляторов охватывает языки программирования, архитектуру вычислительных систем, теорию языков, алгоритмы и технологию создания программного обеспечения. К счастью, при создании трансляторов для широкого круга языков и машин достаточно лишь нескольких основных технологий написания компиляторов. В этой главе мы приступим к рассмотрению компонентов компиляторов, среди, в которой они работают, и программного инструментария, упрощающего их создание.

1.1. Компиляторы

Компилятор — это программа, которая считывает текст программы, написанной на одном языке — *исходном*, и транслирует (переводит) его в эквивалентный текст на другом языке — *целевом* (рис. 1.1). Одним из важных моментов трансляции является сообщение пользователю о наличии ошибок в исходной программе.



Рис. 1.1. Компилятор

На первый взгляд, разнообразие компиляторов потрясает. Используются тысячи исходных языков, от традиционных, таких как Fortran и Pascal, до специализированных, возникающих во всех областях компьютерных приложений. Целевые языки не менее разнообразны — это могут быть другие языки программирования, различные машинные языки — от языков микропроцессоров до суперкомпьютеров. Иногда компиляторы классифицируют как однопроходные, многопроходные, исполняющие (load-and-go), отлаживающие, оптимизирующие — в зависимости от предназначения и принципов и технологий их создания. Несмотря на кажущуюся сложность и разнообразие, основные задачи, выполняемые различными компиляторами, по сути, одни и те же. Понимая эти задачи, мы можем создавать компиляторы для различных исходных языков и целевых машин с использованием одинаковых базовых технологий.

Знания об организации и написании компиляторов существенно возросли со времен первых компиляторов, появившихся в начале 1950-х гг. Сегодня сложно определить, когда именно появился на свет первый компилятор, поскольку в те годы проводилось много

жество экспериментов и разработок различными независимыми группами. В основном целью этих разработок было преобразование в машинный код арифметических формул.

В 50-х годах о компиляторах ходила слава как о программах, крайне сложных в написании (например, первый компилятор Fortran потребовал 18 человеко-лет работы [41]). С тех пор разработаны разнообразные систематические технологии решения многих задач, возникающих при компиляции. Кроме того, разработаны хорошие языки реализации, программные среды и программный инструментарий. Благодаря этому “солидный” компилятор может быть реализован даже в качестве курсовой работы по проектированию компиляторов.

Модель анализа-синтеза компиляции

Компиляция состоит из двух частей: анализа и синтеза. Анализ — это разбиение исходной программы на составные части и создание ее промежуточного представления. Синтез — конструирование требуемой целевой программы из промежуточного представления. В разделе 1.2 мы неформально рассмотрим анализ, а способ синтеза целевого кода в стандартных компиляторах будет представлен в разделе 1.3.

В процессе анализа определяются и записываются в иерархическую древовидную структуру операции, заданные исходной программой. Часто используется специальный вид дерева, называемого синтаксическим (или деревом синтаксического разбора), в котором каждый узел представляет операцию, а его дочерние узлы — аргументы операции. Пример синтаксического дерева инструкции присвоения показан на рис. 1.2.

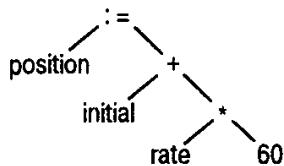


Рис. 1.2. Синтаксическое дерево для оператора `position:=initial+rate*60`

Многие программные инструменты, работающие с исходными программами, сначала выполняют определенный вид анализа. Рассмотрим примеры таких инструментов.

1. *Структурные редакторы.* Эти программы получают в качестве входа последовательность команд для построения исходной программы. Такой редактор не только выполняет обычные для текстового редактора функции по созданию и модификации текста, но и анализирует текст программы, помещая в исходную программу соответствующую иерархическую структуру. Тем самым он выполняет дополнительные задачи, облегчающие подготовку программы. Например, редактор может проверять корректность введенного текста, автоматически добавлять структурные элементы (так, если пользователь введет `while`, редактор добавит соответствующее ему ключевое слово `do` и предложит ввести условное выражение между ними) или переходить от ключевого слова `begin` или левой скобки к соответствующему `end` или правой скобке. Более того, выход такого редактора зачастую подобен выходу после фазы анализа компиляции.
2. *Программы форматированного вывода на печать.* С помощью этих инструментов программа анализируется и распечатывается таким образом, чтобы ее структура была максимально ясной. Например, комментарии могут быть выделены специальным шрифтом, а операторы — выведены с отступами, указывающими уровень вложенности в иерархической структуре операторов.

3. *Статические проверяющие программы*. Данные инструменты считывают программы, анализируют их и пытаются найти потенциальные ошибки без запуска программы. Такой анализ зачастую очень похож на анализ в оптимизирующих компиляторах, обсуждаемых в главе 10, “Оптимизация кода”. Например, статическая проверяющая программа может определить невыполнение какой-то части исходной программы или использование некоторой переменной до ее объявления. Точно так же могут быть обнаружены логические ошибки, например попытки использования действительной переменной в качестве указателя (с применением технологии проверки типов, обсуждаемой в главе 6, “Проверка типов”).
4. *Интерпретаторы*. Вместо создания целевой программы в результате трансляции интерпретатор выполняет операции, указанные в исходной программе. Например, для оператора присвоения он может построить дерево, подобное приведенному на рис. 1.2, а затем выполнить операции, проходя по его узлам. Корень дерева указывает на выполнение присвоения, так что интерпретатор вызовет подпрограмму для вычисления выражения, определяемого правым поддеревом, а затем сохранит его значение в переменной `position`. Правое поддерево указывает подпрограмме, что она должна вычислить сумму двух выражений. Рекурсивный вызов подпрограммы приводит к вычислению значения `rate * 60`, которое затем суммируется со значением переменной `initial`.

Интерпретаторы часто используются для командных языков, поскольку каждый их оператор представляет собой вызов сложной программы, такой как редактор или компилятор. Точно так же и некоторые языки “очень высокого уровня”, типа APL, обычно интерпретируются, поскольку имеется множество атрибутов данных, таких как размер или тип массива, которые не могут быть определены в процессе компиляции.

Традиционно мы говорим о компиляторе как о программе, которая транслирует исходный язык типа Fortran в ассемблер или машинный язык. Однако имеются и другие применения технологии компиляции. Так, анализирующая часть в каждом из приведенных ниже примеров подобна анализатору обычного компилятора.

1. *Форматирование текста*. Программа форматирования текста получает на вход поток символов, большинство из которых представляет выводимый текст, но многие из символов означают абзацы, рисунки или математические структуры, например верхние или нижние индексы. Мы вернемся к использованию анализа при форматировании текста в следующем разделе.
2. “Кремневые” компиляторы (*Silicon compilers*). Такой компилятор имеет исходный язык, схожий с обычным языком программирования. Однако переменные языка представляют не положения в памяти, а логические сигналы (0 или 1) или группы сигналов в коммутируемых линиях. На выходе такого компилятора получается схема устройства на соответствующем языке. (Детальное об этих компиляторах см. в [218], [434] или [437].)
3. *Интерпретаторы запросов*. Данные интерпретаторы транслируют предикаты, содержащие операторы отношений и логические операторы, в команды поиска в базе данных записей, удовлетворяющих данному предикату (см. [102] и [436])¹.

¹ В настоящее время наличие языка структурированных запросов SQL снимает вопрос о том, является ли интерпретация запросов задачей, всего лишь схожей с компиляцией. — Прим. перев.

Контекст компилятора

При создании целевой программы, кроме компилятора, может потребоваться и ряд других программ. Исходная программа может быть разделена на модули, хранимые в отдельных файлах. Задача сбора исходной программы иногда поручается отдельной программе — препроцессору, который может также раскрывать в тексте исходной программы сокращения, называемые макросами.

На рис. 1.3 показана типичная “компиляция”. Целевая программа, создаваемая компилятором, может потребовать дополнительной обработки перед запуском. Компилятор, представленный на рис. 1.3, создает ассемблерный код, который переводится ассемблером в машинный код, а затем связывается (“линкуется”) совместно с некоторыми библиотечными программами в реально запускаемый на машине код.

В следующих двух разделах мы рассмотрим компоненты компилятора; остальные программы, представленные на рис. 1.3, будут рассмотрены в разделе 1.4.



Рис. 1.3. Система обработки языка

1.2. Анализ исходной программы

В этом разделе будет рассмотрен процесс анализа и проиллюстрировано его использование в некоторых языках форматирования текста. Детальнее об этом будет говориться в главах 2–4 и 6. При компиляции анализ состоит из трех фаз.

1. *Линейный анализ*, при котором поток символов исходной программы считывается слева направо и группируется в *токены* (*token*), представляющие собой последовательности символов с определенным совокупным значением.
2. *Иерархический анализ*, при котором символы или токены иерархически группируются во вложенные конструкции с совокупным значением.
3. *Семантический анализ*, позволяющий проверить, насколько корректно совместное размещение компонентов программы.

Лексический анализ

В компиляторах линейный анализ называется *лексическим*, или *сканированием*. Например, при лексическом анализе символы в инструкции присвоения

`position := initial + rate * 60`

будут сгруппированы в следующие токены.

1. Идентификатор `position`.
2. Символ присвоения `:=`.
3. Идентификатор `initial`.
4. Знак сложения.
5. Идентификатор `rate`.
6. Знак умножения.
7. Число 60.

Пробелы, разделяющие символы этих токенов, при лексическом анализе обычно отбрасываются.

Синтаксический анализ

Иерархический анализ называется *разбором* (*parsing*), или *синтаксическим анализом*, который включает группирование токенов исходной программы в грамматические фразы, используемые компилятором для синтеза вывода. Обычно грамматические фразы исходной программы представляются в виде дерева, пример которого показан на рис. 1.4.

В выражении `initial+rate*60` фраза `rate*60` является логической единицей, поскольку обычные соглашения о приоритете арифметических операций гласят, что умножение выполняется до сложения. Поскольку после выражения `initial+rate` следует знак умножения `*`, само по себе оно не группируется в единую фразу на рис. 1.4.

Иерархическая структура программы обычно выражается рекурсивными правилами. Например, при определении выражений можно придерживаться следующих правил.

1. Любой идентификатор (*identifier*) есть выражение (*expression*).
2. Любое число (*number*) есть выражение (*expression*).
3. Если $expression_1$ и $expression_2$ являются выражениями, то выражениями являются и
 - $expression_1 + expression_2$
 - $expression_1 * expression_2$
 - $(expression_1)$.

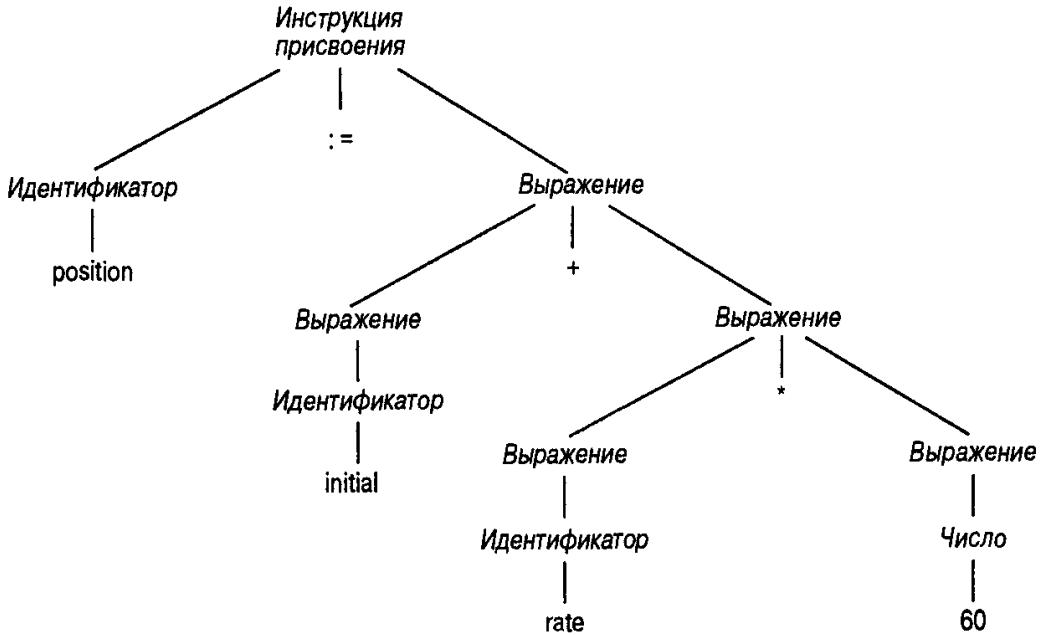


Рис. 1.4. Дерево разбора для выражения `position:=initial+rate*60`

Правила (1) и (2) являются базовыми (нерекурсивными), в то время как (3) определяет выражения с помощью операторов, применяемых к другим выражениям. Согласно правилу (1), `initial` и `rate` представляют собой выражения; правило (2) гласит, что `60` также является выражением. Таким образом, из (3) можно сначала сделать вывод, что `rate*60` — выражение, а затем — что выражением является и `initial+rate*60`.

Точно так же многие языки программирования рекурсивно определяют инструкции языка правилами типа приведенных далее.

1. Если $identifier_1$ является идентификатором, а $expression_2$ — выражением, то
 $identifier_1 := expression_2$
есть инструкция.
2. Если $expression_1$ — выражение, а $statement_2$ — инструкция, то
 $while (expression_1) do statement_2$
 $if (expression_1) then statement_2$
являются инструкциями².

Разделение анализа на лексический и синтаксический достаточно произвольно. Обычно оно используется для упрощения анализа в целом. Одним из факторов, определяющих данное разделение, является использование рекурсии в правилах анализа. Лексические конструкции не требуют рекурсии, в то время как синтаксические редко обходятся без нее. Контекстно-свободные грамматики представляют собой формализацию

² Здесь следует обратить внимание на перевод термина *statement*. Дословно *statement* означает *высказывание, утверждение*, однако в применении к компьютерной тематике это не совсем удачно. Обычно при переводе используется термин *оператор*, но в данной книге, посвященной формальным языкам, этот термин имеет собственное значение. Поэтому при переводе термина *statement* за редкими исключениями было использовано понятие *инструкция* — как наиболее близкое по смыслу, так и уже используемое при описании языка, например, в книге Б. Страуструп. *Язык программирования C++*, 3-е изд. — СПб.; М.: “Невский Диалект” — “Издательство БИНОМ”, 1999. — Прим. перев.

рекурсивных правил, используемых при синтаксическом анализе. Данные грамматики рассматриваются в главе 2 и подробно изучаются в главе 4.

Например, рекурсия не нужна при распознавании идентификаторов, которые обычно представляют собой строки букв и цифр, начинающиеся с буквы. Распознать идентификатор можно с помощью простого последовательного сканирования входящего потока до тех пор, пока в нем не встретится символ, не являющийся символом идентификатора. После этого сканированные символы группируются в токен, представляющий идентификатор. Сгруппированные символы записываются в так называемую таблицу символов, удаляются из входного потока, и начинается сканирование следующего токена.

Однако такое линейное сканирование недостаточно для анализа выражений или инструкций. Например, мы не можем проверить соответствие скобок в выражениях или ключевых слов `begin` и `end` в инструкциях без наложения некоторой иерархической или вложенной структуры на вводимые данные.

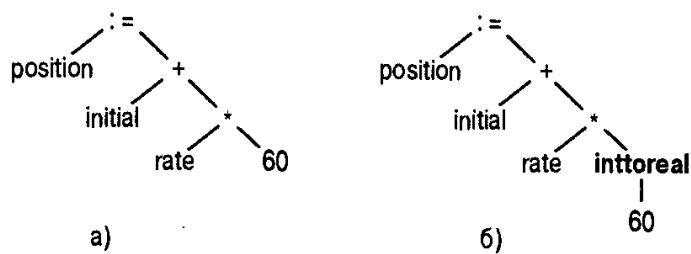


Рис. 1.5. Семантический анализ добавляет преобразование из целого числа в действительное

Дерево разбора, показанное на рис. 1.4, описывает синтаксическую структуру поступающей информации. Более общее внутреннее представление этой синтаксической структуры представлено на рис. 1.5a. Синтаксическое дерево — это “сжатое” дерево разбора, в котором операторы размещены во внутренних узлах, а operandы оператора представлены дочерними ветвями узла, представляющего этот оператор. Построение подобных деревьев (рис. 1.5a) обсуждается в разделе 5.2. В главе 2, “Простой однопроходный компилятор”, мы приступим к рассмотрению *синтаксически управляемой трансляции (syntax-directed translation)*, а в главе 5, “Синтаксически управляемая трансляция”, изучим ее подробнее. При синтаксически управляемой трансляции для построения вывода компилятор использует иерархическую структуру вводимой информации.

Семантический анализ

В процессе семантического анализа проверяется наличие семантических ошибок в исходной программе и накапливается информация о типах для следующей стадии — генерации кода. При семантическом анализе используются иерархические структуры, полученные во время синтаксического анализа для идентификации операторов и operandов выражений и инструкций.

Важным аспектом семантического анализа является проверка типов, когда компилятор проверяет, что каждый оператор имеет operandы допустимого спецификациями языка типа. Например, определение многих языков программирования требует, чтобы при использовании действительного числа в качестве индекса массива генерировалось сообщение об ошибке. В то же время спецификация языка может позволить определенное насилиственное преобразование типов, например, когда бинарный арифметический оператор применяется к operandам целого и действительного типов. В этом случае компилятору может потребоваться преобразование целого числа в действительное. Проверка типов и семантический анализ обсуждаются в главе 6, “Проверка типов”.

Пример 1.1

Битовое представление целого числа в компьютере, вообще говоря, отличается от битового представления действительного числа, даже если эти числа имеют одно и то же значение. Предположим, что все идентификаторы на рис. 1.5 объявлены как имеющие действительный тип, а 60 трактуется как целое число. При проверке типов на рис. 1.5а будет обнаружено, что оператор * применяется к действительному числу `rate` и целому 60. Обычно при этом осуществляется преобразование целого числа в действительное; на рис. 1.5б для этого создается дополнительный узел для оператора `inttoreal`, который неявно преобразует целое число в действительное. Однако, поскольку operand оператора `inttoreal` представляет собой константу, компилятор может вместо этого сам заменить целую константу на эквивалентную действительную. □

Анализ в программах форматирования текста

В программах форматирования текста удобно рассматривать входную информацию как иерархию блоков (*boxes*). Эти блоки являются прямоугольными областями битовых образов, представляющих светлые и темные пиксели на выводящем устройстве.

Так, например, система $T_E\!X$ ([260]) работает именно таким образом. Каждый символ, который не является частью команды, представляет собой блок, содержащий битовый образ этого символа в определенном шрифте требуемого размера. Последовательные символы, не отделенные “разделителями” (пробелами или символами новой строки), группируются в слова, состоящие из последовательностей горизонтальных блоков, как схематически показано на рис. 1.6. Группирование символов в слова (или команды) представляет собой линейный, или лексический аспект анализа программы форматирования текста.



Рис. 1.6. Группировка символов и слов в блоки

В $T_E\!X$ блоки могут быть построены из меньших блоков в различных горизонтальных и вертикальных сочетаниях. Например,

```
\hbox{ <список блоков> }
```

группирует список блоков, собранных по горизонтали. По вертикали блоки группируются с помощью команды `\vbox`. Таким образом, следующая конструкция в $T_E\!X$

```
\hbox{\vbox{! 1} \vbox{@ 2}}
```

представляет набор блоков, показанный на рис. 1.7. Определение иерархического расположения блоков, заданного входным потоком, является частью синтаксического анализа в $T_E\!X$.

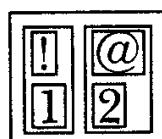


Рис. 1.7. Иерархия блоков в $T_E\!X$

Еще одним примером могут послужить математические препроцессоры EQN ([246]) и \TeX , создающие математические выражения из операторов типа `sub` и `sup` для нижних и верхних индексов. Если EQN встречает входной текст вида

`BOX sub box`

он изменяет размеры блока `box` и присоединяет к блоку `BOX` справа внизу, как показано на рис. 1.8. Оператор `sup` приведет к блоку такого же размера, но размещенному справа вверху.



Рис. 1.8. Построение нижнего индекса в математическом тексте

Такие операторы могут использоваться рекурсивно, т.е. EQN-текст
`a sub { i sup 2 }`

дает в результате a_{i^2} . Группировка операторов `sub` и `sup` в токены представляет собой часть лексического анализа текста EQN. Однако для определения размера и размещения блоков требуется синтаксическая структура текста.

1.3. Фазы компилятора

Концептуально компилятор работает *пофазно*, причем в процессе каждой фазы происходит преобразование исходной программы из одного представления в другое. Типичное разбиение компилятора на фазы показано на рис. 1.9. На практике некоторые фазы могут быть сгруппированы вместе, как упоминается в разделе 1.5, и промежуточные представления программы внутри таких групп могут явно не строиться.

Первые три фазы, формирующие анализирующую часть компилятора, были рассмотрены в предыдущем разделе. Управление таблицей символов и обработка ошибок показаны во взаимодействии с шестью фазами: лексическим анализом, синтаксическим анализом, семантическим анализом, генерацией промежуточного кода, оптимизацией кода и генерацией кода. Неформально диспетчер таблицы символов и обработчик ошибок также могут считаться “фазами” компилятора.

Управление таблицей символов

Одной из важных функций компилятора является запись используемых в исходной программе идентификаторов и сбор информации о различных атрибутах каждого идентификатора. Эти атрибуты предоставляют сведения об отведенной идентификатору памяти, его типе, области видимости (где в программе он может применяться). При использовании имен процедур атрибуты говорят о количестве и типе их аргументов, методе передачи каждого аргумента (например, по ссылке) и типе возвращаемого значения, если таковое имеется.

Таблица символов представляет собой структуру данных, содержащую записи о каждом идентификаторе с полями для его атрибутов. Данная структура позволяет быстро найти информацию о любом идентификаторе и внести необходимые изменения. Таблицы символов подробнее рассматриваются в главах 2, “Простой однопроходный компилятор”, и 7, “Среды времени исполнения”.

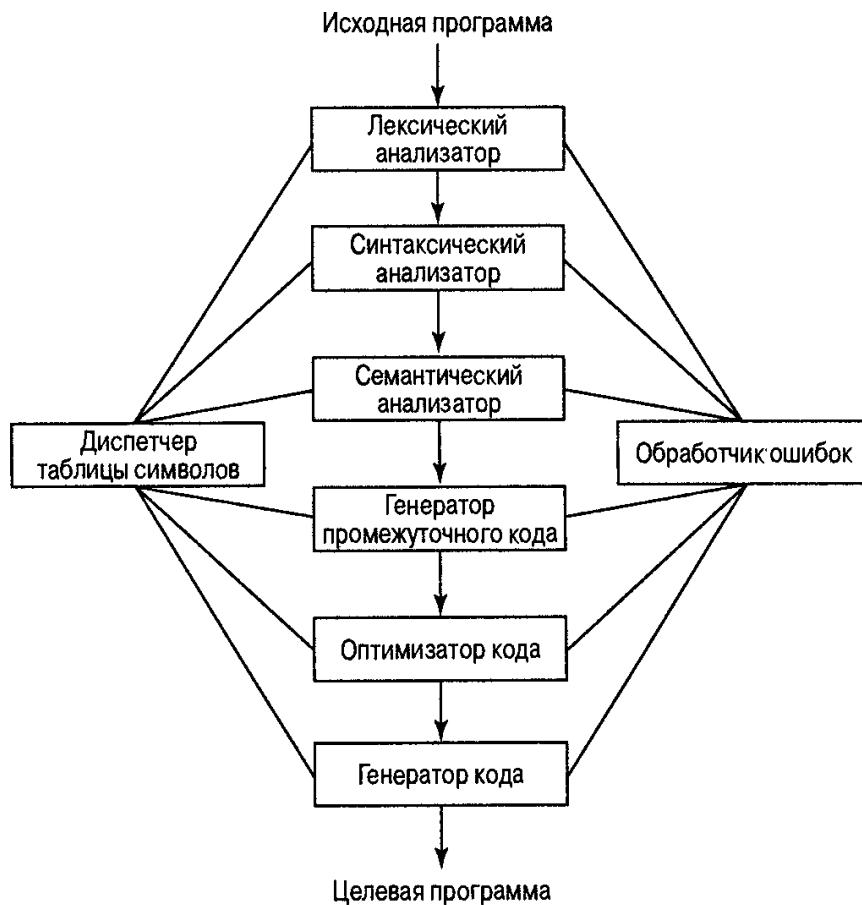


Рис. 1.9. Фазы компилятора

Если лексическим анализатором в исходной программе обнаружен идентификатор, он записывается в таблицу символов. Однако атрибуты идентификатора обычно не могут быть определены в процессе лексического анализа. Например, в объявлении переменных на языке Pascal

`var position, initial, rate : real;`

когда лексический анализатор находит идентификаторы `position`, `initial` и `rate`, их тип `real` еще неизвестен.

В процессе остальных фаз информация об идентификаторах вносится в таблицу символов и используется различными способами. Например, при семантическом анализе и генерации промежуточного кода необходимо знать типы идентификаторов, чтобы гарантировать их корректное использование в исходной программе и сгенерировать правильные операции по работе с ними. Обычно генератор кода вносит в таблицу символов и использует детальную информацию о памяти, назначенной идентификаторам.

Обнаружение ошибок и сообщение о них

В каждой фазе компиляции могут встретиться ошибки. Однако после их обнаружения необходимы определенные действия, чтобы продолжить компиляцию и выявить другие ошибки в исходной программе. Компилятор, который останавливается при обнаружении первой же ошибки, не настолько полезен в работе, каким мог бы быть.

В процессе синтаксического и семантического анализа обычно обрабатывается большая часть ошибок, обнаруживаемых компилятором. При лексическом анализе выявляются ошибки, при которых символы из входного потока не формируют ни один из токенов языка. Ошибки, при которых поток токенов нарушает структурные правила

(синтаксис) языка, определяются в фазе синтаксического анализа. В процессе семантического анализа компилятор пытается обнаружить конструкции, корректные с точки зрения синтаксиса, но не имеющие смысла с точки зрения выполняемых операций, например попытка сложения двух идентификаторов, один из которых — имя массива, а второй — имя процедуры. Обработка ошибок в каждой фазе компиляции будет рассмотрена детальнее в разделах книги, посвященных каждой из фаз в отдельности.

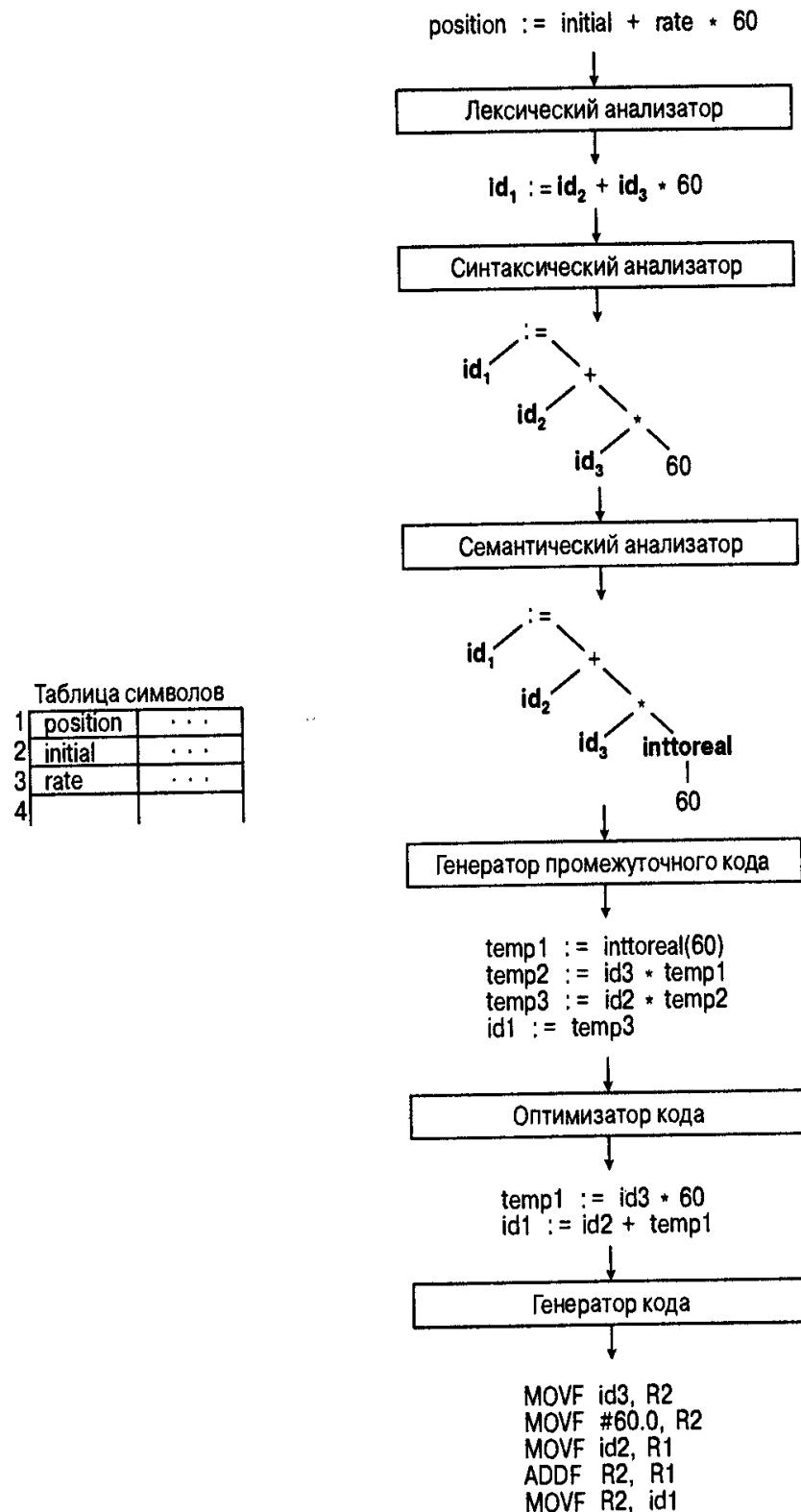


Рис. 1.10. Трансляция инструкции

Фазы анализа

В процессе трансляции изменяется внутреннее представление исходной программы. Для иллюстрации рассмотрим инструкцию.

position := initial + rate * 60 (1.1)

На рис. 1.10 показано внутреннее представление инструкции после каждой фазы.

При лексическом анализе символы исходной программычитываются и группируются в поток токенов, в котором каждый токен представляет логически связанную последовательность символов — идентификатор, ключевое слово (*if*, *while* и т.п.), символ пунктуации или многосимвольный оператор типа `:=`. Последовательность символов, формирующая токен, называется *лексемой* токена.

Некоторые токены дополняются “лексическим значением”. Например, когда найден идентификатор *rate*, лексический анализатор не только генерирует токен, скажем *id*, но и вносит лексему *rate* в таблицу символов, если ее там нет. Лексическое значение, связанное с этим появлением токена *id*, указывает на запись *rate* в таблице символов.

В этом разделе мы будем использовать обозначения *id₁*, *id₂* и *id₃* для *position*, *initial* и *rate*, чтобы подчеркнуть отличие внутреннего представления идентификатора от последовательности символов, формирующих его. Инструкция (1.1) после лексического анализа выглядит так:

id₁ := *id₂* + *id₃* * 60 (1.2)

Нужно также создать токены для многосимвольного оператора `:=` и числа 60, но об этом — в главе 2, “Простой однопроходный компилятор”. Детальнее лексический анализ рассматривается в главе 3, “Лексический анализ”.

Вторая и третья фазы, синтаксический и семантический анализ, также были рассмотрены в разделе 1.2. При синтаксическом анализе на поток токенов налагается иерархическая структура, которую можно изобразить с помощью синтаксического дерева, приведенного на рис. 1.11a. Типичная структура данных для такого дерева показана на рис. 1.11б. Ее внутренний узел представляет собой запись с полем для оператора и двумя полями с указателями на дочерние записи. Лист представляет собой запись с двумя или более полями (для идентификации токена в листе и записи информации о токене). Дополнительная информация о языковых конструкциях может содержаться в дополнительных полях записей для узлов. Синтаксический и семантический анализ будут детальнее рассматриваться в главах 4, “Синтаксический анализ”, и 6, “Проверка типов”, соответственно.

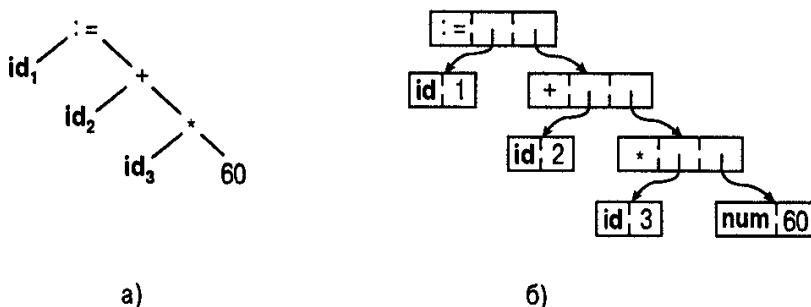


Рис. 1.11. Структура данных (б) для дерева (а)

Генерация промежуточного кода

После синтаксического и семантического анализа некоторые компиляторы генерируют явное промежуточное представление исходной программы, которое можно рассматривать как программу для абстрактной машины. Это представление должно легко создаваться и транслироваться в целевую программу.

Промежуточное представление может иметь ряд форм. В главе 8, “Генерация промежуточного кода”, мы рассмотрим промежуточную форму, называемую “трехадресным кодом”. Этот код похож на язык ассемблера для машины, в которой каждая ячейка памяти может работать в качестве регистра. Трехадресный код состоит из последовательности инструкций, каждая из которых имеет не более трех operandов. Исходная программа (1.1) в трехадресном коде может выглядеть так:

```
temp1 := inttoreal(60)
temp2 := id3 * temp1
temp3 := id2 + temp2
id1 := temp3
```

(1.3)

Данная промежуточная форма имеет ряд специфических свойств. Во-первых, каждая трехадресная инструкция помимо присвоения имеет не более одного оператора. Поэтому при генерации этих инструкций компилятор должен определить порядок выполнения операций; так, в исходной программе (1.1) умножение предшествует сложению. Во-вторых, компилятор должен сгенерировать временное имя для хранения результатов, вычисляемых каждой инструкцией. В-третьих, некоторые трехадресные инструкции имеют меньше трех operandов (например, первая и последняя инструкции в (1.3)).

В главе 8, “Генерация промежуточного кода”, будут рассмотрены основные промежуточные представления, используемые в компиляторах. Вообще говоря, эти представления должны не только вычислять выражения, но и управлять потоком инструкций и вызовов процедур. В главах 5, “Синтаксически управляемая трансляция”, и 8, “Генерация промежуточного кода”, представлены алгоритмы генерации промежуточного кода для типичных конструкций языков программирования.

Оптимизация кода

При оптимизации кода производятся попытки улучшить промежуточный код, чтобы получить более эффективный машинный код. Некоторые оптимизации тривиальны. Например, естественный алгоритм генерирует промежуточный код (1.3), используя инструкцию для каждого оператора в дереве, созданном в результате семантического анализа, хотя те же вычисления можно выполнить с помощью двух инструкций:

```
temp1 := id3 * 60.0
id1 := id2 + temp1
```

(1.4)

Оптимизирующее преобразование может быть выполнено в процессе генерации кода. Во-первых, компилятор может принять решение о преобразовании целого числа 60 в действительное еще во время компиляции, устранив необходимость в операции `inttoreal`. Во-вторых, переменная `temp3` используется лишь для передачи своего значения в `id1`, поэтому вполне корректно использовать `id1` вместо `temp3` и удалить последнюю инструкцию из (1.3), получая код (1.4).

Степень оптимизации кода у различных компиляторов значительно отличается. В некоторых компиляторах, называемых “оптимизирующими”, большая часть времени рабо-

ты уходит именно на оптимизацию. Однако существуют простые приемы оптимизации, которые существенно уменьшают время работы целевой программы, не сильно замедляя работу компилятора. Многие из этих приемов рассматриваются в главе 9, “Генерация кода”, а в главе 10, “Оптимизация кода” вы познакомитесь с технологиями, используемыми более мощными оптимизирующими компиляторами.

Генерация кода

Последняя фаза компиляции состоит в генерации целевого кода, обычно перемещаемого (relocatable) машинного кода или ассемблерного кода. Для каждой переменной программы определяется ее положение в памяти. После этого каждая промежуточная инструкция транслируется в последовательность машинных инструкций, выполняющих ту же самую работу. Ключевой аспект этой фазы заключается в назначении переменных регистрам.

Например, используя регистры 1 и 2 при трансляции кода (1.4), получим следующий код:

```
MOVF id3, R2  
MULF #60.0, R2  
MOVF id2, R1  
ADDF R2, R1  
MOVF R1, id1
```

(1.5)

Первый и второй операнды каждой инструкции определяют источник и приемник. Ей говорит о том, что в инструкции используются числа с плавающей точкой. Этот код перемещает содержимое адреса³ id3 в регистр 2, затем умножает его на действительную константу 60.0. Символ # означает, что 60.0 представляет собой константу. Третья инструкция переносит значение id2 в регистр 1, а затем к нему добавляется вычисленное ранее значение из регистра 2. И наконец, значение из регистра 1 переносится в ячейку памяти с адресом id1. Тем самым код реализует присвоение, показанное на рис. 1.10. О генерации кода вы прочтете в главе 9, “Генерация кода”.

1.4. Родственники компилятора

Как видно из рис. 1.3, входная информация для компилятора может порождаться одним или несколькими препроцессорами; кроме того, после компиляции может потребоваться дополнительная обработка для получения выполняемого машинного кода. В этом разделе мы рассмотрим типичное окружение, в котором работает компилятор.

Препроцессоры

Препроцессоры создают входной поток информации для компилятора. С их помощью можно выполнить следующие функции.

1. *Обработка макросов.* Пользователь может определить макросы — краткие записи длинных конструкций.

³ Следует упомянуть о важности вопроса выделения памяти для размещения идентификаторов исходной программы. Как мы увидим в главе 7, “Среды времени исполнения”, распределение памяти в процессе работы зависит от компилируемого языка. Принятие решения о выделении памяти происходит либо в процессе создания промежуточного кода, либо при генерации целевого кода.

2. *Включение файлов.* В текст программы можно включить заголовочные файлы. Например, при обработке файла препроцессор С заменяет выражение `#include <global.h>` содержимым файла `global.h`.
3. *“Интеллектуальные” препроцессоры.* К старым языкам добавляются более современные возможности управления выполнением программы и работы со сложными структурами данных. Например, с помощью таких препроцессоров можно использовать встроенные макросы для построения циклов `while` или условных конструкций, отсутствующих в языке программирования.
4. *Языковые расширения.* Примером может послужить язык EqueL ([413]) — язык запросов к базе данных, внедренный в код С. Препроцессор получает инструкции, начинающиеся с `##` (это инструкции доступа к базе данных, не имеющие никакого отношения к С), и переводит их в вызовы процедур, реализующих обращения к базе данных.

Обработчики макросов работают с двумя видами инструкций — определение макросов и их использование. Определения обычно указываются с помощью определенного символа или ключевого слова типа `define` или `macro` и состоят из имени определяемого макроса и его *тела*, формируя определение макроса. Зачастую макропроцессоры позволяют применять в определениях макросов *формальные параметры*, т.е. символы, заменяемые значениями при использовании макроса (в данном контексте “значение” — строка символов). Использование макроса представляет собой его имя с *фактическими параметрами*, т.е. значениями для подстановки вместо формальных параметров. Макропроцессор подставляет в тело макроса фактические значения вместо формальных параметров; затем преобразованное тело макроса замещает его имя в программе.

Пример 1.2

\TeX , о котором упоминалось в разделе 1.2, позволяет работать с макросами. Определение макроса имеет вид

```
\define <имя макроса> <шаблон> {<тело>}
```

Имя макроса представляет собой строку символов, начинающуюся с обратной косой черты. Шаблон — строка символов, в которой строки типа `#1`, `#2` ... `#9` рассматриваются как формальные параметры. Эти символы могут появляться в теле макроса сколько угодно раз. Например, следующий макрос определяет ссылку на *Journal of the ACM*.

```
\define \JACM #1;#2;#3.  
{{\sl J. ACM} {\bf #1}:#2, pp. #3.}
```

Имя макроса — `\JACM`, а шаблон — “`#1;#2;#3.`”; точки с запятой разделяют отдельные параметры, а за последним параметром следует точка. Использование такого макроса должно иметь тот же вид, что и шаблон, с тем отличием, что вместо формальных параметров могут использоваться произвольные строки.⁴ Таким образом, мы можем записать

⁴ “Почти произвольные” строки, поскольку производится только простое сканирование макроса. Как только при сканировании находится символ, соответствующий тексту, следующему после `#1` в шаблоне, сканированная строка считается соответствующей формальному параметру `#1`. Таким образом, если мы попытаемся подставить `ab; cd` вместо `#1`, обнаружим, что параметру `#1` соответствует строка `ab`, а параметру `#2` — строка `cd`.

\JACM 17; 4; 715–728.

и получить при этом

J. ACM 17:4, pp. 715–728.

Часть тела макроса `\s1 J. ACM` обеспечивает вывод текста *J. ACM* наклонным (slanted⁵) шрифтом. Выражение `\bf #1` говорит о том, что первый фактический параметр должен быть выведен полужирным шрифтом (boldface). Этот параметр представляет собой номер тома журнала.

\TeX позволяет использовать любые знаки пунктуации или строки текста для разделения тома, выпуска и номеров страниц в определении макроса `\JACM`. Можно обойтись и без разделителей — в этом случае \TeX будет считать фактическими параметрами отдельные символы или строки, взятые в фигурные скобки `{ }`. \square

Ассемблеры

Некоторые компиляторы создают ассемблерный код, как в (1.5), который передается для дальнейшей обработки ассемблеру. Другие компиляторы самостоятельно выполняют работу ассемблера, производя перемещаемый машинный код, который непосредственно передается загрузчику/редактору связей. Читатель наверняка знает, как выглядит ассемблерный код и что такое ассемблер. Здесь же мы рассмотрим отношения между ассемблерным и машинным кодами.

Ассемблерный код представляет собой мнемоническую версию машинного кода, в которой вместо бинарных кодов операций используются их имена; кроме того, адресам памяти также могут присваиваться имена. Типичная последовательность инструкций выглядит как

```
MOV a, R1  
ADD #2, R1  
MOV R1, b
```

(1.6)

Этот код перемещает содержимое памяти по адресу *a* в регистр 1, затем добавляет к нему константу 2, рассматривая содержимое регистра 1 как число с фиксированной точкой, и сохраняет результат в именованной ячейке памяти *b*. Таким образом вычисляется *b* := *a* + 2.

Имеет ли язык ассемблера возможности работы с макросами, зависит от типа ассемблера.

Двухпроходный ассемблер

Простейший ассемблер делает два прохода по входному потоку (в данном случае проход — разовое считывание входного файла). При первом проходе находятся все идентификаторы, обозначающие ячейки памяти, и размещаются в таблице символов (отличной от таблицы символов компилятора). Идентификаторам назначаются адреса в памяти, так что после чтения (1.6) таблица символов может содержать записи, показанные на рис. 1.12. Мы предположили, что для каждого идентификатора выделяется одно слово памяти, состоящее из четырех байт, и адреса начинаются с нулевого адреса.

⁵ Slanted — наклоненный. — Прим. перев.

ИДЕНТИФИКАТОР	АДРЕС
a	0
b	4

Рис. 1.12. Таблица символов ассемблера с идентификаторами из (1.6)

При втором проходе ассемблер вновь сканирует входной поток. В этот раз он переводит каждый код операции в последовательность битов, представляющих операцию на машинном языке, а каждый идентификатор — в адрес, назначенный идентификатору в таблице символов.

В результате второго прохода обычно получается *перемещаемый* (relocatable) машинный код, что означает, что он может быть загружен в память с любого стартового адреса L. Если L будет добавлено ко всем адресам в коде, то все ссылки будут совершенно корректны. Таким образом, выходной код ассемблера должен различать части инструкций, ссылающиеся на адреса, которые могут быть перенесены.

Пример 1.3

Далее следует гипотетический машинный код, в который переводятся инструкции из (1.6).

0001 01 00 00000000 *	
0011 01 10 00000010	(1.7)
0010 01 00 00000100 *	

Инструкция представлена в виде слова, в котором первые четыре бита являются кодом инструкции (0001, 0010 и 0011 соответствуют загрузке, сохранению и сложению). Под загрузкой и сохранением подразумевается перемещение из памяти в регистр и наоборот. Следующие два бита определяют используемый регистр; 01 означает, что во всех трех командах используется первый регистр. Два последующих бита определяют “дескриптор”. 00 означает режим обычной адресации, при котором последние восемь бит представляют собой адрес памяти; дескриптор 10 указывает на “непосредственный” режим, когда последние восемь бит являются операндом. Этот режим используется во второй команде (1.7).

В (1.7) есть также символ “*” — *бит перемещаемости* — который имеется у каждого операнда в перемещаемом машинном коде. Предположим, что адресное пространство содержит данные, загруженные начиная с адреса L. В этом случае символ * означает, что L должно быть добавлено к адресу операнда. Таким образом, если L=00001111, т.е. 15, то a и b размещаются по адресам 15 и 19. Теперь (1.7) в *абсолютном* (или неперемещаемом) машинном коде будет выглядеть как

0001 01 00 00001111	
0011 01 10 00000010	(1.8)
0010 01 00 00010011	

Заметьте, у второй команды в (1.7) нет связанного бита перемещаемости, поэтому во второй команде прибавления L не происходит (так как, по сути, восьмибитовое значение представляет собой не адрес 2, а константу 2). □

Загрузчики и редакторы связей

Обычно программа, называемая *загрузчиком*, выполняет две функции — загрузку и редактирование связей. Процесс загрузки заключается в получении перемещаемого ма-

шинного кода, изменении перемещаемых адресов, как было описано в примере 1.3, и размещении измененных команд и данных по корректным адресам в памяти.

Редактор связей позволяет собрать единую программу из нескольких файлов с перемещаемым машинным кодом. Эти файлы могут быть результатом различных компиляций; один или несколько из них могут представлять собой библиотечные файлы подпрограмм, предоставляемых системой и доступных любой программе.

Если эти файлы используются вместе, то необходимы *внешние ссылки*, благодаря которым код одного файла ссылается на содержимое другого. Такая ссылка может указывать на данные, определенные в одном файле и использованные в другом, или представлять собой точку входа процедуры, находящейся в одном файле и вызываемой из другого. Файл с перемещаемым машинным кодом должен содержать информацию в таблице символов для каждого данного или инструкции, на которые могут иметься внешние ссылки. Если заранее неизвестно, на что именно в коде могут иметься ссылки, то в качестве части перемещаемого машинного кода должна включаться полная ассемблерная таблица символов.

Например, код (1.7) должен предваряться таблицей

a	0
b	4

Если загружается файл с кодом (1.7), ссылающимся на *b*, то эта ссылка будет заменена на 4 плюс смещение, на которое перемещены данные из файла (1.7).

1.5. Группировка фаз

Фазы компиляции, рассмотренные в разделе 1.3, представляют собой логическую организацию компилятора. При реализации зачастую происходит объединение действий, выполняемых в различных фазах.

Предварительная и заключительная стадии

Зачастую фазы объединяются в *начальную* (front end) и *заключительную* (back end) стадии. Начальная стадия объединяет те фазы компилятора (или части фаз), которые зависят в первую очередь от исходного языка и практически не зависят от целевой машины. Обычно сюда входят лексический и синтаксический анализ, создание таблицы символов, семантический анализ и генерация промежуточного кода. На этом этапе может быть выполнена и определенная часть оптимизации кода. Кроме того, начальная стадия включает обработку ошибок, которая сопровождает каждую фазу компилятора.

Заключительная стадия состоит из тех фаз компилятора, которые в первую очередь зависят от целевой машины, для которой выполняется компиляция, и, вообще говоря, не зависят от исходного языка (а только от промежуточного). Кроме того, сюда входит часть оптимизации кода и генерация выходного кода, сопровождаемые необходимой обработкой ошибок и работой с таблицей символов.

Таким образом, создание компиляторов одного и того же исходного языка для различных машин является достаточно простой операцией. При этом можно использовать одну и ту же начальную стадию, не зависящую от целевой машины. Более того, при умелой разработке заключительной стадии не потребуется даже внесение существенных изменений в нее (см. главу 9, “Генерация кода”). Соблазнительно также компилировать различные исходные языки в один промежуточный и использовать общую заключитель-

ную стадию для разных предварительных стадий, получая тем самым ряд компиляторов для различных языков с одной целевой машиной. Однако в данном случае успех не гарантирован из-за множества нюансов в различных языках программирования.

Проходы

Обычно фазы компиляции реализуются в одном *проходе* (pass), состоящем из чтения входного файла и записи выходного. Имеется множество способов группировки фаз компилятора по проходам, и поэтому рассмотрение процесса компиляции в данной книге в большей степени привязано к фазам, а не проходам. В главе 12, “Некоторые компиляторы” будут рассмотрены некоторые типичные компиляторы и способы распределения фаз компиляции по проходам.

Как упоминалось, группирование нескольких фаз в одном проходе широко распространено; их выполнение чередуется во время прохода. Например, лексический, синтаксический и семантический анализы, а также генерация промежуточного кода могут быть сгруппированы в одном проходе. В этом случае поток токенов после лексического анализа может быть транслирован непосредственно в промежуточный код. Синтаксический анализ можно рассматривать как “руководящий” процесс. Он получает поток токенов с помощью вызова лексического анализатора для получения следующего токена в потоке. После того как определена грамматическая структура, синтаксический анализатор вызывает генератор промежуточного кода для семантического анализа и генерации части кода. Такой компилятор описан в главе 2, “Простой однопроходный компилятор”.

Уменьшение количества проходов

Желательно иметь компилятор с минимальным числом проходов, особенно если учесть время, необходимое для чтения и записи промежуточных файлов. Однако при группировании нескольких фаз в одном проходе может потребоваться размещение программы в памяти целиком — поскольку одной фазе может потребоваться информация в порядке, отличном от того, в котором ее выдает предыдущая фаза. Программа во внутреннем представлении может быть значительно больше исходной программы или программы, получаемой в результате работы компилятора, поэтому вопрос о пространстве далеко не тривиален.

Группировка некоторых фаз в одном проходе не представляет проблем. Как упоминалось выше, с одной стороны, интерфейс между лексическим и синтаксическим анализами зачастую ограничивается единственным токеном. С другой стороны, часто очень сложно выполнить генерацию кода до завершения создания промежуточного представления. Например, языки типа PL/I и Algol 68 позволяют использование переменных до их объявления. Невозможно сгенерировать целевой код для языковой конструкции, если не известны типы используемых в ней переменных. Подобная же ситуация возникает и в языках, которые позволяют использовать оператор безусловного перехода вперед по коду (таких языков программирования подавляющее большинство). Определить целевой адрес такого оператора безусловного перехода невозможно без генерации кода для инструкций между оператором безусловного перехода и его местом назначения.

Иногда удается оставить необходимое для отсутствующей информации пустое место и заполнить его позже, когда информация станет доступной. В частности, генерация промежуточного и целевого кодов часто может быть объединена в один проход с использованием технологии “обратных поправок” (backpatching). До тех пор, пока в гла-

ве 8, “Генерация промежуточного кода”, не будут рассмотрены вопросы, связанные с генерацией промежуточного кода, невозможно пояснить все детали этой технологии. Однако мы попытаемся проиллюстрировать технологию обратных поправок на примере ассемблера. В предыдущем разделе рассматривался двухпроходный ассемблер, когда в первом проходе производился поиск всех идентификаторов, представляющих ячейки памяти, и происходило назначение им адресов, а во втором — идентификаторы заменились адресами.

Можно скомбинировать эти действия следующим образом. При использовании ассемблерной инструкции со ссылкой вперед

```
GOTO target
```

мы генерируем инструкцию с машинным кодом операции GOTO и пустым местом вместо адреса. Все инструкции с пустыми местами вместо адреса target хранятся в списке, связанном с записью для идентификатора target в таблице символов. Эти пустые места заполняются, как только появляется инструкция типа

```
target: MOV foobar, R1
```

и определяется значение идентификатора target, которое является адресом текущей инструкции. Затем производим “обратную поправку”, проходя по списку, связанному с идентификатором target, и внося реальное значение адреса в пустые поля адресов. Такой подход прост в реализации, если инструкции хранятся в памяти до определения всех целевых адресов.

Такой подход вполне допустим в ассемблере, который может хранить весь вывод в памяти. Поскольку промежуточное и окончательное представления кода в случае ассемблера практически одинаковы и имеют близкие размеры, применение технологии обратных поправок не сталкивается с неразрешимыми проблемами. Однако в компиляторах с большим промежуточным кодом, возможно, придется ограничить использование метода обратных поправок некоторым диапазоном, определяемым доступной памятью.

1.6. Инструментарий для создания компиляторов

Создатели компиляторов, как и прочие программисты, могут с пользой применять такие программные инструменты, как отладчики, средства контроля версий, профайлеры и т.п. В главе 11, “Создание компилятора” мы узнаем об использовании некоторых из этих инструментов при реализации компиляторов. В дополнение к этим средствам может использоваться ряд более специализированных инструментов, которые будут вкратце рассмотрены в этом разделе. Подробнее они будут описаны в последующих главах книги.

Вскоре после написания первых компиляторов появились инструменты, облегчающие их создание. Такие системы получили названия *компиляторов компиляторов*, *генераторов компиляторов* или *систем написания трансляторов*. Как правило, они ориентированы на языки определенной модели и наиболее подходят для генерации компиляторов языков именно этой модели.

Например, можно предположить, что лексические анализаторы для всех языков, по сути, одинаковы — за исключением множеств ключевых слов и знаков. Многие компиляторы компиляторов используют фиксированные программы лексического анализа в генерируемых компиляторах. Эти программы пользуются только списком ключевых слов, и все, что требуется от пользователя, — предоставить этот список. Такой подход вполне корректен, но может оказаться неработоспособным, если необходимо распозна-

вание нестандартных токенов, например идентификаторов, которые включают символы, отличающиеся от букв и цифр.

Некоторые инструменты созданы для автоматической разработки определенных компонентов компиляторов. Они используют специализированные языки для определения и реализации компонентов и применяют весьма интеллектуальные алгоритмы. Наиболее эффективные инструменты скрывают детали алгоритма генерации и производят компоненты, легко интегрируемые с остальными частями компилятора. Ниже представлен список некоторых инструментов для создания компиляторов.

1. *Генераторы синтаксических анализаторов.* Эти генераторы производят синтаксические анализаторы, обычно по входной информации, основанной на контекстно-свободной грамматике. В первых компиляторах синтаксический анализ был весьма неэффективен и трудоемок. В настоящее время эта фаза является одной из простейших при реализации. Многие “микроязыки”, использовавшиеся при создании этой книги, такие как PIC ([245]) и EQN, реализованы за несколько дней с помощью анализатора, описанного в разделе 4.7. Многие генераторы синтаксических анализаторов используют мощные алгоритмы разбора, которые слишком сложны для ручной реализации.
2. *Генераторы сканеров.* Этот инструментарий автоматически генерирует лексические анализаторы, как правило, с использованием спецификаций, построенных на регулярных выражениях (об этом рассказывается в главе 3, “Лексический анализ”). В основном, лексические анализаторы действуют по принципу конечного автомата. Типичный генератор сканеров и его реализация описаны в разделах 3.5 и 3.8.
3. *Средства синтаксически управляемой трансляции.* С помощью данного инструментария создаются наборы программ прохода по дереву разбора (наподобие представленного на рис. 1.4) и генерации промежуточного кода. Основная идея состоит в одной или нескольких “трансляциях”, связанных с каждым узлом дерева разбора; при этом каждая трансляция определяется с учетом соседних узлов дерева. Эти средства обсуждаются в главе 5, “Синтаксически управляемая трансляция”.
4. *Автоматические генераторы кода.* Эти инструменты получают набор правил, которые указывают способ трансляции каждой операции промежуточного языка в определенный машинный язык. Правила должны быть достаточно детальны, чтобы обеспечить работу с различными способами доступа к данным. Например, переменные могут находиться в регистрах, фиксированных (статических) ячейках памяти или в стеке. В данном случае применяется технология “согласованных шаблонов”. Инструкции промежуточного кода замещаются “шаблонами”, которые представляют собой последовательности машинных инструкций, согласованные по способу хранения переменных. Поскольку обычно имеется множество вариантов размещения переменных (например, в одном из регистров или в памяти), возможно использование различных способов замещения промежуточного кода набором шаблонов. Необходимо выбрать хорошее замещение шаблонами, без перебора всех возможных вариантов и замедления компиляции. Подобные инструменты рассматриваются в главе 9, “Генерация кода”.
5. *Средства работы с потоком данных.* Для получения оптимизированного выходного кода требуется проведение серьезного анализа потока данных, т.е. сбора и анализа информации о том, каким образом значения передаются из одной части программы в другую. Такие задачи могут быть решены, по сути, с помощью одинаковых программ, одна из которых рассматривается в разделе 10.11.

Библиографические примечания

В 1962 году при работе над историей компиляторов Кнут [251] заметил, что “в этой области необычайно много разработок одних и тех же технологий независимо работающими исследователями”. Он также заметил, что ряд исследователей в действительности открыли “различные аспекты технологий и довели их до весьма изящных алгоритмов, которые так и не были полностью реализованы ни одним из первооткрывателей”. Описать заслуги тех или иных исследователей — достаточно сложная задача, и библиографические примечания в этой книге служат не более чем дополнительной информацией по тому или иному вопросу.

Исторические замечания по разработке программных языков и компиляторов до появления Fortran можно найти в [264]. Работа [454] содержит исторические сведения о ряде языков программирования от участников их разработок.

Ряд некоторых ранних фундаментальных статей о компиляции собран в работах [381] и [349]. *Communications of the ACM* (январь 1961 года) содержит информацию о создании компиляторов в то время. В работе [363] можно познакомиться с компилятором Algol 60.

В 60-х годах теоретические исследования, начатые с изучения вопросов синтаксиса, оказали значительное влияние на развитие технологий компиляции. С тех пор интерес к этой области исследований несколько иссяк, однако она и сегодня продолжает оставаться предметом исследований. Плоды этих исследований будут представлены в следующих главах книги при более детальном изучении вопросов компиляции.

ГЛАВА 2

Простой однопроходный компилятор

Данная глава является вводной к главам 3–8 и представляет ряд базовых технологий компиляции, проиллюстрированных разработкой рабочей программы на С, которая транслирует инфиксные выражения в постфиксную форму. Здесь основное внимание уделяется предварительной стадии компиляции, т.е. лексическому анализу, разбору и генерации промежуточного кода. Главы 9, “Генерация кода”, и 10, “Оптимизация кода” посвящены вопросам генерации и оптимизации кода.

2.1. Обзор

Язык программирования может быть определен с помощью описания того, как должна выглядеть программа (*синтаксис языка*) и что она означает (*семантика языка*). Для определения синтаксиса языка рекомендуем широко применяемую запись, называемую контекстно-свободной грамматикой, или BNF (Backus-Naur Form, форма Бэкуса-Наура). На сегодня описание семантики языка — гораздо более сложная задача, чем описание синтаксиса; для ее решения необходимо использовать неформальные описания и примеры.

Кроме определения синтаксиса языка, контекстно-свободная грамматика используется при трансляции программ. Грамматически управляемая технология компиляции, известная также как *синтаксически управляемая трансляция* (syntax-directed translation), очень полезна при разработке предварительной стадии компилятора и будет широко использоваться в данной главе.

В процессе обсуждения синтаксически управляемой трансляции мы построим компилятор, переводящий инфиксные выражения в постфиксные, в которых операторы появляются после своих операндов. Например, постфиксная форма выражения $9 - 5 + 2$ выглядит как $9\ 5\ -2\ +$. Постфиксная запись может быть преобразована непосредственно в компьютерный код, который выполняет все необходимые вычисления с использованием стека. Давайте начнем с построения простой программы для трансляции в постфиксную форму выражений из цифр, разделенных знаками “плюс” и “минус”. После того как станет понятна основная идея, мы расширим программу для обработки более общих конструкций языка программирования. Каждый из наших трансляторов будет представлять собой расширение предыдущего.

В нашем компиляторе *лексический анализатор* конвертирует поток входных символов в поток токенов, которые представляют собой входной поток для следующей фазы, как показано на рис. 2.1. В данном случае “синтаксически управляемый транслятор” представляет собой комбинацию синтаксического анализатора и генератора промежуточного кода. Одна из причин, по которой мы первоначально ограничиваемся только выражениями, состоящими из цифр и операторов, — простота лексического анализатора

(каждый входной символ представляет собой отдельный токен). Позже мы расширим наш язык, включив в него такие лексические конструкции, как числа, идентификаторы и ключевые слова. Для такого расширенного языка мы построим лексический анализатор, который будет накапливать последовательные символы из входного потока и преобразовывать их в токены. Детально построение такого лексического анализатора будет рассмотрено в главе 3, “Лексический анализ”.

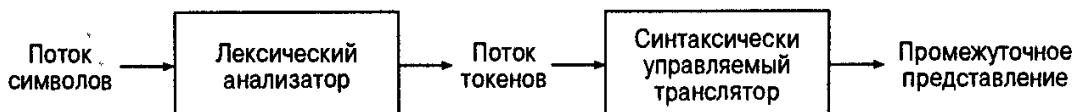


Рис. 2.1. Структура предварительной стадии компилятора

2.2. Определение синтаксиса

В этом разделе для определения синтаксиса языка будет рассмотрен способ записи, называемый контекстно-свободной грамматикой (или, для краткости, просто грамматикой). Данный способ будет использоваться как часть спецификации предварительной стадии компилятора на протяжении всей книги.

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

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

Таким образом, инструкция **if-else** представляет собой ключевое слово **if**, за которым следует открывающая круглая скобка, выражение, закрывающая скобка, инструкция, ключевое слово **else** и еще одна инструкция (в С нет ключевого слова **then**). Используя переменную *expr* для обозначения выражения и переменную *stmt* для обозначения инструкции, можно записать это структурное правило так:

$$stmt \rightarrow if(expr) stmt \text{ else } stmt \quad (2.1)$$

Здесь символ \rightarrow можно прочесть как “может иметь вид”. Такое правило называется *продукцией* (production). В продукции лексические элементы вроде ключевого слова **if** и скобок называются *токенами*. Переменные типа *expr* и *stmt* представляют последовательности токенов¹ и называются *нетерминальными символами*, или просто *нетерминалами* (nonterminals).

Контекстно-свободная грамматика имеет четыре компонента.

1. Множество токенов, представляющих собой *терминальные символы* (или просто *терминалы*).
2. Множество нетерминальных символов.
3. Множество продукции, каждая из которых состоит из нетерминала, называемого *левой частью* продукции, стрелки и последовательности токенов и/или нетерминалов, называемых *правой частью* продукции.
4. Указание одного из нетерминальных символов как *стартового*, или *начального*.

Следует придерживаться правила, согласно которому грамматика определяется перечислением ее продукции, причем первая продукция указывает стартовый символ. Циф-

¹ Вообще говоря, нетерминал представляет множество последовательностей токенов. — Прим. ред.

ры, знаки вроде \leq и выделенные полужирным шрифтом слова типа `while` являются терминальными символами. Выделенные курсивом слова являются нетерминалами, а все слова или символы, поданные без выделения, могут рассматриваться как токены². Для удобства записи правые части продукции с одними и теми же нетерминалами слева могут быть сгруппированы с помощью символа “|” (“или”).

Пример 2.1

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

$$list \rightarrow list + digit \quad (2.2)$$

$$list \rightarrow list - digit \quad (2.3)$$

$$list \rightarrow digit \quad (2.4)$$

$$digit \rightarrow 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 \quad (2.5)$$

Правые части трех продукции с нетерминалом *list* в левой части могут быть объединены:

$$list \rightarrow list + digit | list - digit | digit$$

Здесь, в соответствии с нашими соглашениями, токенами грамматики являются символы

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

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

Продукция называется *продукцией нетерминала*, если он записан в левой части. Стока токенов является последовательностью из нуля или нескольких токенов. Стока, содержащая нуль токенов и записываемая как ϵ , называется *пустой*.

Грамматика *выводит*, или *порождает* строки, начиная со стартового символа и неоднократно замещая нетерминалы правыми частями продукции этих нетерминалов. Строки токенов, порождаемые из стартового символа, образуют *язык*, определяемый грамматикой.

Пример 2.2

Язык, определяемый грамматикой из примера 2.1, состоит из списков цифр, разделенных знаками “плюс” и “минус”.

Десять продукции для нетерминала *digit* позволяют ему быть любым из токенов 0, 1, ..., 9. Из продукции (2.4) следует, что список может состоять и из одной цифры, т.е. цифра сама по себе является списком. Продукции (2.2) и (2.3) выражают тот факт, что если мы возьмем любой список и добавим к нему знак “плюс” или “минус” с последующей цифрой, то получим новый список.

Оказывается, что продукции (2.2)–(2.5) — все, что надо для определения языка. Например, можно сделать вывод, что $9-5+2$ является списком, следующим образом.

² Отдельные символы, выделенные курсивом, будут использоваться и для других целей при детальном изучении грамматики в главе 4, “Синтаксический анализ”. Например, они будут применяться при указании символов, которые могут представлять собой либо токены, либо нетерминалы. Однако выделенное курсивом имя из двух или более символов будет всегда означать нетерминал.

1. 9 — список в соответствии с продукцией (2.4), поскольку 9 — цифра.
2. 9-5 — список в соответствии с продукцией (2.3), так как 9 — список, а 5 — цифра.
3. 9-5+2 — список в соответствии с продукцией (2.2), поскольку 9-5 — список, а 2 — цифра.

Данное утверждение продемонстрируем деревом, представленным на рис. 2.2. Каждый узел дерева помечен символом грамматики. Внутренний узел и его дочерние узлы соответствуют продукции, причем узел соответствует левой части продукции, а потомки — правой. Такие деревья называются деревьями разбора (parse trees); они будут рассмотрены ниже. \square

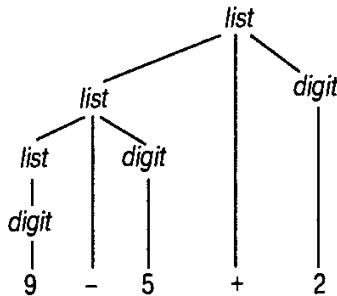


Рис. 2.2. Дерево разбора выражения 9-5+2 в соответствии с грамматикой из примера 2.1

Пример 2.3

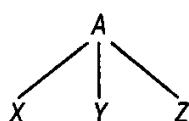
Еще один пример списков — последовательность инструкций, разделенных точками с запятыми; такую последовательность можно найти в Pascal, в блоке **begin-end**. Однако между токенами **begin** и **end** может и не быть инструкций. Разработку грамматики для блока **begin-end** можно начать со следующих продуктов:

$$\begin{aligned} \textit{block} &\rightarrow \textit{begin } \textit{opt_stmts } \textit{end} \\ \textit{opt_stmts} &\rightarrow \textit{stmt_list} \mid \epsilon \\ \textit{stmt_list} &\rightarrow \textit{stmt_list} ; \textit{stmt} \mid \textit{stmt} \end{aligned}$$

Заметьте, что правой частью для *opt_stmts* может являться ϵ , т.е. пустая строка символов. Таким образом, *opt_stmts* может быть заменено пустой строкой, и блок может оказаться строкой из двух токенов **begin end**. Обратите также внимание, что продукции для *stmt_list* аналогичны продукциям для *list* в примере 2.1 (точка с запятой вместо арифметической операции и с *stmt* вместо *digit*). В данном примере не представлены продукции для *stmt*, но далее вкратце будут рассмотрены продукции для различных инструкций, таких как инструкции присвоения, инструкции **if** и др. \square

Деревья разбора

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



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

1. Корень дерева помечен стартовым символом.
2. Каждый лист помечен токеном или ϵ .
3. Каждый внутренний узел представляет нетерминальный символ.
4. Если A является нетерминалом и помечает некоторый внутренний узел, а X_1, X_2, \dots, X_n — отметки его дочерних узлов, перечисленные слева направо, то $A \rightarrow X_1 X_2 \dots X_n$ — продукция. Здесь X_1, X_2, \dots, X_n могут представлять собой как терминальные, так и нетерминальные символы. В качестве специального случая продукции $A \rightarrow \epsilon$ соответствует узел A с единственным дочерним узлом ϵ .

Пример 2.4

На рис. 2.2 корневой узел помечен нетерминалом *list*, стартовым символом грамматики из примера 2.1. Дочерние узлы имеют отметки *list*, + и *digit*. Заметьте, что

$$list \rightarrow list + digit$$

является продукцией грамматики из примера 2.1. К левому дочернему узлу применен тот же шаблон, со знаком “минус” вместо знака “плюс”. Все три узла, помеченные как *digit*, имеют по одному дочернему узлу с метками-цифрами. \square

Листья дерева разбора, читаемые слева направо, образуют *крону* (yield) дерева, которая представляет собой строку, *выведенную*, или *порожденную* из нетерминального символа в корне дерева. На рис. 2.2 порожденная строка — 9-5+2. Здесь все листья показаны на одном, нижнем, уровне; в дальнейшем выравнивать листья деревьев таким образом не будем. Они должны рассматриваться в определенном порядке слева направо: если a и b — два дочерних узла одного родителя и узел a находится слева от b , то все потомки a будут находиться слева от любого потомка b .

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

Неоднозначность

Будьте предельно внимательны при рассмотрении структуры строки, соответствующей грамматике. Очевидно, что каждое дерево порождает единственную строку (путем считывания листьев этого дерева), однако для данной строки токенов грамматика может иметь более одного дерева. Такая грамматика называется *неоднозначной*. Чтобы убедиться в ее неоднозначности, достаточно найти строку токенов, которая имеет более одного дерева разбора. Поскольку такая строка обычно имеет не единственный смысл, следует использовать либо однозначные (непротиворечивые), либо неоднозначные грамматики с дополнительными правилами для разрешения неоднозначностей.

Пример 2.5

Предположим, что цифры и списки в примере 2.1 не различаются. Тогда грамматику можно записать следующим образом.

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

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

Однако из рис. 2.3 видно, что выражение типа $9-5+2$ теперь имеет больше одного дерева разбора. В данном случае два дерева разбора соответствуют двум вариантам расстановки скобок в выражении: $(9-5)+2$ и $9-(5+2)$. Это второе выражение дает в результате значение 2 вместо обычного 6. Грамматика в примере 2.1 не допускает такой интерпретации. \square

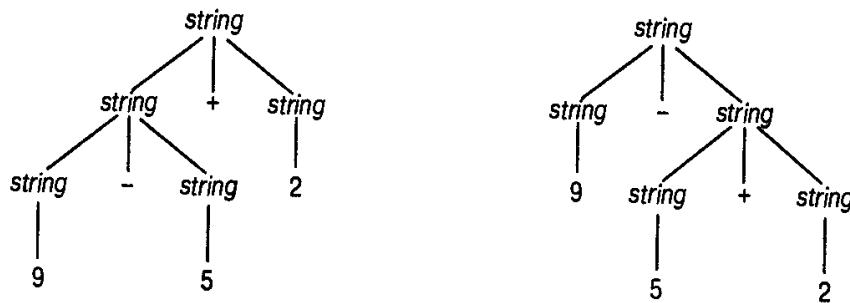


Рис. 2.3. Два дерева разбора для выражения $9-5+2$

Ассоциативность операторов

По соглашению $9+5+2$ эквивалентно $(9+5)+2$, а $9-5-2$ эквивалентно $(9-5)-2$. Когда operand типа 5 имеет операторы и слева, и справа, необходимо установить, какой именно оператор использует этот operand. Мы говорим, что оператор **+ левоассоциативен**, поскольку operand со знаками “плюс” с обеих сторон используется левым оператором. В большинстве языков программирования четыре арифметических оператора — сложение, вычитание, умножение и деление — левоассоциативны.

Некоторые распространенные операторы, например возвведение в степень, **правоассоциативны**. Другим примером правоассоциативного оператора может служить оператор присвоения ($=$) в C, где выражение $a=b=c$ трактуется как $a=(b=c)$.

Строки типа $a=b=c$ с правоассоциативным оператором генерируются следующей грамматикой.

right \rightarrow letter = *right* | letter
letter \rightarrow a | b | ... | z

Различия между деревьями разбора для левоассоциативных операторов типа “ $-$ ” и правоассоциативных операторов вроде “ $=$ ” показаны на рис. 2.4. Обратите внимание, что дерево разбора для $9-5-2$ растет вниз влево, в то время как дерево разбора для $a=b=c$ — вниз вправо.

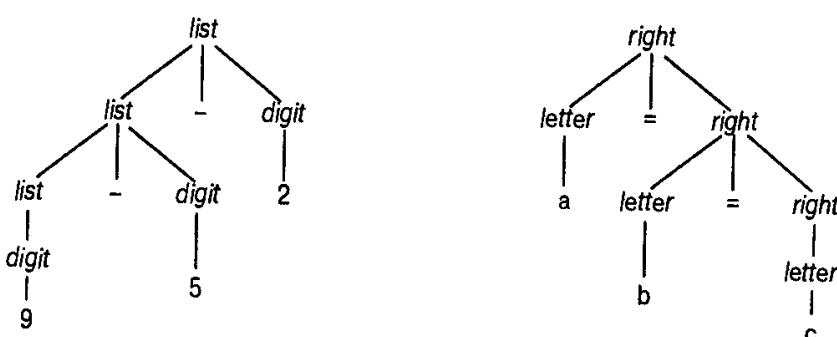


Рис. 2.4. Деревья разбора для лево- и правоассоциативных операторов

Приоритет операторов

Рассмотрим выражение $9+5*2$, которое можно интерпретировать как $(9+5)*2$ и $9+(5*2)$. Ассоциативность + и * не позволяет разрешить эту неоднозначность. Поэтому необходимо знать о приоритетах операторов.

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

Синтаксис выражений

Грамматика арифметических выражений может быть построена на основе таблицы, показывающей ассоциативность и приоритет операторов. Начнем с четырех основных арифметических операторов и таблицы приоритетов, представляющей операторы в порядке увеличения приоритетов (операторы, расположенные в одной строке, имеют одинаковый приоритет).

Левоассоциативные: + -

Левоассоциативные: * /

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

factor → digit | (*expr*)

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

term → *term* * *factor*
| *term* / *factor*
| *factor*

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

expr → *expr* + *term*
| *expr* - *term*
| *term*

В результате грамматика выглядит следующим образом.

expr → *expr* + *term* | *expr* - *term* | *term*
term → *term* * *factor* | *term* / *factor* | *factor*
factor → digit | (*expr*)

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

Синтаксис инструкций

В большинстве языков распознать инструкции можно с помощью ключевых слов. Например, все инструкции Pascal начинаются с ключевого слова, за исключением присвоения и вызова процедур. Некоторые из инструкций Pascal определяются следующей (неоднозначной) грамматикой, в которой токен *id* представляет идентификатор.

```
stmt → id := expr  
| if expr then stmt  
| if expr then stmt else stmt  
| while expr do stmt  
| begin opt_stmts end
```

Нетерминальный символ *opt_stmts* порождает список инструкций (возможно, пустой), разделенных точками с запятой, с использованием продукции из примера 2.3.

2.3. Синтаксически управляемая трансляция

Для трансляции конструкций языка программирования компилятору, помимо генерации кода, может потребоваться отследить множество различных параметров. Например, компилятору может понадобиться информация о типе конструкции, расположении первой инструкции в целевом коде или количестве сгенерированных инструкций. Таким образом, можно говорить о некоторых абстрактных *атрибутах*, связанных с языковыми конструкциями — типах, строках, адресах памяти и др.

В этом разделе будет рассмотрено *синтаксически управляемое определение* (syntax-directed definition), предназначенное для определения трансляции конструкций языка программирования. Синтаксически управляемое определение задает трансляцию конструкции в терминах атрибутов, связанных с ее синтаксическими компонентами. Позже синтаксически управляемые определения будут использованы для определения ряда трансляций в предварительной стадии компиляции.

Для определения трансляции воспользуемся более процедурно ориентированной записью, называемой схемой трансляции. В данной главе эти схемы используются для перевода инфиксных выражений в постфиксную запись. Более детальное обсуждение синтаксически управляемых определений и их реализации содержится в главе 5, “Синтаксически управляемая трансляция”.

Постфиксная запись

Постфиксная запись (postfix notation) выражения *E* может быть индуктивно определена следующим образом.

1. Если *E* является переменной или константой, то постфиксная запись *E* представляет собой *E*.
2. Если *E* — выражение вида *E₁ op E₂*, где *op* — произвольный бинарный оператор, то постфиксная запись *E* представляет собой *E₁' E₂' op*, где *E₁'* и *E₂'* — постфиксные записи для *E₁* и *E₂* соответственно.
3. Если *E* — выражение вида (*E₁*), то постфиксная запись для *E₁* представляет собой также и постфиксную запись для *E*.

Скобки в постфиксной записи не используются; последовательность и количество аргументов операторов допускают только один способ декодирования постфиксного вы-

ражения. Например, постфиксной записью для $(9-5)+2$ является $95-2+$, а для $9-(5+2)$ — $952+-$.

Синтаксически управляемые определения

Синтаксически управляемое определение (syntax-directed definition) использует контекстно-свободную грамматику для определения синтаксической структуры входа. С каждым символом грамматики оно связывает набор атрибутов, а с каждой продукцией — набор семантических правил для вычисления значений атрибутов, связанных с используемыми в продукциях символами. Грамматика и набор семантических правил составляют синтаксически управляемое определение.

Трансляция представляет собой отображение входного потока информации в выходной. Выход для каждого входа x определяется следующим образом. Вначале для x строится дерево разбора. Предположим, что в этом дереве узел n помечен символом грамматики X . Для указания значения атрибута a этого узла используем запись $X.a$. Значение $X.a$ в n вычисляется с помощью семантического правила для атрибута a , связанного с X -продукцией в узле n . Дерево разбора, указывающее значения атрибутов в каждом узле, называется *аннотированным* (annotated).

Синтезируемые атрибуты

Об атрибуте говорят, что он *синтезируемый*, если его значение в узле дерева разбора определяется значениями атрибутов в дочерних узлах. Синтезируемые атрибуты хороши тем, что вычисляются в процессе одного подхода снизу вверх по дереву разбора. В этой главе используются только синтезируемые атрибуты; “наследуемые” атрибуты рассматриваются в главе 5, “Синтаксически управляемая трансляция”.

Пример 2.6

Синтаксически управляемое определение для трансляции выражений, состоящих из цифр, разделенных знаками “плюс” и “минус”, в постфиксную форму показано на рис. 2.5. Связанный с каждым нетерминалом строковый атрибут t представляет постфиксную запись для выражения, порожденного этим нетерминалом в дереве разбора.

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

Рис. 2.5. Синтаксически управляемое определение для трансляции инфиксной записи в постфиксную

Постфиксная форма цифры — это сама цифра. Итак, семантическое правило продукции $term \rightarrow 9$ определяет, что при ее использовании $term.t$ в узле дерева разбора представляет собой просто 9. При использовании продукции $expr \rightarrow term$ значение $term.t$ становится значением $expr.t$.

Продукция $expr \rightarrow expr_1 + term$ порождает выражение, содержащее оператор “плюс” (индекс в $expr_1$ позволяет отличить левую часть выражения от правой). Левый операнд оператора сложения — $expr_1$, а правый — $term$. Семантическое правило данной продукции

$expr.t := expr_1.t \parallel term.t \parallel '+'$

определяет значение атрибута $expr.t$ путем конкатенации постфиксных форм $expr_1.t$ и $term.t$ левого и правого operandов, к которым затем добавляется знак “плюс”. Оператор “ \parallel ” в семантических правилах представляет конкатенацию строк.

На рис. 2.6 представлено аннотированное дерево разбора, соответствующее дереву на рис. 2.2. Значение атрибута t в каждом узле вычислялось с использованием семантического правила, связанного с продукцией данного узла. Значение атрибута в корне представляет собой постфиксную запись строки, порождаемой деревом разбора. □

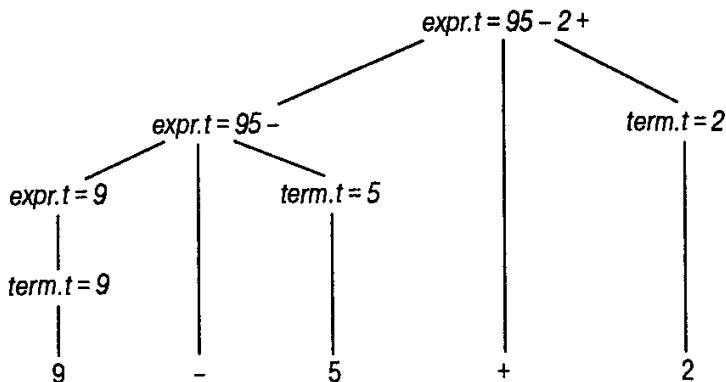


Рис. 2.6. Значения атрибутов в узлах дерева разбора

Пример 2.7

Рассмотрим пример с роботом, которому можно указать перемещения из исходного положения на один шаг на восток (east), север (north), запад (west) или юг (south). Последовательность таких инструкций порождается следующей грамматикой.

$seq \rightarrow seq\ instr\mid begin$
 $instr \rightarrow east\mid north\mid west\mid south$

Изменения положения робота при обработке входной последовательности

begin west south east east east north north

приведены на рис. 2.7

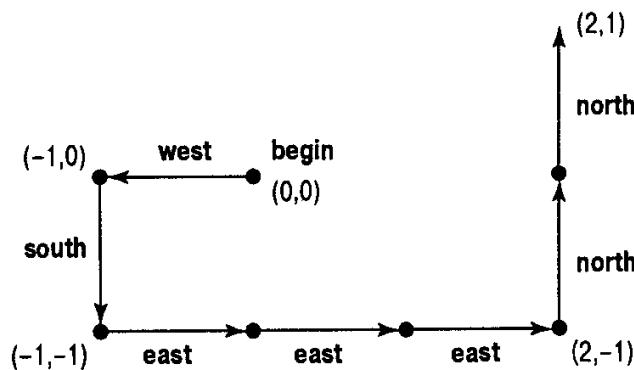


Рис. 2.7. Траектория перемещения робота

На рисунке положение робота определяется парой значений (x,y) , где x и y — количество шагов от стартовой позиции на восток и на север соответственно (если значение x отрицательно, то робот находится западнее стартовой позиции, если y отрицательно — южнее).

Построим синтаксически управляемое определение для трансляции последовательности инструкций в позицию робота. Для отслеживания положения робота в процессе порождения последовательности инструкций нетерминалом seq будем использовать два атрибута, $seq.x$ и $seq.y$. Изначально, при генерации **begin**, $seq.x$ и $seq.y$ принимают значения 0, как показано в левом внутреннем узле дерева для последовательности **begin west south** (рис. 2.8).

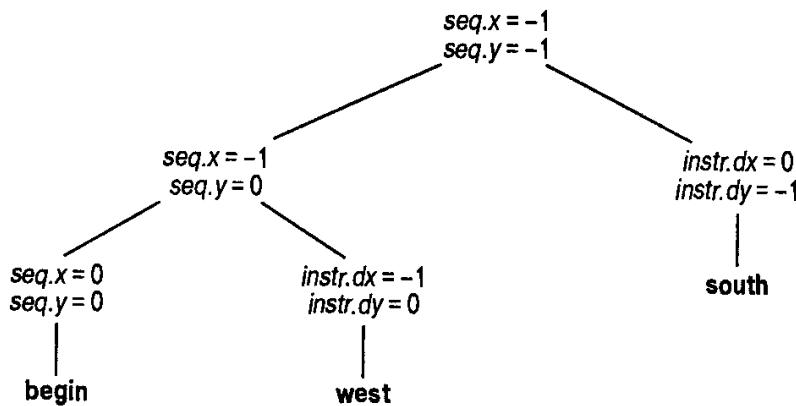


Рис. 2.8. Аннотированное дерево разбора для последовательности **begin west south**

Изменение положения с учетом отдельных инструкций определяется атрибутами $instr.dx$ и $instr.dy$. Например, если $instr$ принимает значение **west**, то $instr.dx = -1$, а $instr.dy = 0$. Предположим, что последовательность seq формируется следующей за последовательностью seq_1 новой инструкцией $instr$. В таком случае новое положение робота определяется правилами

$$\begin{aligned} seq.x &:= seq_1.x + instr.dx \\ seq.y &:= seq_1.y + instr.dy \end{aligned}$$

Синтаксически управляемое определение для трансляции последовательности инструкций в позицию робота показано на рис. 2.9. \square

Продукция	Семантические правила
$seq \rightarrow begin$	$seq.x := 0$ $seq.y := 0$
$seq \rightarrow seq_1 instr$	$seq.x := seq_1.x + instr.dx$ $seq.y := seq_1.y + instr.dy$
$instr \rightarrow east$	$instr.dx := 1$ $instr.dy := 0$
$instr \rightarrow north$	$instr.dx := 0$ $instr.dy := 1$
$instr \rightarrow west$	$instr.dx := -1$ $instr.dy := 0$
$instr \rightarrow south$	$instr.dx := 0$ $instr.dy := -1$

Рис. 2.9. Синтаксически управляемое определение положения робота

Рекурсивный обход дерева

Синтаксически управляемое определение не задает конкретный порядок вычисления атрибутов в дереве разбора. Пригоден любой порядок, который определяет атрибуты *a* после всех атрибутов, от которых зависит *a*. Вообще говоря, некоторые атрибуты нужно вычислять при первом же посещении узла, а какие-то — во время или после прохода по дочерним узлам. Детальнее различные способы вычисления рассматриваются в главе 5, “Синтаксически управляемая трансляция”.

В данной главе все трансляции могут быть реализованы применением семантических правил вычисления атрибутов в дереве разбора по определенному порядку. Обход (traversal) дерева начинается с корня и состоит в посещении каждого узла дерева в определенном порядке. В данной главе семантические правила применяются с использованием рекурсивного обхода, показанного на рис. 2.10. Он начинается в корне дерева и рекурсивно проходит в порядке слева направо по всем дочерним узлам данного узла, как показано на рис. 2.11. Семантические правила в данном узле применяются после посещения всех его потомков. Этот обход имеет также название “сперва вглубь”, или “в глубину” (depth-first), поскольку в первую очередь посещаются еще не пройденные дочерние узлы.

```
procedure visit(n: node);
begin
    for каждый дочерний узел m узла n в порядке слева направо
do
    visit(m);
    применить семантические правила в узле n
end
```

Рис. 2.10. Рекурсивный обход дерева “в глубину”

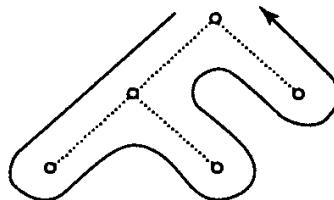


Рис. 2.11. Пример рекурсивного обхода дерева “в глубину”

Схемы трансляции

В этой главе для определения трансляции будет использована процедурная спецификация. *Схема трансляции* является контекстно-свободной грамматикой, в которой программные фрагменты, называемые *семантическими действиями* (semantic actions), вставлены в правые части продукции. Схема трансляции подобна синтаксически управляемому определению, однако здесь явно указан порядок применения семантических правил. Выполнение семантических действий указывается с помощью фигурных скобок в правой части продукции:

rest → + *term* { *print*('+') } *rest*₁

Схема трансляции порождает выход для каждого предложения *x* грамматики путем выполнения действий в том порядке, в каком они появляются при обходе в глубину де-

рева разбора для x . Рассмотрим дерево разбора с узлом $rest$, представляющее указанную продукцию. Действие $\{ print('+') \}$ будет выполнено после обхода поддерева для элемента $term$, но до посещения узла $rest_1$.

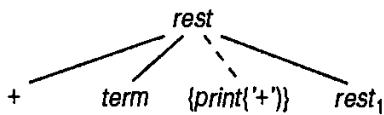


Рис. 2.12. Дополнительные листы дерева для семантических действий

При изображении дерева разбора для схемы трансляции действия для данной конструкции указываются в виде дополнительного дочернего узла, соединенного с родительским пунктирной линией (рис. 2.12). Узел семантического действия не имеет дополнительных узлов, так что действия выполняются сразу же при появлении такого узла при обходе дерева.

Вывод результатов трансляции

В этой главе семантические действия в схемах трансляции будут записывать выход процесса трансляции в файл. Например, мы транслируем $9-5+2$ в $95-2+$ однократным выводом каждого символа из $9-5+2$ без использования какой-либо дополнительной памяти для трансляции подвыражений. При генерации вывода таким инкрементальным путем особое значение имеет порядок выхода символов.

Заметим, что упомянутые синтаксически управляемые определения имеют одно важное свойство. Стока, представляющая трансляцию нетерминала в левой части каждой продукции, является конкатнацией трансляций нетерминалов справа в том же порядке, что и в продукции, возможно, с некоторыми дополнительными строками между ними. Синтаксически управляемое определение, обладающее таким свойством, называется *простым* (simple). Рассмотрим первую продукцию и семантическое правило из синтаксически управляемого определения на рис. 2.5.

ПРОДУКЦИЯ	СЕМАНТИЧЕСКОЕ ПРАВИЛО	
$expr \rightarrow expr_1 + term$	$expr.t := expr_1.t \parallel term.t \parallel '+'$	(2.6)

Здесь трансляция $expr.t$ представляет собой конкатенацию трансляций $expr_1$ и $term$, за которыми следует символ $+$. Заметим, что $expr_1$ появляется в правой части продукции перед $term$.

Дополнительная строка появляется между $term.t$ и $rest_1.t$.

ПРОДУКЦИЯ	СЕМАНТИЧЕСКОЕ ПРАВИЛО	
$rest \rightarrow term\ rest_1$	$rest.t := term.t \parallel '+' \parallel rest_1.t$	(2.7)

Однако и в этом случае нетерминал $term$ располагается в правой части перед $rest_1$.

Простые синтаксически управляемые определения могут быть реализованы схемами трансляции, в которых действия выводят дополнительные строки в порядке их появления в определении. Действия в следующих продукциях выводят дополнительные строки в (2.6) и (2.7) соответственно.

$$\begin{aligned} expr \rightarrow & \quad expr_1 + term \{ print('+') \} \\ rest \rightarrow & \quad + term \{ print('+') \} rest_1 \end{aligned}$$

Пример 2.8

Рис. 2.5 содержит простое определение для трансляции выражения в постфиксный вид. Схема трансляции, порождаемая этим определением, приведена на рис. 2.13, а дерево разбора с действиями для выражения $9 - 5 + 2$ — на рис. 2.14. Обратите внимание, что хотя на рис. 2.6 и 2.14 представлено одно и то же отображение входящего потока в выходящий, трансляция в этих случаях строится по-разному. На рис. 2.6 выход присоединяется к корню дерева разбора, в то время как на рис. 2.14 производится инкрементальный вывод.

$expr \rightarrow expr + term$	$\{ print('+') \}$
$expr \rightarrow expr - term$	$\{ print(' - ') \}$
$expr \rightarrow term$	
$term \rightarrow 0$	$\{ print('0') \}$
$term \rightarrow 1$	$\{ print('1') \}$
\dots	
$term \rightarrow 9$	$\{ print('9') \}$

Рис. 2.13. Действия, транслирующие выражения в постфиксную запись

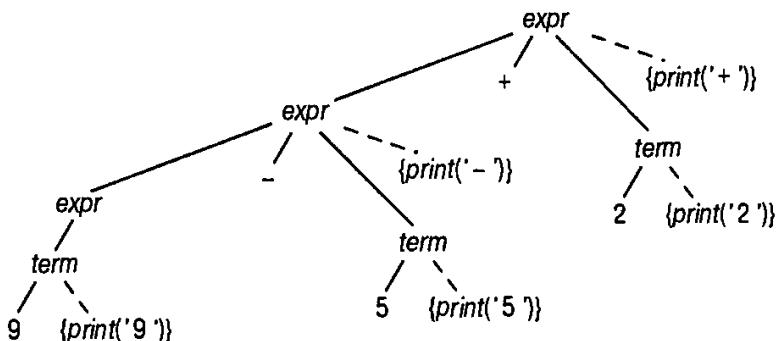


Рис. 2.14. Действия, транслирующие $9 - 5 + 2$ в $95 - 2 +$

На рис. 2.14 корень представляет первую продукцию рис. 2.13. При рекурсивном обходе вглубь вначале выполняем все действия в поддереве левого операнда $expr$ (при обходе левого поддерева корня). Затем переходим в лист $+$, где действия отсутствуют, осуществляя все действия в поддереве правого операнда $term$ и, наконец, выполняем семантическое действие $\{ print(+) \}$ в дополнительном узле.

Продукции для $term$ имеют в правой части только цифры, которые выводятся соответствующими действиями. Продукция $expr \rightarrow term$ не осуществляет никакого вывода, а действием для первых двух продуктов является вывод соответствующего оператора. Действия, выполняемые при рекурсивном обходе дерева разбора (рис. 2.14) вглубь, приводят к выводу $95 - 2 +$. \square

В целом, большинство методов разбора обрабатывают входной поток слева направо “жадным” способом, т.е. перед чтением очередного входного токена строится наибольшее возможное дерево разбора. В простой схеме трансляции (порождаемой простым синтаксически управляемым определением) действия также выполняются слева направо. Следовательно, для реализации простой схемы трансляций можно выполнять семантические действия в процессе разбора и строить дерево разбора вообще нет необходимости.

2.4. Разбор

В процессе разбора определяется, может ли некоторая строка токенов быть сгенерирована данной грамматикой. При обсуждении этого вопроса рассматривается построение дерева разбора, хотя в действительности оно может и не создаваться компилятором. Однако анализатор должен быть способен построить такое дерево, иначе невозможно гарантировать корректность трансляции.

В этом разделе представлены методы разбора, используемые для построения синтаксически управляемых трансляторов. В следующем разделе приведена законченная программа на языке программирования C, реализующая схему трансляции на рис. 2.13. Альтернативой является использование программного инструментария для генерации транслятора непосредственно на основе схемы трансляции (описание такого инструментария вы найдете в разделе 4.9).

Анализатор может быть построен для любой грамматики, хотя используемые на практике грамматики имеют специальный вид. Для любой контекстно-свободной грамматики существует анализатор, который производит разбор строки, состоящей из n токенов, за время, не превышающее $O(n^3)$. Однако такое кубическое время разбора слишком велико. Для конкретного языка программирования можно построить грамматику, которая будет анализироваться намного быстрее. Для разбора почти всех встречающихся на практике языков достаточно линейного алгоритма. Анализаторы языков программирования почти всегда делают один проход входного потока слева направо, заглядывая вперед на один токен.

Большинство методов разбора делится на два типа — нисходящие (*сверху вниз*, top-down) и восходящие (*снизу вверх*, bottom-up). Это связано с порядком, в котором строятся узлы дерева разбора. При первом методе построение начинается от корня по направлению к листьям, в то время как при втором методе — от листьев по направлению к корню. Популярность анализаторов “сверху вниз” обусловлена тем фактом, что построить эффективный анализатор вручную проще с использованием методов “сверху вниз”. Однако разбор “снизу вверх” может работать с большим классом грамматик и схем трансляции, так что программный инструментарий для генерации анализаторов непосредственно по грамматикам тяготеет к восходящим методам.

Нисходящий анализ

Для ознакомления с нисходящим анализом начнем с рассмотрения грамматики, которая хорошо подходит для методов разбора этого типа (позже будет рассмотрено построение нисходящих анализаторов в целом). Приведенная ниже грамматика генерирует подмножество типов языка Pascal. Для “...” воспользуемся токеном **dotdot**, чтобы подчеркнуть, что данная последовательность символов трактуется как единое целое.

<i>type</i>	\rightarrow	<i>simple</i>
		\uparrow id
		array [<i>simple</i>] of <i>type</i>
<i>simple</i>	\rightarrow	integer
		char
		num dotdot num

Построение дерева разбора сверху вниз начинается с корня, помеченного стартовым нетерминалом, и осуществляется многократным выполнением следующих двух шагов (рис. 2.15).

1. В узле n , помеченному нетерминалом A , выбираем одну из продукции для A и строим дочерние узлы для символов из правой части продукции.
2. Находим следующий узел, в котором должно быть построено поддерево.

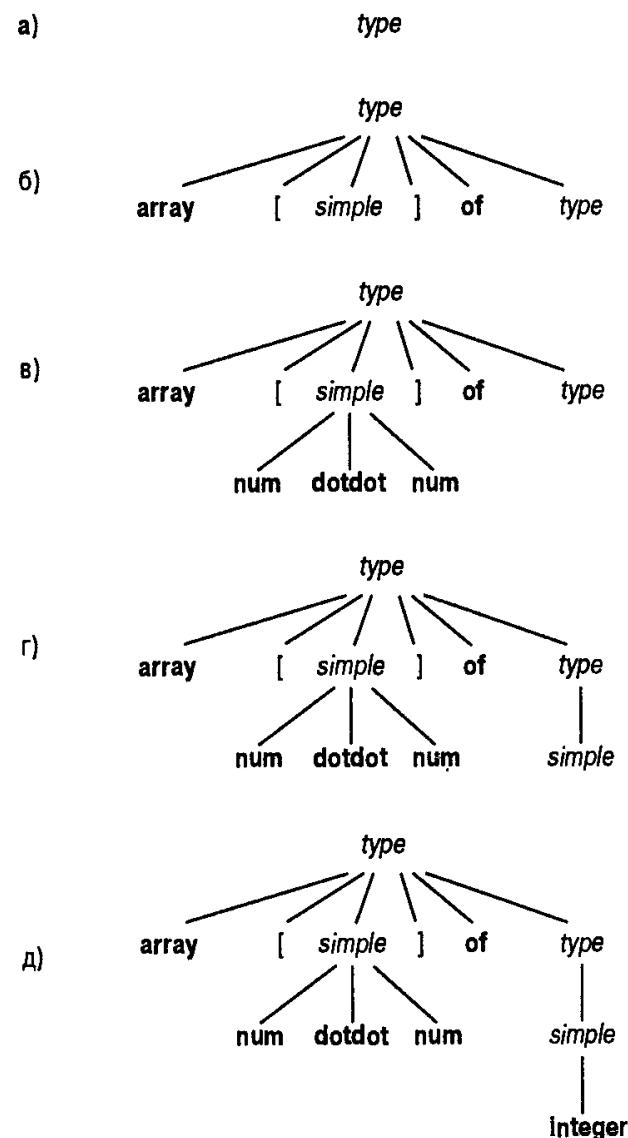


Рис. 2.15. Шаги построения дерева разбора сверху вниз

В некоторых грамматиках описанные шаги могут быть реализованы во время однократного сканирования входной строки слева направо. Текущий токен называют *сканируемым символом*, или “предсимволом” (lookahead symbol — дословно: предвиденный символ. — Прим. перев.). Изначально текущим символом является первый, т.е. самый левый токен входной строки. На рис. 2.16 проиллюстрирован анализ строки

array [num dotdot num] of integer

Изначально текущим сканируемым символом является токен **array**, а известная часть дерева разбора состоит из корня, помеченного как стартовый нетерминал **type** (рис. 2.16a). Задача состоит в построении остальной части дерева разбора таким образом, чтобы строка, порожденная деревом разбора, совпадала с входной строкой.

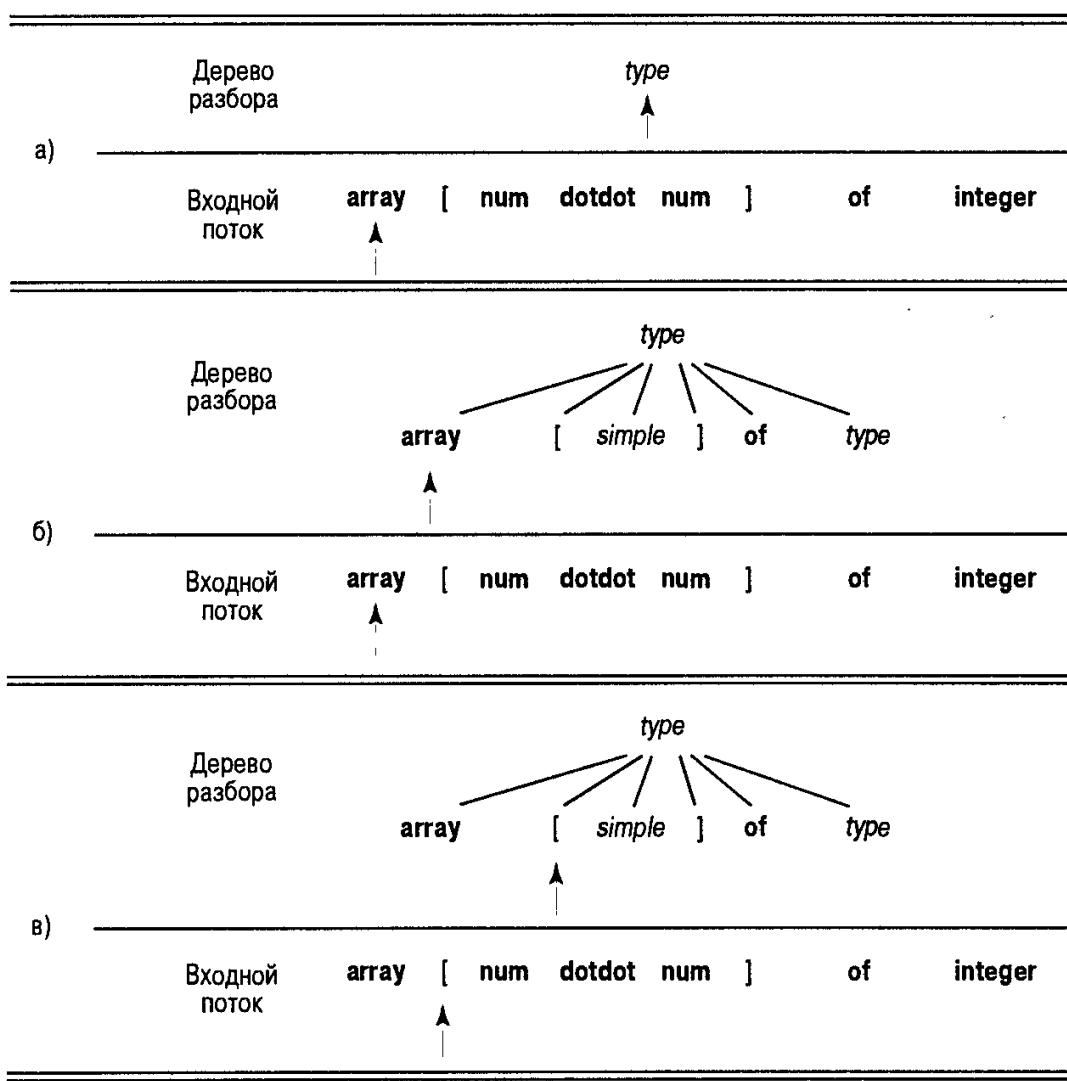


Рис. 2.16. Нисходящий анализ при сканировании входного потока слева направо

Чтобы соответствовать входной строке, нетерминал *type* (рис. 2.16 a) должен выводить строку, начинающуюся со сканируемого символа *array*. В грамматике (2.8) для *type* имеется только одна продукция, которая способна породить такую строку. Нужно выбрать эту продукцию и построить дочерние по отношению к корню узлы, помеченные символами из правой части продукции.

На каждом из трех “снимков” (рис. 2.16) имеются стрелки, указывающие сканируемый символ входного потока и рассматриваемый узел дерева разбора. После построения всех дочерних узлов текущего узла переходим к рассмотрению самого левого дочернего узла. На рис. 2.16 b показан именно этот этап, когда построены дочерние по отношению к корню узлы и мы переходим к рассмотрению самого левого дочернего узла, а именно — *array*.

Если рассматриваемый узел дерева разбора соответствует терминалу, а этот терминал совпадает со сканируемым символом, можно продвигаться вперед как по дереву разбора, так и по входному потоку. Следующий токен из входного потока становится текущим сканируемым символом, и мы приступаем к рассмотрению следующего дочернего узла. На рис. 2.16 c стрелка в дереве разбора перемещается к следующему дочернему по отношению к корню узлу, а стрелка во входном потоке — к следующему токену [. После следующего перемещения стрелка в дереве разбора будет указывать на узел, помеченный как нетерминал *simple*. При рассмотрении узла, помеченного нетерминалом, процесс выбора продукции повторяется.

В целом при выборе продукции для нетерминала может использоваться метод проб и ошибок. Это означает, что можно применить продукцию и, в случае неуспешной попытки, выполнить откат, а затем перейти к другим продукциям для данного нетерминалного символа. Попытка использования продукции считается неуспешной, если после нее невозможно завершить дерево, соответствующее входной строке (однако при так называемом предиктивном анализе откат отсутствует).

Предиктивный анализ

Анализ методом рекурсивного спуска (recursive-descent parsing) представляет собой способ нисходящего синтаксического анализа, при котором выполняется ряд рекурсивных процедур для обработки входного потока (процедура связана с каждым нетерминалным символом грамматики). Здесь будет рассмотрен специальный вид анализа методом рекурсивного спуска, именуемый предиктивным (или предсказывающим) анализом, при котором сканируемый символ однозначно определяет процедуру, выбранную для каждого нетерминала. Последовательность процедур, вызываемых при обработке входного потока, неявно определяет его дерево разбора.

```
procedure match(t: token);
begin
    if lookahead = t then
        lookahead := nexttoken
    else error
end;

procedure type;
begin
    if lookahead is in { integer, char, num } then
        simple
    else if lookahead = '↑' then begin
        match('↑'); match(id)
    end
    else if lookahead = array then begin
        match(array); match('[');
        simple; match(']'); match(of); type.
    end
    else error
end;

procedure simple;
begin
    if lookahead = integer then
        match(integer)
    else if lookahead = char then
        match(char)
    else if lookahead = num then begin
        match(num); match(dotdot); match(num)
    end
    else error
end;
```

Рис. 2.17. Псевдокод предиктивного анализатора

Предиктивный анализатор (рис. 2.17) состоит из процедур для нетерминальных символов *type* и *simple* грамматики (2.8) и дополнительной процедуры *match*. В данном случае *match* используется для упрощения кода процедур *type* и *simple*; эта процедура переходит к следующему входному токену, если ее аргумент *t* совпадает со сканируемым символом. Таким образом, *match* изменяет переменную *lookahead*, которая представляет собой текущий сканируемый входной токен.

Разбор начинается с вызова процедуры для стартового нетерминала грамматики *type*. При том же входном потоке, что и на рис. 2.16, *lookahead* изначально принимает значение, равное первому токену входной строки — **array**. Процедура *type* выполняет код

$$\text{match(array); } \text{match('['); simple; } \text{match(']'); } \text{match(of); type} \quad (2.9)$$

который соответствует правой части продукции

type → **array** [*simple*] of *type*

Заметим, что каждый терминал в правой части соответствует сканируемому символу, а каждый нетерминал правой части приводит к вызову связанной с ним процедуры.

При входном потоке, показанном на рис. 2.16, после того как выявлено соответствие токенов **array** и **[**, текущим сканируемым символом становится **num**. В этот момент вызывается процедура *simple*, в которой выполняется код

match(num); match(dotdot); match(num)

Текущий сканируемый символ позволяет выбрать используемую продукцию. Если правая часть продукции начинается с токена, то продукция может использоваться, когда текущий сканируемый символ представляет собой соответствующий токен. А теперь рассмотрим правую часть продукции, начинающуюся с нетерминального символа.

type → *simple* (2.10)

Эта продукция используется тогда, когда текущий сканируемый символ может быть порожден из *simple*. Предположим, что в процессе выполнения фрагмента кода (2.9) в момент вызова процедуры *type* текущий сканируемый символ — **integer**. Для нетерминального символа *type* нет ни одной продукции, начинающейся с токена **integer**. Однако такая продукция имеется для нетерминала *simple*, поэтому в процедуре *type* при значении *lookahead*, равном **integer**, выполняется вызов процедуры *simple*.

Предиктивный анализ базируется на информации о том, какой первый символ может быть порожден правой частью продукции. Более строго, пусть α — правая часть продукции для нетерминала *A*. Определим $\text{FIRST}(\alpha)$ как множество токенов, которые могут появиться в качестве первого символа одной или нескольких строк, полученных из α . Если α представляет собой ϵ или может порождать ϵ , то ϵ также входит в $\text{FIRST}(\alpha)$ ³. Например,

$$\begin{aligned}\text{FIRST}(\text{simple}) &= \{ \text{integer}, \text{char}, \text{num} \} \\ \text{FIRST}(\uparrow \text{id}) &= \{ \uparrow \} \\ \text{FIRST}(\text{array} [\text{simple}] \text{ of type }) &= \{ \text{array} \}\end{aligned}$$

³ Продукция с ϵ в правой части усложняет определение первых символов, порождаемых нетерминалами. Например, если из нетерминала *B* можно вывести пустую строку и имеется продукция $A \rightarrow BC$, то первый символ, порождаемый *C*, может быть первым символом, порождаемым *A*. Если *C* также порождает ϵ , то и $\text{FIRST}(A)$, и $\text{FIRST}(BC)$ содержат ϵ .

На практике многие правые части продукции начинаются с токенов, что упрощает построение множеств FIRST. Алгоритм для построения FIRST приводится в разделе 4.4.

Множества FIRST следует рассматривать, если имеются две продукции $A \rightarrow \alpha$ и $A \rightarrow \beta$. Анализ методом рекурсивного спуска без отката требует, чтобы множества $\text{FIRST}(\alpha)$ и $\text{FIRST}(\beta)$ были непересекающимися. Тогда текущий сканируемый символ может использоваться для принятия решения, какая из продукции будет применена. Если сканируемый символ принадлежит множеству $\text{FIRST}(\alpha)$, используется продукция α ; в противном случае, когда сканируемый символ принадлежит множеству $\text{FIRST}(\beta)$, применяется продукция β .

Использование ϵ -продукций

Продукции с ϵ в правой части требуют специальной трактовки. Анализатор методом рекурсивного спуска использует такую продукцию в качестве продукции по умолчанию, когда не может быть использована никакая другая. Рассмотрим

$$\begin{aligned}stmt &\rightarrow \text{begin } opt_stmts \text{ end} \\opt_stmts &\rightarrow stmt_list \mid \epsilon\end{aligned}$$

Если при анализе opt_stmts сканируемый символ не принадлежит множеству $\text{FIRST}(stmt_list)$, то используется ϵ -продукция. Такой выбор совершенно верен, когда сканируемый символ — `end`. Любой другой символ приведет к ошибке, обнаруживаемой в процессе анализа $stmt$.

Создание предиктивного анализатора

Предиктивный анализатор представляет собой программу, содержащую процедуры для каждого нетерминального символа. Каждая такая процедура решает две задачи.

1. Принимает решение, какая продукция будет использоваться, исходя из текущего сканируемого символа. Если сканируемый символ принадлежит множеству $\text{FIRST}(\alpha)$, применяется продукция с правой частью α . Продукция с ϵ в правой части используется в случае, когда текущий сканируемый символ не принадлежит множеству FIRST для любой другой правой части. В случае конфликта двух правых частей продукции для некоторого сканируемого символа этот метод анализа неприменим для рассматриваемой грамматики.
2. Использует продукцию, имитируя ее правую часть. Нетерминал приводит к вызову процедуры, соответствующей этому нетерминалу, а токен, совпадающий с текущим сканируемым символом, — к чтению следующего токена из входного потока. Если в какой-то момент токен продукции не совпадает со сканируемым символом, вызывается процедура обработки ошибки. На рис. 2.17 представлен результат применения этих правил к грамматике (2.8).

Так же, как схема трансляции образуется расширением грамматики, синтаксически управляемый транслятор может быть получен расширением предиктивного анализатора. Алгоритм для решения этой задачи приведен в разделе 5.5. Поскольку схемы трансляции, реализуемые в данной главе, не связывают атрибуты с нетерминалами, достаточно следующего ограниченного построения.

1. Построить предиктивный анализатор, игнорируя действия в продукциих.

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

Такой транслятор будет построен в следующем разделе.

Левая рекурсия

Анализатор, работающий методом рекурсивного спуска, может оказаться в состоянии зацикливания при работе с леворекурсивной продукцией типа

$$expr \rightarrow expr + term$$

в которой левый символ правой части идентичен нетерминалу в левой части продукции. Предположим, что процедура для $expr$ принимает решение о применении этой продукции. Так как правая часть продукции начинается с $expr$, вызывается та же процедура, и анализатор зацикливается. Заметьте, что текущий сканируемый символ изменяется только в том случае, когда он совпадает с терминалом в правой части продукции. Поскольку продукция начинается с нетерминала $expr$, между рекурсивными вызовами процедуры не происходит никаких изменений во входном потоке, что и приводит к бесконечному циклу.

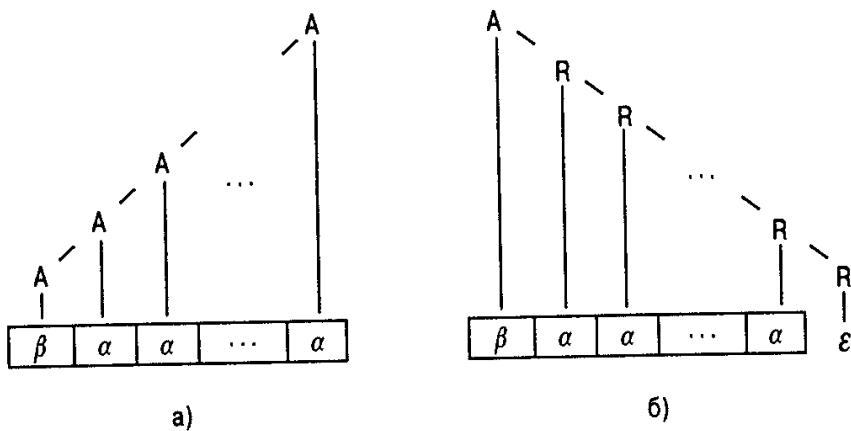


Рис. 2.18. Лево- и праворекурсивные способы генерации строки

Леворекурсивных продукции можно избежать, переписав продукцию, доставляющие неприятности. Рассмотрим нетерминал A с двумя продукциями

$$A \rightarrow A\alpha | \beta$$

где α и β представляют собой последовательности терминалов и нетерминалов, которые не начинаются с A . Например, в

$$\begin{aligned} expr &\rightarrow expr + term | term \\ A &= expr, \alpha = + term \text{ и } \beta = term. \end{aligned}$$

Нетерминал A леворекурсивен, поскольку продукция $A \rightarrow A\alpha$ в качестве крайнего левого символа правой части содержит A . При повторном применении этой продукции создается последовательность α справа от A , как показано на рис. 2.18а. Когда, наконец, A будет заменено на β , получим β , за которой будет следовать нуль или несколько α .

Тот же результат можно получить, как показано на рис. 2.18б, записав продукцию для A следующим образом.

$$\begin{aligned} A &\rightarrow \beta R \\ R &\rightarrow \alpha R \mid \epsilon \end{aligned} \tag{2.11}$$

Здесь R — новый нетерминал. Продукция $R \rightarrow \alpha R$ представляет собой *правую рекурсию*, поскольку продукция для R в качестве крайнего справа символа правой части содержит R . Праворекурсивные продукции приводят к деревьям, которые растут вниз вправо, как показано на рис. 2.18б. Такие деревья затрудняют трансляцию выражений, содержащих левоассоциативные операторы, например оператор вычитания. Однако в следующем разделе мы увидим, что корректной трансляции выражений в постфиксную запись можно добиться аккуратным построением схемы трансляции, основанной на праворекурсивной грамматике.

В главе 4, “Синтаксический анализ”, мы рассмотрим более общие виды левой рекурсии и покажем, как устранить левую рекурсию из грамматики.

2.5. Транслятор для простых выражений

С помощью технологий, описанных в предыдущих разделах, приступим к разработке синтаксически управляемого транслятора в виде рабочей программы на С, которая будет транслировать арифметические выражения в постфиксный вид. Для того чтобы не усложнять программу, начнем с выражений, состоящих из цифр, разделенных плюсами и минусами. В дальнейшем этот язык будет дополнен числами, идентификаторами и другими операторами. Поскольку выражения встречаются в качестве конструкций практически во всех языках, имеет смысл детально изучить их трансляцию.

<i>expr</i>	\rightarrow	<i>expr + term</i>	{ <i>print('+')</i> }
<i>expr</i>	\rightarrow	<i>expr - term</i>	{ <i>print('-')</i> }
<i>expr</i>	\rightarrow	<i>term</i>	
<i>term</i>	\rightarrow	0	{ <i>print('0')</i> }
<i>term</i>	\rightarrow	1	{ <i>print('1')</i> }
		...	
<i>term</i>	\rightarrow	9	{ <i>print('9')</i> }

Рис. 2.19. Начальная спецификация транслятора инфиксной записи в постфиксную

Схема синтаксически управляемой трансляции часто может служить в качестве определения транслятора. В данном случае мы используем схему, приведенную на рис. 2.19 (повторяющем рис. 2.13). Зачастую грамматика, лежащая в основе данной схемы, требует модификации перед тем, как сможет быть разобрана предиктивным анализатором. В частности, грамматика, лежащая в основе схемы, представленной на рис. 2.19, леворекурсивна, но, как видно из предыдущего раздела, предиктивный анализатор не может работать с леворекурсивной грамматикой. Устранив леворекурсивность, мы получим грамматику, пригодную для использования предиктивного анализатора, работающего методом рекурсивного спуска.

Абстрактный и конкретный синтаксис

Полезно начать рассмотрение трансляции входной строки с *дерева абстрактного синтаксиса*, или просто *синтаксического дерева*, в котором каждый узел представляет оператор, а дочерние узлы — операнды. В отличие от дерева абстрактного синтаксиса,

дерево разбора называется *деревом конкретного синтаксиса*, а лежащая в его основе грамматика — *конкретным синтаксисом языка*.

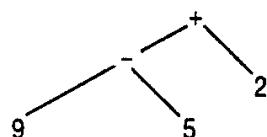


Рис. 2.20. Синтаксическое дерево для выражения 9-5+2

Например, на рис. 2.20 показано синтаксическое дерево для выражения 9-5+2. Поскольку + и - имеют один и тот же приоритет, а операторы с равным приоритетом выполняются слева направо, в дереве 9-5 сгруппировано в качестве подвыражения. Сравнивая рис. 2.20 с рис. 2.2, заметим, что синтаксическое дерево связывает оператор с внутренним узлом, а не представляет его в виде одного из дочерних узлов.

Желательно, чтобы схема трансляции базировалась на грамматике, деревья разбора которой были бы по возможности ближе к синтаксическим деревьям. Группирование подвыражений грамматикой (рис. 2.19) похоже на их группирование в синтаксических деревьях. К сожалению, грамматика на рис. 2.19 леворекурсивна, а следовательно, предиктивный анализ к ней неприменим. Итак, с одной стороны, нам нужна грамматика, упрощающая процесс разбора, с другой — для простоты трансляции требуется совершенно другая грамматика. Очевидное разрешение этого противоречия состоит в устранении леворекурсивности грамматики. Однако, как показано в следующем примере, это нужно делать предельно аккуратно и осторожно.

Пример 2.9

Приведенная ниже грамматика не подходит для трансляции выражений в постфиксный вид, хотя она порождает в точности тот же язык, что и грамматика на рис. 2.19, и может использоваться для анализа методом рекурсивного спуска.

$$\begin{aligned} \textit{expr} &\rightarrow \textit{term} \textit{rest} \\ \textit{rest} &\rightarrow + \textit{expr} \mid - \textit{expr} \mid \epsilon \\ \textit{term} &\rightarrow 0 \mid 1 \mid \dots \mid 9 \end{aligned}$$

Проблема этой грамматики в том, что операнды операторов, порождаемых продукциями $\textit{rest} \rightarrow + \textit{expr}$ и $\textit{rest} \rightarrow - \textit{expr}$, не очевидны из самих продукции. Ни один из приведенных ниже вариантов получения трансляции $\textit{rest}.t$ из трансляции $\textit{expr}.t$ не является приемлемым.

$$\textit{rest} \rightarrow - \textit{expr} \{ \textit{rest}.t := '-' \parallel \textit{expr}.t \} \quad (2.12)$$

$$\textit{rest} \rightarrow - \textit{expr} \{ \textit{rest}.t := \textit{expr}.t \parallel '-' \} \quad (2.13)$$

(Здесь приведены продукции и семантические действия только для оператора “минус”.) Трансляция 9-5 в постфиксную форму дает 95-, однако, если использовать действие из (2.12), знак “-” появляется перед $\textit{expr}.t$ и 9-5 не изменяется после трансляции.

При использовании (2.13) и аналогичного правила для знака “+” операторы последовательно перемещаются к правому концу, и 9-5+2 транслируется в 952+- (в то время как корректной является трансляция 95-2+). □

Адаптация схемы трансляции

Технология устранения леворекурсивности, схематически изображенная на рис. 2.18, может быть применена к продукциям, содержащим семантические действия. Воспользуемся преобразованиями из раздела 5.5 при работе с синтезируемыми атрибутами. Эта технология преобразовывает продукцию $A \rightarrow A\alpha | A\beta | \gamma R$

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

При работе с семантическими действиями мы переносим их в преобразованные продукции. В данном случае $A = \text{expr}$, $\alpha = + \text{term} \{ \text{print}('+') \}$, $\beta = - \text{term} \{ \text{print}('-) \}$ и $\gamma = \text{term}$ и описанное выше преобразование приводят к схеме трансляции (2.14). Продукции для expr (рис. 2.19) преобразуются в продукцию для expr и нового нетерминала rest (2.14). Продукции для term повторяют продукции для term из рис. 2.19. Обратите внимание, что лежащая в основе схемы грамматика отличается от описанной в примере 2.9, и эти различия делают возможной требуемую трансляцию.

$$\begin{aligned} \text{expr} &\rightarrow \text{term rest} \\ \text{rest} &\rightarrow + \text{term} \{ \text{print}('+') \} \text{ rest} | - \text{term} \{ \text{print}('-) \} \text{ rest} | \epsilon \\ \text{term} &\rightarrow 0 \{ \text{print}'(0)' \} \\ \text{term} &\rightarrow 1 \{ \text{print}'(1)' \} \\ &\dots \\ \text{term} &\rightarrow 9 \{ \text{print}'(9)' \} \end{aligned} \tag{2.14}$$

На рис. 2.21 показано, каким образом описанная грамматика транслирует $9-5+2$.

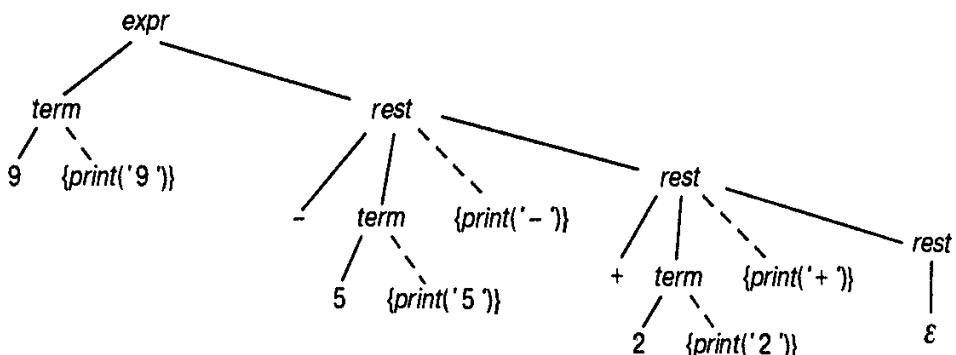


Рис. 2.21. Трансляция $9-5+2$ в $95-2+$

Процедуры для нетерминалов expr , term и rest

Теперь приступим к реализации транслятора на С с использованием синтаксически управляемой схемы трансляции (2.14). Квинтэссенция транслятора — код функций expr , term и rest на языке С (рис. 2.22). Эти функции реализуют соответствующие нетерминалы в (2.14).

```

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

rest()
{
}
  
```

```

if (lookahead == '+') {
    match('+'); term();
    putchar('+'); rest();
}
else if (lookahead == '-') {
    match('-'); term();
    putchar('-'); rest();
}
else ;
}
term()
{
    if (isdigit(lookahead)) {
        putchar(lookahead);
        match(lookahead);
    }
    else error();
}

```

Рис. 2.22. Функции для нетерминалов expr, rest и term

Функция `match`, которая будет представлена позже, является С-эквивалентом части кода на рис. 2.17. Данная часть кода отвечает за проверку соответствия токена текущему сканируемому символу и перемещение по входному потоку. Поскольку в данном случае каждый токен представляет собой отдельный символ, функция `match` может быть реализована простым сравнением и считыванием символов.

Для тех, кто незнаком с языком программирования С, вкратце опишем основные различия между С и другими производными от Algol языками программирования, такими как Pascal. Программа на языке С состоит из последовательности определений функций; работа программы начинается с выполнения специальной функции `main`. Определения функций не могут быть вложенными. Аргументы функций указываются в круглых скобках (они необходимы даже в том случае, когда никакие аргументы функции не передаются, именно поэтому мы и пишем `expr()`, `term()` и `rest()`). Функции сообщаются друг с другом либо путем передачи параметров “по значению”, либо с помощью глобальных данных, доступных всем функциям. Например, функции `term()` и `rest()` работают с текущим сканируемым символом с помощью глобальной переменной `lookahead`.

С и Pascal используют разные символы для присвоения и проверки равенства.

Операция	C	Pascal
Присвоение	=	:=
Проверка равенства	==	=
Проверка неравенства	!=	<>

Функции для нетерминалов имитируют правые части продукции. Например, продукция $expr \rightarrow term\ rest$ реализуется путем вызовов `term()` и `rest()` в функции `expr()`.

В качестве другого примера рассмотрим функцию `rest()`, которая использует первую продукцию для `rest` из (2.14), если текущий сканируемый символ представляет собой знак “+”, вторую продукцию — в случае знака “-” и продукцию $rest \rightarrow \epsilon$ по умолчанию. Первая продукция реализуется первой инструкцией `if` на рис. 2.22. Если сканируемый

символ — “плюс”, его соответствие токену и перемещение по входному потоку выполняется вызовом `match('+'')`. После вызова `term()` стандартная библиотечная функция С `putchar('+'')` выполняет семантическое действие, путем вывода символа “плюс”. Поскольку последняя продукция для `rest` представляет собой ϵ , последняя инструкция `else` в функции `rest()` не выполняет никаких действий.

Десять продуктов `term` порождают десять цифр. На рис. 2.22 подпрограмма `is-digit` проверяет, является ли сканируемый символ цифрой. Если да, функция `term()` выводит этот символ и вызывает функцию `match()` для перемещения по входному потоку к следующему символу. Если же считанный символ — не цифра, значит, произошла ошибка. (Обратите внимание: поскольку вызов `match()` изменяет текущий сканируемый символ, печать цифры должна производиться *до* этого вызова.) Перед тем как создать полную программу, осуществим некоторые улучшения кода (рис. 2.22) в целях повышения его быстродействия.

Оптимизация транслятора

Некоторые рекурсивные вызовы можно заменить итерациями. Когда рекурсивный вызов текущей процедуры производится ее последней инструкцией, говорим об *оконечной рекурсии* (tail recursion). Например, вызовы `rest()` в пятой и девятой строках ее кода представляют собой оконечную рекурсию, поскольку после каждого такого вызова происходит выход из функции. Можно ускорить работу программы, заменив оконечную рекурсию итерацией. В случае вызова процедуры без передачи ей параметров оконечная рекурсия может быть просто заменена безусловным переходом к началу процедуры, и код при этом будет выглядеть следующим образом.

```
rest()
{
L:  if (lookahead == '+') {
        match('+'); term();
        putchar('+'); goto L;
    }
    else if (lookahead == '-') {
        match('-'); term();
        putchar('-'); goto L;
    }
    else ;
}
```

До тех пор, пока сканируемый символ представляет собой знак “плюс” или “минус”, функция `rest()` вызывает функцию `match()`, передавая ей в качестве параметра символ, затем функцию `term()` для обработки цифры и повторяет процесс. Заметим, поскольку функция `match()` переходит к следующему символу входного потока, описанный цикл будет выполняться только при чередующейся последовательности цифр и знаков. Если внести описанные изменения в исходный текст (рис. 2.22), то увидим, что единственный вызов функции `rest()` имеется в функции `expr()`, в третьей строке кода. Таким образом, две функции можно объединить в одну, как показано на рис. 2.23. Кроме того, в С циклическое выполнение инструкции `stmt` обеспечивается записью

```
while(1) stmt
```

поскольку условие `l` всегда истинно. Выйти из цикла можно с помощью оператора `break`⁴. В результате, код функции `rest()` приобретает вид, приведенный на рис. 2.23.

```
expr()
{
    term();
    while(1)
        if (lookahead == '+') {
            match('+'); term(); putchar('+');
        }
        else if (lookahead == '-') {
            match('-'); term(); putchar('-');
        }
        else break;
}
```

Рис. 2.23. Вид функции `expr()` после оптимизации

Полная программа

Полностью код транслятора на языке С приведен на рис. 2.24. Первая строка, начинаящаяся с `#include`, загружает `<ctype.h>` — файл, содержащий, помимо прочих стандартных подпрограмм С, код предиката `isdigit`.

Токены, состоящие из отдельных символов, поставляются стандартной библиотечной функцией `getchar()`. Переменная `lookahead` описана как имеющая тип `int`, чтобы в дальнейшем, по мере развития программы, она могла хранить и другие токены. Поскольку переменная описана вне тела любой из функций, она является глобальной и может использоваться любой функцией, определенной после второй строки программы.

Функция `match()` проверяет соответствие текущего сканируемого символа токену и в случае соответствия считывает из входного потока следующий символ, а в ином случае вызывает функцию `error()`. Последняя использует стандартную библиотечную функцию `printf` для вывода сообщения о синтаксической ошибке и прекращает работу программы вызовом другой стандартной библиотечной функции С — `exit(1)`.

```
#include <ctype.h> /*Файл с предикатом isdigit */
int lookahead;

main()
{
    lookahead = getchar();
    expr();
    putchar('\n'); /* Символ перехода на новую строку */
}

expr()
{
    term();
    while(1)
        if (lookahead == '+') {
```

⁴ Все сказанное об инструкции `while(1)` справедливо и для инструкции `for(;;)`. — Прим. ред.

```

        match('+'); term(); putchar('+');
    }
    else if (lookahead == '-') {
        match('-'); term(); putchar('-');
    }
    else break;
}

term()
{
    if (isdigit(lookahead)) {
        putchar(lookahead);
        match(lookahead);
    }
    else error();
}

match(t)
int t;
{
    if (lookahead == t)
        lookahead = getchar();
    else error();
}

error()
{
    printf("Syntax error\n"); /* Сообщение об ошибке */
    exit(1); /* Прекращение выполнения программы */
}

```

Рис. 2.24. Программа на языке С для трансляции инфиксных выражений в постфиксные

2.6. Лексический анализ

Теперь к транслятору из предыдущего раздела добавим лексический анализатор, который считывает входную строку и превращает ее в поток токенов для последующего разбора анализатором. Вспомним из определения грамматики в разделе 2.2, что предложения языка состоят из строк токенов. Последовательность входных символов, входящих в состав единичного токена, называется лексемой. Лексический анализатор “ограждает” синтаксический анализатор от непосредственной работы с лексемами, представляющими токены. Рассмотрение лексического анализатора начнем с перечисления некоторых выполняемых им функций.

Удаление пробелов и комментариев

Транслятор выражений, разработанный в предыдущем разделе, воспринимает каждый символ из входной строки как отдельный токен, а потому наличие дополнительных символов, таких как символ пробела, приведет к сообщению о синтаксической ошибке или прекращению трансляции. Однако многие языки программирования допускают наличие “пустых символов” (таких, как пробелы, символы табуляции, символы перевода

каретки) между токенами. Поскольку комментарии в программах также игнорируются трансляторами, их можно отнести к этому же списку и рассматривать как некоторый специальный пробел.

Если “пустые символы” будут удалены лексическим анализатором, синтаксический анализатор никогда не столкнется с ними. Альтернативный способ, состоящий в модификации грамматики для включения “пустых символов” в синтаксис, достаточно сложен для реализации.

Константы

Везде, где в выражении встречается отдельная цифра, на ее месте естественно использовать произвольную целую константу. Поскольку целая константа — не что иное, как последовательность цифр, ее участие в трансляции может быть разрешено либо добавлением продукции к грамматике выражений, либо созданием токена для таких констант. Работа по сборке последовательных цифр в целые числа, как правило, поручается лексическому анализатору, поскольку целые числа рассматриваются как единицы трансляции.

Пусть `num` — токен, представляющий целое число. Когда во входном потоке встречается последовательность цифр, лексический анализатор передает синтаксическому анализатору токен `num`. Значение целого числа, представленного этим токеном, может передаваться как атрибут токена `num`. Логически лексический анализатор передает синтаксическому анализатору как токен, так и его атрибут. Если записать токен и его атрибут как кортеж, заключенный в угловые скобки `<>`, то входная строка

`31 + 28 + 59`

трансформируется в последовательность кортежей

`<num, 31> <+, > <num, 28> <+, > <num, 59>`

Токен + атрибутов не имеет. Вторые компоненты кортежей, представляющие атрибуты токенов, не играют роли в процессе разбора, но требуются на этапе трансляции.

Распознавание идентификаторов и ключевых слов

Языки используют идентификаторы в качестве имен переменных, массивов, функций и т.п. Грамматика языка часто рассматривает идентификатор как токен. Синтаксический анализатор, основанный на такой грамматике, должен получать один и тот же токен (скажем, `id`) всякий раз, когда во входном потоке встречается идентификатор. Так, вход

`count = count + increment;` (2.15)

должен конвертироваться лексическим анализатором в поток токенов

`id = id + id ;` (2.16)

который будет разбираться синтаксическим анализатором.

Говоря о лексическом анализе строки типа (2.15), следует четко понимать различие между токеном `id` и лексемами `count` и `increment`, связанными с экземплярами этого токена. Транслятор должен знать, что первые два токена `id` (2.16) образованы лексемой `count`, а последний токен `id` — лексемой `increment`.

Когда лексема, образующая идентификатор, поступает на вход, требуется некоторый механизм определения того, не появлялась ли эта лексема ранее. Как упоминалось в главе 1, “Введение в компиляцию”, в качестве такого механизма может использоваться таб-

лица символов. Лексемы хранятся в таблице символов, а указатель на соответствующую запись таблицы становится атрибутом токена *id*.

Многие языки используют определенные заранее строки символов, такие как *begin*, *end*, *if* и т.п., в качестве символов пунктуации или для указания конкретных конструкций. Такие строки символов, именуемые *ключевыми словами* (keyword), обычно удовлетворяют правилам образования идентификаторов, а потому требуется дополнительный механизм, позволяющий отличать ключевые слова от прочих идентификаторов. Проблема упрощается, если ключевые слова *зарезервированы*, т.е. они не могут использоваться как идентификаторы. В таком случае строка символов формирует идентификатор, только если она не является ключевым словом.

Проблема разделения токенов возникает и в том случае, когда одни и те же символы встречаются в лексемах нескольких токенов, как, например, <, <= и <> в Pascal. Технологии эффективного распознавания таких токенов обсуждаются в главе 3, “Лексический анализ”.

Интерфейс к лексическому анализатору

Если лексический анализатор находится между синтаксическим анализатором и входящим потоком, он взаимодействует с ними, как показано на рис. 2.25. Анализатор считывает символы из входного потока, группирует их в лексемы и передает токены, обозначаемые этими лексемами, вместе с их атрибутами последующим стадиям компиляции. В некоторых ситуациях лексический анализатор считывает несколько следующих символов, прежде чем принять решение, какой токен должен быть передан синтаксическому анализатору. Например, лексический анализатор Pascal должен прочитать символ, следующий после >. Если следующий символ — =, то формирующая токен лексема ($>=$) означает оператор “больше или равно”. В противном случае лексема > соответствует оператору “больше”. При этом лексический анализатор прочитал один лишний символ из входного потока. Этот символ следует вернуть во входной поток, поскольку он может оказаться началом следующей лексемы.

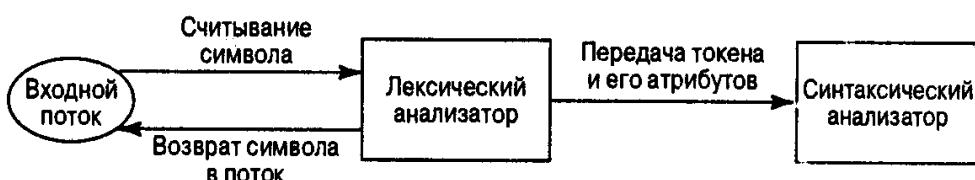


Рис. 2.25. Лексический анализатор между входным потоком и синтаксическим анализатором

Лексический и синтаксический анализаторы образуют пару *производитель–потребитель* (producer–customer). Лексический анализатор производит токены, потребляемые синтаксическим анализатором. До использования токены могут храниться в буфере. Взаимодействие между двумя анализаторами ограничивается только размером буфера: лексический анализатор не может продолжать работу, когда буфер заполнен, а синтаксический анализатор — когда буфер пуст. Обычно в буфере хранится только один токен. В этом случае взаимодействие реализуется с помощью разработки лексического анализатора в виде процедуры, вызываемой синтаксическим анализатором для получения очередного токена.

Считывание символа из входного потока и возврат во входной поток реализуются соответствующей настройкой входного буфера. Одновременно в буфер считывается блок символов; проанализированная часть входного потока отслеживается с помощью указа-

теля. При этом возврат символа во входной поток обеспечивается простым перемещением указателя на одну позицию назад. Входные символы могут понадобиться и для сообщения об ошибке, поскольку обычно указывается, где именно во входном тексте обнаружена ошибка. Кроме того, буферизация входного потока обеспечивает более высокую эффективность работы, поскольку считывание блока символов эффективнее чтения по одному символу. Технология буферизации входного потока обсуждается в разделе 3.2.

Лексический анализатор

Теперь приступим к созданию простейшего лексического анализатора для транслятора выражений из раздела 2.5. Цель лексического анализатора состоит в том, чтобы разрешить в выражениях использование пробелов и чисел. В следующем разделе мы продолжим работу над данным анализатором с тем, чтобы допустить в выражениях применение идентификаторов.

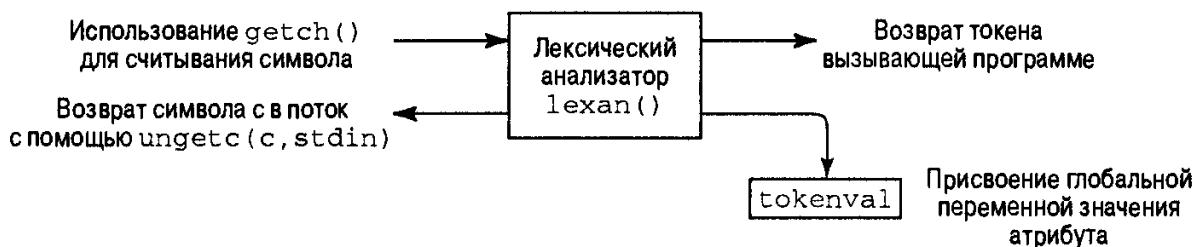


Рис. 2.26. Реализация взаимодействия, показанного на рис. 2.25

На рис. 2.26 показано, каким образом лексический анализатор, используемый в виде функции `lexan()` на языке С, реализует показанные на рис. 2.25 взаимодействия с входным потоком и синтаксическим анализатором. Работу с входным буфером обеспечивают функции `getchar` и `ungetc` из стандартного включаемого файла `<stdio.h>`. Функция `lexan` считывает символы и возвращает их во входной поток посредством вызовов функций `getchar` и `ungetc`. Если переменная с объявлена как символ (`char`), то пара инструкций

```
c = getchar(); ungetc(c, stdin);
```

оставляет входной поток в исходном состоянии. Вызов `getchar` присваивает следующий входной символ переменной `c`, а вызов `ungetc` возвращает `c` в стандартный входной поток `stdin`.

Если язык реализации лексического анализатора не позволяет возврат из функции структур данных, то токены и их атрибуты должны передаваться раздельно. Функция `lexan` возвращает целое число, представляющее код токена. Токен символа может быть обычным целым числом, представляющим этот символ. Другие токены, такие как `num`, должны иметь значения, большие целого значения любого символа, например 256. Чтобы упростить изменение кодов токенов, для указания токена `num` воспользуемся символьной константой `NUM`. В Pascal связь `NUM` и кода осуществляется с помощью объявления `const`; в С — с использованием инструкции

```
#define NUM 256
```

Функция `lexan` возвращает `NUM` в том случае, если во входном потоке встречается последовательность цифр. Глобальная переменная `tokenval` устанавливается равной числовому значению последовательности цифр. Таким образом, если во входном потоке непосредственно за символом 7 следует символ 6, `tokenval` получит значение 76.

Для использования чисел в выражениях нужно внести соответствующие изменения в грамматику (рис. 2.19). Необходимо заменить отдельные цифры нетерминалом *factor* и добавить следующие продукции и семантические действия:

```
factor → ( expr )
         | num { print(num.value) }
```

Соответствующий код на языке С (рис. 2.27) представляет собой непосредственную реализацию приведенных продуктов. Когда lookahead равно NUM, значение атрибута num.valueдается глобальной переменной tokenval. Выполнение семантического действия осуществляется стандартной библиотечной функцией printf. Первый аргумент printf — строка в двойных кавычках, определяющая используемый при выводе остальных аргументов формат. Так, %d в строке обеспечивает десятичный вывод следующего аргумента, и в результате приведенный код выводит пробел, за которым следует десятичное значение числа и еще один пробел.

```
factor()
{
    if (lookahead == '(') {
        match('('); expr(); match(')');
    }
    else if (lookahead == NUM) {
        printf(" %d ",tokenval); match(NUM);
    }
    else error();
}
```

Рис. 2.27. Код на языке С для нетерминала factor

Реализация функции lexan приведена на рис. 2.28. При каждом выполнении тела инструкции while (строки 8–28) в строке 9 происходит считывание нового символа в переменную t. Если этот символ — пробел или символ табуляции (обозначаемый в С как '\t'), производится повторное выполнение цикла. Если считывается символ перехода на новую строку, происходит увеличение глобальной переменной lineno на единицу для отслеживания номера текущей строки входного потока (это поможет отследить место возникновения ошибки), но токен не возвращается.

```
(1) #include <stdio.h>
(2) #include <ctype.h>
(3) int lineno = 1;
(4) int tokenval = NONE;
(5) int lexan()
(6) {
(7)     int t;
(8)     while(1) {
(9)         t = getchar();
(10)        if (t == ' ' || t == '\t')
(11)            ; /* Удаление пробелов и табуляции */
(12)        else if (t == '\n')
(13)            lineno++;
(14)        else if (isdigit(t)) {
(15)            tokenval = t - '0';
(16)            t = getchar();
```

```

(17)         while(isdigit(t)) {
(18)             tokenval = tokenval * 10 + t - '0';
(19)             t = getchar();
(20)         }
(21)         ungetc(t, stdin);
(22)         return NUM;
(23)     }
(24)     else {
(25)         tokenval = NONE;
(26)         return t;
(27)     }
(28) }
(29)

```

Рис. 2.28. Код на языке С лексического анализатора, удаляющего пробелы и собирающего числа в токены

Код,читывающий последовательность символов, приведен в строках 14–23. В строках 14 и 17 для определения, является ли считанный символ *t* цифрой, используется предикат *isdigit(t)* из стандартного включаемого файла <ctype.h>. Если этот символ — цифра, ее значение определяется выражением *t-'0'*, что справедливо как при использовании ASCII, так и EBCDIC. При применении других наборов символов, возможно, потребуется более сложный способ вычисления значения. В разделе 2.9 мы интегрируем построенный лексический анализатор в транслятор выражений.

2.7. Использование таблиц символов

Структура данных, именуемая таблицей символов, как правило, применяется для хранения информации о различных конструкциях исходного языка. Информация накапливается во время фазы анализа, а используется фазой синтеза при генерации целевого кода. Например, во время лексического анализа строка символов, или лексема, формирует идентификатор, хранящийся в виде записи в таблице символов. При дальнейшей компиляции к этим записям может добавляться дополнительная информация, например, о типе идентификаторов, их использовании (в качестве процедуры, переменной или метки) и размещении в памяти. Затем эта информация используется для генерации корректного кода хранения и доступа к этим переменным. В разделе 7.6 будет детально рассмотрена реализация и использование таблиц символов. В этом разделе мы проиллюстрируем, как лексический анализатор из предыдущего раздела может взаимодействовать с таблицей символов.

Интерфейс таблицы символов

Процедуры для работы с таблицей символов ориентированы, в основном, на хранение и получение лексем. После того как лексема сохранена, сохраняем токен, связанный с этой лексемой. Другими словами, над таблицей символов должны выполняться следующие операции.

insert(s, t) : Возвращает индекс новой записи для строки *s* и токена *t*.

lookup(s) : Возвращает индекс записи для строки *s* или 0, если строка не найдена.

Лексический анализатор использует операцию *lookup* (просмотр) для определения, имеется ли в таблице символов запись, соответствующая данной лексеме. Если такой записи нет,

для ее создания применяется операция `insert` (вставка). Рассмотрим реализацию, при которой и лексический, и синтаксический анализаторы имеют доступ к записям таблицы символов (и, соответственно, должны уметь работать с используемым форматом таблицы).

Обработка зарезервированных ключевых слов

Приведенные выше программы могут работать с любым набором зарезервированных ключевых слов. Например, рассмотрим токены `div` и `mod` с лексемами `div` и `mod` соответственно. Таблицу символов можно инициализировать вызовами

```
insert("div", div);
insert("mod", mod);
```

Любой последующий вызов `lookup("div")` вернет токен `div`, а потому `div` не может быть использован в качестве идентификатора.

Таким образом, с помощью соответствующей инициализации таблицы символов можно работать с любым набором зарезервированных ключевых слов.

Реализация таблицы символов

Образец структуры данных для реализации таблицы символов приведен на рис. 2.29. Для хранения лексем, образующих идентификаторы, не стоит отводить память фиксированного размера — этой памяти может оказаться недостаточно для очень длинного идентификатора и слишком много для коротких идентификаторов типа `i`, что влечет за собой нерациональное использование памяти. На рис. 2.29 строки символов, образующие лексемы, хранятся в специальном массиве `lexemes`. Строки завершаются символом конца строки (`end-of-string`), обозначенным как `EOS`, который не может встречаться в идентификаторе. Каждая запись в массиве таблицы символов `symtable` представляет собой запись, состоящую из двух полей — `lexptr`, указывающего на начало лексемы, и `token`. Для хранения атрибутов могут использоваться и дополнительные поля.

На рис. 2.29 нулевая запись пуста, поскольку для указания, что для данной строки в таблице символов не имеется записи, функция `lookup` возвращает 0. Первая и вторая записи используются для зарезервированных ключевых слов `div` и `mod`. В третьей и четвертой записях хранятся указатели на идентификаторы `count` и `i`.

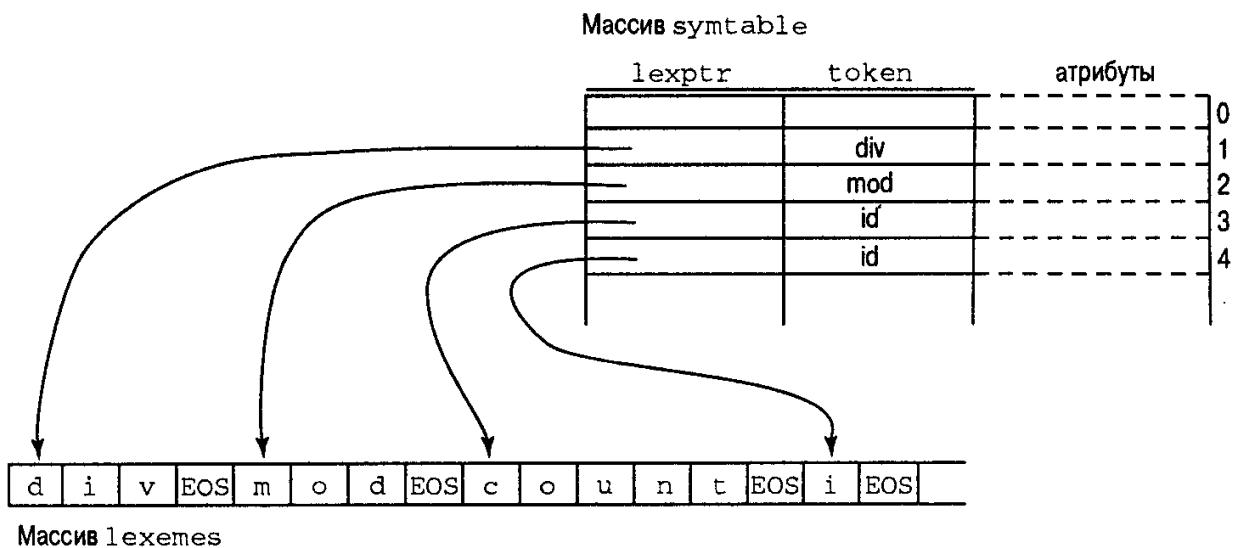


Рис. 2.29. Таблица символов и массив для хранения строк

Псевдокод лексического анализатора приведен на рис. 2.30, а его реализация на языке С представлена в разделе 2.9. Пробелы и целые константы обрабатываются этим лексическим анализатором так же, как и на рис. 2.28.

```
function lexan: integer;
var lexbuf: array [0..100] of char;
    c:      char;
begin
    loop begin
        Прочесть символ в с
        if с - пробел или табулятор then
            ничего не делать
        else if с - символ новой строки then
            lineno := lineno + 1
        else if с - цифра then begin
            Установить tokenval равным значению
            этой и последующих цифр
            return NUM
        end
        else if с - буква then begin
            Поместить с и последующие символы и цифры в lexbuf
            p := lookup(lexbuf);
            if p = 0 then
                p := insert(lexbuf, ID);
            tokenval := p;
            return Поле token записи p таблицы
        end
        else begin /* Токен - отдельный символ */
            Установить tokenval равным NONE /*Нет атрибутов */
            return Целое, представляющее код символа с
        end
    end
end
```

Рис. 2.30. Псевдокод лексического анализатора

При считывании буквы лексический анализатор начинает записывать буквы и цифры в буфер lexbuf. Затем сохраненная в буфере строка ищется в таблице символов с помощью операции lookup. Поскольку таблица символов изначально инициализирована записями для ключевых слов div и mod, как показано на рис. 2.29, их можно найти, если lexbuf содержит обе эти лексемы. Если в таблице символов нет записи, соответствующей строке в lexbuf (т.е. функция lookup вернет значение 0), значит, lexbuf содержит лексему нового идентификатора. Запись для нового идентификатора создается с помощью функции insert. После вставки новой записи в таблицу символов переменная p содержит индекс записи в таблице для строки в lexbuf. Этот индекс передается синтаксическому анализатору путем присвоения переменной tokenval; функция при этом возвращает токен, хранящийся в поле token записи в таблице символов.

Действие по умолчанию состоит в возврате целого значения, являющегося кодом считанного символа, в качестве очередного токена. Поскольку токены, представляющие отдельные символы, не имеют атрибутов, переменной `tokenval` присваивается значение `NONE`.

2.8. Абстрактная стековая машина

Начальная стадия компилятора строит промежуточное представление исходной программы, из которого на заключительной стадии генерируется целевая программа. Один из популярных видов промежуточного представления — код для абстрактной стековой машины. Как упоминалось в разделе 1.5, разделение процесса компиляции на начальную и заключительную стадии упрощает адаптацию компилятора для работы на новой машине.

В этом разделе рассматривается абстрактная стековая машина и показано, как можно сгенерировать код для нее. Машина имеет раздельную память для инструкций и данных. Все арифметические выражения вычисляются над значениями в стеке. Инструкции такой машины существенно ограничены и разделяются на три класса: целой арифметики, работы со стеком и управления исполнением. Данная машина представлена на рис. 2.31. Указатель `pc` определяет инструкцию, которая должна выполняться в настоящий момент. Далее будет вкратце пояснен смысл показанных инструкций.

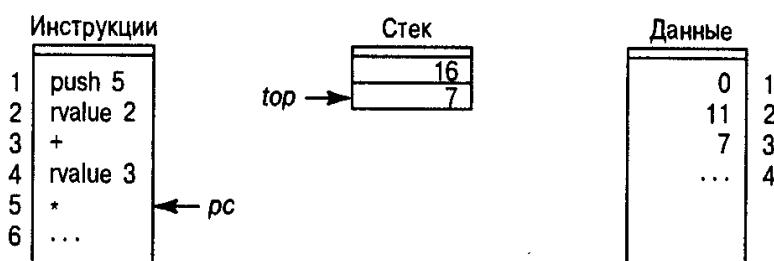


Рис. 2.31. Снимок стековой машины после выполнения первых четырех инструкций

Арифметические инструкции

Абстрактная машина должна выполнять все операции промежуточного языка. Базовые операции, такие как сложение или вычитание, поддерживаются абстрактной машиной непосредственно. Более сложные операции могут быть реализованы как последовательность инструкций абстрактной машины. Можно упростить описание машины, полагая, что для каждого арифметического оператора есть своя инструкция.

Код абстрактной машины для арифметических выражений имитирует вычисление постфиксного представления выражения с использованием стека. Вычисления производятся путем обработки постфиксного представления слева направо, при этом каждый встречающийся операнд помещается в стек. Если встречается k -арный оператор, его крайний слева аргумент находится на $k-1$ позицию ниже вершины стека, а его крайний справа аргумент — на вершине стека. При вычислении используются k верхних значений, которые при этом удаляются из стека, и результат помещается в стек. Например, при вычислении постфиксного выражения $1\ 3\ +\ 5\ *$ выполняются следующие действия.

1. В стек помещается 1.

2. В стек помещается 3.
3. Суммируются два верхних элемента стека (при этом они удаляются из стека), и в стек заносится результат сложения — 4.
4. В стек помещается 5.
5. Умножаются два верхних элемента стека (при этом они удаляются из стека), и в стек заносится результат умножения — 20.

По окончании вычислений значение на вершине стека (в данном случае равное 20) представляет собой результат вычисления всего выражения.

В данном промежуточном языке все значения являются целыми; `false` соответствует нулевое значение, а `true` — ненулевое целое. Логические операторы `and` и `or` требуют вычисления обоих своих аргументов.

I-значения и r-значения

Идентификаторы слева и справа в операторе присваивания имеют два разных смысла. В каждом из присваиваний

```
i := 5;
i := i + 1;
```

правая сторона определяет целое значение, в то время как левая — адрес памяти, где это значение будет храниться. Точно так же и в случае с инструкцией (если `p` и `q` — указатели на символы)

```
p↑ := q↑;
```

ее правая часть q^\uparrow указывает символ, в то время как левая часть p^\uparrow — место в памяти, куда следует поместить этот символ. Термины *l-значение* и *r-значение* (*l-value* и *r-value*) определяют значения, которые могут использоваться в левой и правой частях инструкции присвоения соответственно. Таким образом, *r-значения* — это то, что обычно подразумевается под словом “значение”, в то время как *l-значения* — это адреса памяти.

Работа со стеком

Кроме очевидных инструкций для помещения целой константы в стек и удаления значения с вершины стека, имеются следующие инструкции для обращения к данным.

<code>push v</code>	Поместить <code>v</code> в стек.
<code>rvalue l</code>	Поместить в стек содержимое памяти по адресу <code>l</code> .
<code>lvalue l</code>	Поместить в стек адрес ячейки памяти <code>l</code> .
<code>pop</code>	Удалить значение с вершины стека.
<code>:=</code>	<i>r</i> -значение на вершине стека размещается по адресу, представленному <i>l</i> -значением, следующим за ним в стеке; обе величины после этого удаляются из стека.
<code>copy</code>	Поместить копию значения на вершине стека в стек.

Трансляция выражений

Код для вычисления выражения в стековой машине тесно связан с постфиксной записью этого выражения. Постфиксная форма выражения $E+F$ по определению представляет собой конкатенацию постфиксной записи E , постфиксной записи F и $+$. Точно так же

код стековой машины для вычисления $E+F$ является конкатенацией кода для вычисления E , кода для вычисления F и инструкции, суммирующей полученные значения. Следовательно, трансляцию выражений в код стековой машины можно выполнить путем адаптации транслятора из разделов 2.6 и 2.7.

Сгенерируем теперь стековый код для выражений, в которых ячейки данных адресуются символически (выделение памяти для идентификаторов обсуждается в главе 7, “Среды времени исполнения”). Выражение $a+b$ транслируется в

```
rvalue a  
rvalue b  
+
```

Другими словами, помещаем содержимое ячеек памяти a и b в стек, затем снимаем эти значения из стека, суммируем, и результат заносим в стек.

Трансляция присвоений в стековой машине выполняется следующим образом. l -значение идентификатора, которому будет присвоено значение, помещается в стек, затем вычисляется выражение, и его r -значение присваивается идентификатору. Например, присвоение

$day := (1461*y) \text{ div } 4 + (153*m + 2) \text{ div } 5 + d$ (2.17)

транслируется в код, приведенный на рис. 2.32.

```
lvalue day          push    2  
push   1461          +  
rvalue y            push    5  
*                  div  
push   4             +  
div                rvalue d  
push   153           +  
rvalue m           :=  
*
```

Рис. 2.32. Трансляция $day := (1461*y) \text{ div } 4 + (153*m + 2) \text{ div } 5 + d$

Все эти замечания можно формализовать следующим образом. Каждый нетерминал имеет атрибут t , задающий его трансляцию. Атрибут $lexeme$ токена id дает строку, представляющую идентификатор

$$stmt \rightarrow id := expr \{ stmt.t := 'lvalue' \parallel id.lexeme \parallel expr.t \parallel ':=' \}$$

Управление выполнением

Стековая машина выполняет инструкции последовательно, если иное не предписано инструкциями условного или безусловного перехода. Для определения целевого адреса перехода имеется ряд опций.

1. Операнд инструкции дает целевой адрес.
2. Операнд инструкции определяет относительную дистанцию перехода (положительную или отрицательную).
3. Целевой адрес указывается символически (т.е. машина поддерживает метки).

При работе с первыми двумя опциями имеется дополнительная возможность получения операнда с вершины стека.

Для данной абстрактной машины выберем третью опцию, поскольку при этом упрощается генерация переходов. Более того, символьическая адресация не должна изменяться при оптимизации сгенерированного кода (когда некоторые инструкции могут добавляться в код или удаляться из него).

Инструкции управления выполнением в стековой машине следующие:

<code>label 1</code>	Указание цели переходов к 1; никаких других действий не выполняет.
<code>goto 1</code>	Следующая выполняемая инструкция расположена по адресу 1.
<code>gofalse 1</code>	Снять значение с вершины стека и осуществить переход, если оно нулевое.
<code>gotrue 1</code>	Снять значение с вершины стека и осуществить переход, если оно ненулевое.
<code>halt</code>	Прекратить выполнение.

Трансляция инструкций

На рис. 2.33 схематически показан код абстрактной машины для условного перехода и для инструкции `while`. Дальнейшее обсуждение вопроса посвящено созданию меток.

Рассмотрим код для инструкции `if` (рис. 2.33). При трансляции исходной программы может быть только одна инструкция `label out`, в противном случае возникнет конфликт при выполнении инструкции `goto out`. Следовательно, требуется механизм, обеспечивающий согласованную замену `out` уникальной меткой при трансляции инструкции `if`.

Предположим, что *newlabel* представляет собой процедуру, которая возвращает новую метку при каждом вызове. В следующем семантическом действии метка, возвращаемая вызовом *newlabel*, записывается с использованием локальной переменной *out*.

$$\begin{aligned}
 \text{stmt} \rightarrow & \text{ if } \text{expr} \text{ then } \text{stmt}_1 \quad \{ \text{out} := \text{newlabel}; \\
 & \text{stmt}.t := \text{expr}.t || \\
 & \quad \quad \quad \text{'gofalse'} \text{ out } || \\
 & \quad \quad \quad \text{stmt}_1.t || \\
 & \quad \quad \quad \text{'label'} \text{ out } \}
 \end{aligned} \tag{2.18}$$

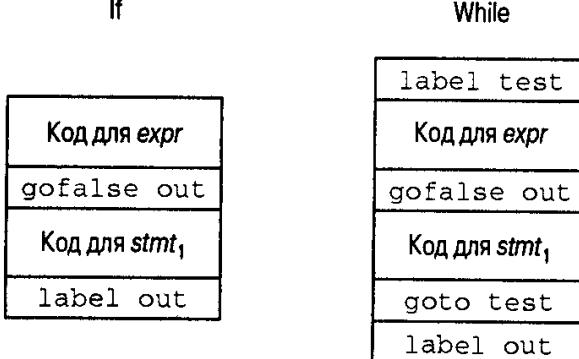


Рис. 2.33. Схема кода условного перехода и инструкции *while*

Вывод результатов трансляции

Трансляторы выражений из раздела 2.5 для инкрементальной генерации трансляции выражений использовали оператор печати. Схожие операторы могут применяться и при трансляции инструкций. Вместо операции печати используем процедуру

ру *emit* для скрытия деталей вывода. Например, используя процедуру *emit*, вместо (2.18) можем записать

```
stmt → if
    expr { out := newlabel; emit('gofalse', out); }
    then
        stmt1 { emit('label', out); }
```

Когда в продукции встречаются семантические действия, элементы ее правой части рассматриваются слева направо. В приведенной выше продукции вначале выполняются действия, соответствующие анализу *expr*; затем *out* присваивается значение, возвращаемое функцией *newlabel*; в код вносится инструкция *gofalse*; осуществляются действия, соответствующие анализу *stmt₁*; и, наконец, в код вносится инструкция *label*. Поскольку в процессе анализа *expr* и *stmt₁* создается необходимый код для этих нетерминалов, описанные продукции реализуют код, приведенный на рис. 2.33.

```
procedure stmt;
var test, out: integer; /* Для меток */
begin
    if lookahead = id then begin
        emit('lvalue', tokenval); match(id); match(':='); expr
    end
    else if lookahead = 'if' then begin
        match('if');
        expr;
        out := newlabel;
        emit('gofalse', out);
        match('then');
        stmt;
        emit('label', out);
    end
    /* Код для трансляции других инструкций */
    else error
end
```

Рис. 2.34. Псевдокод для трансляции инструкций

Псевдокод для трансляции присвоений и условных инструкций показан на рис. 2.34. Поскольку переменная *out* локальна для процедуры *stmt*, ее значение не влияет на вызовы процедур *expr* и *stmt*. Генерация меток требует определенных усилий. Предположим, что при трансляции метки имеют вид L1, L2 и т.д. Псевдокод, работающий с такими метками, использует целое число, следующее за L. В результате переменная *out* объявлена как целое число, функция *newlabel* возвращает целую величину, которая становится значением переменной *out*, и *emit* выводит метку, заданную целым числом.

Макет кода для инструкции *while* (рис. 2.33) можно перевести в псевдокод подобным образом. Трансляция последовательности инструкций представляет собой простую конкатенацию инструкций этой последовательности и предлагается читателю в качестве небольшого упражнения.

Трансляция большинства конструкций с одним входом (*single-entry*) и выходом (*single-exit*) схожа с трансляцией инструкции `while`. Проиллюстрируем это на примере управления потоком выполнения в выражениях.

Пример 2.10

Лексический анализатор в разделе 2.7 содержит условный оператор типа
`if t = blank or t = tab then ...`

Если *t* является пробелом, то совершенно очевидно, что не нужно проверять, является ли *t* символом табуляции, поскольку с учетом первого равенства все выражение в целом истинно. Выражение

`expr1 or expr2`

может быть реализовано как

`if expr1 then true else expr2`

Читатель может убедиться, что следующий код реализует оператор `or`:

Код для *expr₁*
`copy /* Копирование значения expr1 */
gotrue out
pop /* Снятие со стека значения expr1 */`
Код для *expr₂*
`label out`

Вспомним, инструкции `gotrue` и `gofalse` удаляют значение с вершины стека, чтобы упростить генерацию кода для инструкций `if` и `while`. Копируя значение *expr₁*, можно гарантировать, что значение на вершине стека остается равным `true` при переходе посредством инструкции `gotrue`. □

2.9. Сборка транслятора

В этой главе было рассмотрено множество синтаксически управляемых технологий для построения начальной стадии компилятора. В данном разделе мы объединим все рассмотренные технологии в программе на языке C, которая служит транслятором инфиксной записи в постфиксную для языка, состоящего из последовательности выражений, разделенных точками с запятой. Выражения состоят из чисел, идентификаторов и операторов `+`, `-`, `*`, `/`, `div` и `mod`. Выход транслятора представляет собой постфиксное представление каждого выражения. Транслятор является расширением программ, разработанных в разделах 2.5–2.7. Полный листинг программы приведен в конце раздела.

Описание транслятора

Транслятор создан с использованием схемы синтаксически управляемой трансляции, представленной на рис. 2.35. Токен `id` представляет непустую последовательность букв и цифр, начинающуюся с буквы, `num` — последовательность цифр, а `eof` — символ конца файла. Токены разделяются последовательностями пробелов, символов табуляций и новой строки. Атрибут *lexeme* токена `id` указывает строку, образующую токен. Атрибут *value* токена `num` содержит целое число, представляющее `num`.

Код транслятора разделен на семь модулей, каждый из которых хранится в отдельном файле. Выполнение начинается с модуля `main.c`, в котором последовательно вызываются функции `init()` — для инициализации, и `parse()` — для трансляции. Остальные шесть модулей показаны на рис. 2.36. Имеется также глобальный заголовочный файл `global.h`, содержащий определения, которые встречаются в нескольких модулях. Первая инструкция каждого модуля

```
#include "global.h"
```

приводит к включению этого файла в модуль в качестве составной части. Перед тем как привести код транслятора, вкратце опишем каждый из модулей.

<i>start</i>	\rightarrow	<i>list eof</i>	
<i>list</i>	\rightarrow	<i>expr ; list</i>	
		ϵ	
<i>expr</i>	\rightarrow	<i>expr + term</i>	{ <code>print('+')</code> }
		<i>expr + term</i>	{ <code>print(' - ')</code> }
		<i>term</i>	
<i>term</i>	\rightarrow	<i>term * factor</i>	{ <code>print('*')</code> }
		<i>term / factor</i>	{ <code>print('/')</code> }
		<i>term div factor</i>	{ <code>print('DIV')</code> }
		<i>term mod factor</i>	{ <code>print('MOD')</code> }
		<i>factor</i>	
<i>factor</i>	\rightarrow	(<i>expr</i>)	
		<i>id</i>	{ <code>print(id.lexeme)</code> }
		<i>num</i>	{ <code>print(num.value)</code> }

Рис. 2.35. Спецификация транслятора из инфиксной формы записи в постфиксную

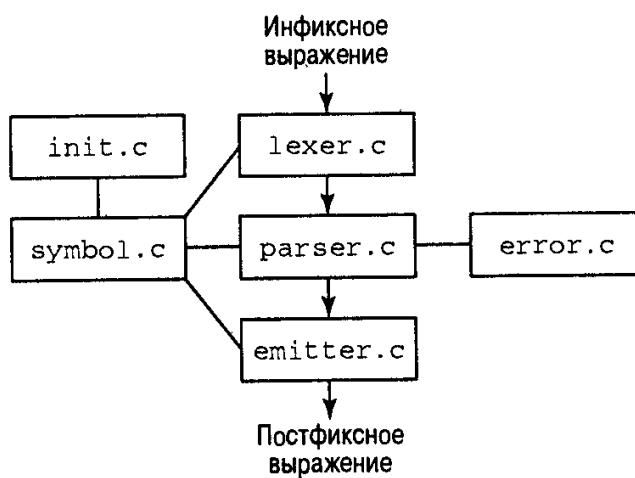


Рис. 2.36. Модули транслятора из инфиксной формы записи в постфиксную

Модуль лексического анализа `lexer.c`

Лексический анализатор представляет собой функцию `lexan()`, вызываемую синтаксическим анализатором для получения токена. Построенная на основе псевдокода из рис. 2.30, функция считывает по одному символу из входного потока и возвращает син-

таксическому анализатору найденный токен. Значение атрибута, связанного с токеном, присваивается глобальной переменной `tokentval`.

Синтаксический анализатор воспринимает следующие токены.

+ - * / DIV MOD () ID NUM DONE

Здесь ID соответствует идентификатору, NUM — числу, а DONE — концу файла. Пробелы удаляются лексическим анализатором автоматически. На рис. 2.37 показаны токены и атрибуты, производимые лексическим анализатором для различных лексем.

ЛЕКСЕМА	ТОКЕН	ЗНАЧЕНИЕ АТРИБУТА
Пробел		
Последовательность цифр	NUM	Числовое значение последовательности
div	DIV	
mod	MOD	
Прочие последовательности символов, начинающиеся с буквы и состоящие из букв и цифр	ID	Индекс в таблице символов
Символ конца файла	DONE	
Прочие символы	Этот символ	NONE

Рис. 2.37. Описание токенов

Лексический анализатор использует для работы с таблицей символов функцию `lookup`, которая определяет, имеется ли в таблице символов данная лексема, и `insert`, сохраняющую новую лексему в таблице символов. Кроме того, лексический анализатор увеличивает глобальную переменную `lineno` всякий раз, когда во входящем потоке встречается символ новой строки.

Модуль синтаксического анализатора `parser.c`

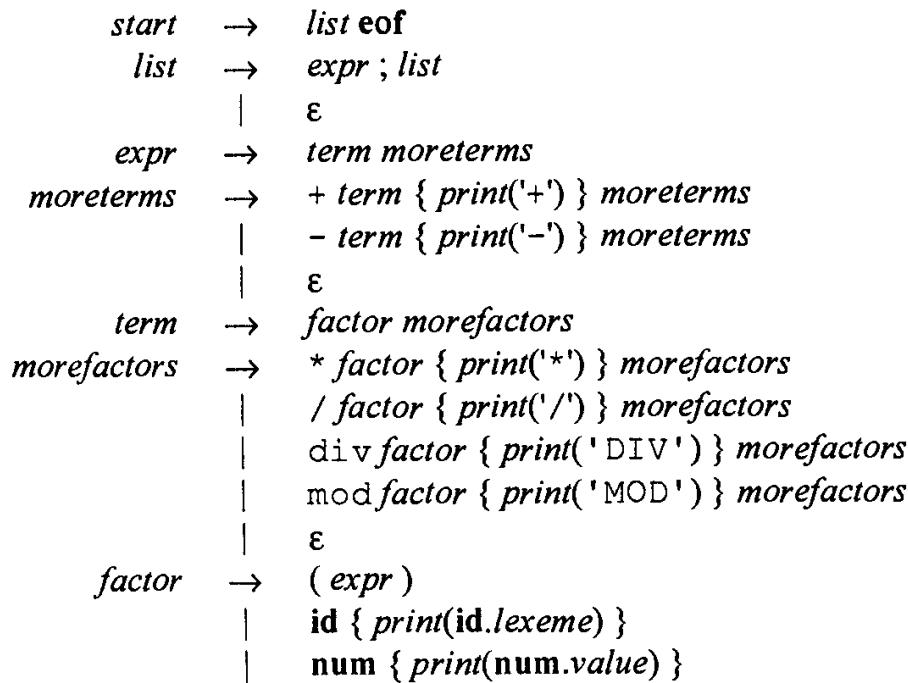


Рис. 2.38. Схема синтаксически управляемой трансляции после устранения левой рекурсии

Синтаксический анализатор строится с использованием технологий, описанных в разделе 2.5. Вначале из схемы трансляции (рис. 2.35) устраним левую рекурсию, чтобы иметь возможность проанализировать грамматику с помощью метода рекурсивного спуска. Преобразованная схема показана на рис. 2.38.

Затем создаем функции для нетерминалов *expr*, *term* и *factor*, как показано на рис. 2.24. Функция *parse()* реализует стартовый символ грамматики. Чтобы получить новый токен, эта функция вызывает функцию *lexan()*. Для генерации выхода синтаксический анализатор использует функцию *emit*, а для вывода сообщения об ошибке — функцию *error*.

Модуль вывода `emitter.c`

Модуль вывода состоит из единственной функции *emit(t, tval)*, которая генерирует выход для токена *t* со значением атрибута *tval*.

Модули работы с таблицей символов `symbol.c` и `init.c`

Модуль работы с таблицей символов `symbol.c` реализует структуру данных, показанную на рис. 2.29 в разделе 2.7. Записи в массиве `symtable` представляют собой пары, состоящие из указателя на массив `lexemes` и целой величины, описывающей хранимый токен. Операция *insert(s, t)* возвращает индекс в таблице `symtable` для лексемы *s*, образующей токен *t*. Функция *lookup(s)* возвращает индекс записи в таблице `symtable` для лексемы *s* (или 0, если такой записи нет).

Модуль `init.c` используется для первоначального занесения в таблицу символов зарезервированных ключевых слов. Лексемы и токены, представляющие зарезервированные ключевые слова, хранятся в массиве `keywords`, который имеет тот же тип, что и массив `symtable`. Функция *init()* проходит последовательно по всем элементам массива `keywords` и использует функцию *insert* для помещения ключевых слов в таблицу символов. Такая технология позволяет легко и просто изменить набор ключевых слов и представление их токенов.

Модуль обработки ошибок `error.c`

Этот модуль используется при выводе сообщения об ошибке. При возникновении ошибки транслятор просто выводит сообщение о ней и прекращает работу. Однако было бы разумнее применить метод, при котором возникшая ошибка приводила бы к пропуску оставшейся части строки до символа ";" и продолжению анализа. Читатель может самостоятельно внести в исходные тексты необходимые изменения. Ряд интеллектуальных технологий восстановления после ошибок представлен в главе 4, "Синтаксический анализ".

Создание компилятора

Код модулей расположен в семи файлах: `lexer.c`, `parser.c`, `emitter.c`, `symbol.c`, `init.c`, `error.c` и `main.c`. Файл `main.c` содержит функцию `main()`, которая вызывает функции `init()`, `parser()`, а при успешном завершении программы — `exit(0)`.

В операционной системе UNIX компилятор может быть создан с помощью команды
cc lexer.c parser.c emitter.c symbol.c init.c error.c main.c
либо отдельной компиляцией файлов
cc -c filename.c

и компоновкой полученных файлов filename.o

cc lexer.o parser.o emitter.o symbol.o init.o error.o main.o

Команда cc создает файл a.out, содержащий транслятор. Транслятор может быть запущен вводом в командной строке a.out⁵, а затем — транслируемых выражений

2 + 3 * 5;
12 div 5 mod 2;

или любых других.

Листинг

Далее приводится листинг программы на языке C, реализующий транслятор. Вначале приведен общий заголовочный файл global.h, за которым следует семь исходных файлов.

```
***** global.h *****  
  
#include <stdio.h> /* Загрузка программ ввода-вывода */  
#include <ctype.h> /* Загрузка программ проверки символов */  
  
#define BSIZE 128 /* Размер буфера */  
#define NONE -1  
#define EOS '\0'  
  
#define NUM 256  
#define DIV 257  
#define MOD 258  
#define ID 259  
#define DONE 260  
  
extern int tokenval; /* Значение атрибута токена */  
extern int lineno;  
  
struct entry { /* Запись в таблице символов */  
    char *lexptr;  
    int token;  
};  
  
extern struct entry symtable[]; /* Таблица символов */  
  
***** lexer.c *****  
  
#include "global.h"
```

⁵ Обычно для запуска используется ввод ./a.out — Прим. ред.

```

char lexbuf[BSIZE];
int lineno = 1;
int tokenval = NONE;

int lexan()           /* Лексический анализатор */ {
    int t;
    while (1) {
        t = getchar();
        if (t == ' ' || t == '\t')
            ;                                /* Отбрасываем разделители-пробелы */
        else if (t == '\n')
            lineno++;
        else if (isdigit(t)) {             /* t - цифра */
            ungetc(t, stdin);
            scanf("%d", &tokenval);
            return NUM;
        }
        else if (isalpha(t)) {           /* t - буква */
            int p, b = 0;
            while (isalnum(t)) {        /* t - буква или цифра */
                lexbuf[b++] = t;
                t = getchar();
                if (b >= BSIZE)
                    error("compiler error");
            }
            lexbuf[b] = EOS;
            if (t != EOF)
                ungetc(t, stdin);
            p = lookup(lexbuf);
            if (p == 0)
                p = insert(lexbuf, ID);
            tokenval = p;
            return symtable[p].token;
        }
        else if (t == EOF)
            return DONE;
        else {
            tokenval = NONE;
            return t;
        }
    }
}

***** parser.c *****

#include "global.h"

int lookahead;

parse()           /* Разбор и трансляция списка выражений */
{
    lookahead = lexan();

```

```

        while (lookahead != DONE) {
            expr(); match(';', );
        }
    }

expr()
{
    int t;
    term();
    while (1)
        switch (lookahead) {
            case '+': case '-':
                t = lookahead;
                match(lookahead); term(); emit(t, NONE);
                continue;
            default:
                return;
        }
}

term()
{
    int t;
    factor();
    while (1)
        switch (lookahead) {
            case '*': case '/': case DIV: case MOD:
                t = lookahead;
                match(lookahead); factor(); emit(t, NONE);
                continue;
            default:
                return;
        }
}

factor()
{
    switch (lookahead) {
        case '(':
            match('('); expr(); match(')'); break;
        case NUM:
            emit(NUM, tokenval); match(NUM); break;
        case ID:
            emit(ID, tokenval); match(ID); break;
        default:
            error("syntax error");
    }
}

match(int t)
{
    if (lookahead == t)
        lookahead = lexan();
}

```

```

        else error("syntax error");
    }

/****** emitter.c *****/
#include "global.h"

emit(int t, int tval) /*Генерация вывода */ {
    switch (t) {
        case '+': case '-': case '*': case '/':
            printf("%c\n", t); break;
        case DIV:
            printf("DIV\n"); break;
        case MOD:
            printf("MOD\n"); break;
        case NUM:
            printf("%d\n", tval); break;
        case ID:
            printf("%s\n", symtable[tval].lexptr); break;
        default:
            printf("token %d, tokenval %d\n", t, tval);
    }
}

```

```

/****** symbol.c *****/
#include "global.h"

#define STRMAX      999 /* Размер массива лексем */
#define SYMMAX      100 /* Размер таблицы символов */

char lexemes[STRMAX];
int lastchar = -1; /* Последняя использованная */
                   /* позиция в lexemes */
struct entry symtable[SYMMAX];
int lastentry = 0; /* Последняя использованная */
                   /* позиция в таблице символов */

int lookup(char s[]) /* Возвращает положение в таблице */
{                  /* символов для s */
    int p;
    for(p = lastentry; p > 0; p--)
        if (strcmp(symtable[p].lexptr, s) == 0)
            return p;
    return 0;
}

int insert(char s[], /* Возвращает положение в таблице */
           int tok) /* символов для s */
{
    int len;

```

```

len = strlen(s);      /* strlen вычисляет длину строки */
if (lastentry + 1 >= SYMMAX)
    error("symbol table full");
if (lastchar + len + 1 >= STRMAX)
    error("lexemes array full");
lastentry++;
symtable[lastentry].token = tok;
symtable[lastentry].lexptr = &lexemes[lastchar + 1];
lastchar += len + 1;
strcpy(symtable[lastentry].lexptr, s);
return lastentry;
}

```

```

*****      init.c ****
#include "global.h"

struct entry keywords[] = {
    "div", DIV,
    "mod", MOD,
    0, 0
};

init()           /* Загрузка ключевых слов в таблицу символов */
{
    struct entry *p;
    for(p = keywords; p->token; p++)
        insert(p->lexptr, p->token);
}

```

```

*****      main.c ****
#include "global.h"

main()
{
    init();
    parse();
    exit(0);          /* Успешное завершение работы программы */
}
*****
```

Упражнения

2.1. Рассмотрим контекстно-свободную грамматику

$$S \rightarrow SS^+ | SS^* | a$$

- a) покажите, что этой грамматикой может быть сгенерирована строка aa+a*;

- b) постройте дерево разбора для этой строки;
- c) какой язык генерируется этой грамматикой? Обоснуйте свой ответ.

2.2. Какой язык генерируется следующей грамматикой? (В каждом случае обоснуйте ответ.)

- a) $S \rightarrow 0 S 1 | 0 1$
- b) $S \rightarrow + S S | - S S | a$
- c) $S \rightarrow S(S)S | \epsilon$
- d) $S \rightarrow a S b S | b S a S | \epsilon$
- e) $S \rightarrow a | S + S | S S | S^* | (S)$

2.3. Какая из грамматик в упр. 2.2 неоднозначна?

2.4. Постройте однозначную контекстно-свободную грамматику для каждого из следующих языков (и покажите ее корректность).

- a) арифметические выражения в постфиксной записи;
- b) левоассоциативные списки идентификаторов, разделенных запятыми;
- c) правоассоциативные списки идентификаторов, разделенных запятыми;
- d) арифметические выражения с целыми и идентификаторами с четырьмя бинарными операциями $+, -, *, /$;
- e) добавьте к арифметическим операторам из п. d унарные плюс и минус.

***2.5.** Покажите, что все двоичные строки, порождаемые следующей грамматикой, имеют значение, кратное 3. (Указание. Используйте индукцию по количеству узлов в дереве разбора.)

$$num \rightarrow 11 | 1001 | num\ 0 | num\ num$$

Генерирует ли эта грамматика все двоичные строки со значениями, кратными 3?

2.6. Постройте контекстно-свободную грамматику для римских чисел.

2.7. Постройте схему синтаксически управляемой трансляции, которая преобразует арифметические выражения из инфиксной записи в префиксную (оператор записывается перед своими operandами). Например, $-x u$ представляет собой префиксную запись выражения $x - u$. Постройте аннотированные деревья разбора для входных строк $9 - 5 + 2$ и $9 - 5 * 2$.

2.8. Постройте схему синтаксически управляемой трансляции, которая переводит арифметические выражения из постфиксной в инфиксную запись. Постройте аннотированные деревья разбора для входных строк $95 - 2^*$ и $952^* -$.

2.9. Постройте схему синтаксически управляемой трансляции для перевода целых чисел в римские.

2.10. Постройте схему синтаксически управляемой трансляции для преобразования римских чисел в целые.

2.11. Постройте для грамматик a, b и c из упр. 2.2 синтаксические анализаторы, работающие по методу рекурсивного спуска.

- 2.12.** Постройте синтаксически управляемый транслятор, который проверяет сбалансированность скобок во входной строке.
- 2.13.** Следующие правила определяют перевод английских слов на “поросячью латынь”⁶:
- если слово начинается с непустой строки согласных, переместите начальную строку согласных в конец слова и добавьте суффикс AY (pig становится igray);
 - если слово начинается с гласной, добавьте суффикс YAY (owl становится owl yay);
 - U, следующее за Q, считается согласной;
 - Y в начале слова считается гласной, если за ней не следует гласная;
 - слово из одной буквы не изменяется.

Постройте схему синтаксически управляемой трансляции для “поросячей латыни”.

- 2.14.** В языке программирования C инструкция `for` имеет следующий вид:

`for (expr1 ; expr2 ; expr3) stmt`

Первое выражение выполняется до цикла; обычно оно используется для инициализации индекса цикла. Второе выражение представляет собой тест, выполняемый перед каждой итерацией цикла. Когда это выражение принимает значение 0, происходит выход из цикла (цикл состоит из инструкции `{ stmt expr3 ; }`). Третье выражение выполняется в конце каждой итерации и обычно используется для увеличения индекса цикла. Инструкция `for` подобна

`expr1 ; while (expr2) { stmt expr3 ; }`

Постройте схему синтаксически управляемой трансляции для преобразования инструкции `for` языка C в код стековой машины.

- *2.15.** Рассмотрим следующую инструкцию `for`.

`for i := 1 step 10 - j until 10 * j do j := j + 1`

Эта инструкция может иметь три определения семантики. Во-первых, вычисление предела $10 \cdot j$ и инкремента $10 - j$ может осуществляться однократно, до работы цикла, как в PL/I. Например, если до выполнения цикла $j = 5$, цикл будет выполнен десять раз. Во-вторых, значение предела и инкремент могут вычисляться при каждом прохождении цикла. В этом случае, если перед входом в цикл $j = 5$, цикл станет бесконечным. В-третьих, возможен смысл, как в языке Algol, когда при отрицательном инкременте проверка условия выхода из цикла становится $i < 10 \cdot j$, а не $i > 10 \cdot j$. Для каждого из трех семантических определений постройте свою схему синтаксически управляемой трансляции для преобразования приведенного цикла в код стековой машины.

⁶ Из англо-русского словаря: pig Latin — детск. жарг. “поросячья латынь” (манера коверкать слова, переставляя первый согласный звук в конец слова и добавляя слог ау, например Oodgay orningmay = Good morning). — Прим. перев.

- 2.16.** Рассмотрим следующий фрагмент грамматики для инструкций if-then и if-then-else:

```
stmt → if expr then stmt  
| if expr then stmt else stmt  
| other
```

где **other** соответствует прочим инструкциям языка.

Выполните следующее:

- покажите, что эта грамматика неоднозначна;
- постройте эквивалентную однозначную грамматику, в которой каждый **else** связан с тем ближайшим предшествующим **then**, с которым еще не связан записанный выше другой **else**;
- постройте основанную на этой грамматике схему синтаксически управляемой трансляции для преобразования условных инструкций в код стековой машины.

- *2.17.** Постройте схему синтаксически управляемой трансляции, которая переводит арифметические выражения в инфиксном виде в инфиксную запись без излишних скобок. Приведите аннотированное дерево разбора для входной строки $((1+2) * (3*4)) + 5$.

Программные упражнения

- P2.1.** На основе схемы синтаксически управляемой трансляции из упр. 2.9 реализуйте транслятор для перевода целых чисел в римские.

- P2.2.** Модифицируйте транслятор из раздела 2.9 для получения кода абстрактной стековой машины из раздела 2.8.

- P2.3.** Модифицируйте модуль обработки ошибок из раздела 2.9, чтобы при ошибке он пропускал неверное выражение и переходил к следующему.

- P2.4.** Расширьте транслятор из раздела 2.9 на обработку всех выражений языка Pascal.

- P2.5.** Дополните компилятор из раздела 2.9 для трансляции в код стековой машины инструкций, порождаемых следующей грамматикой:

```
stmt → id := expr  
| if expr then stmt  
| while expr do stmt  
| begin opt_stmts end  
opt_stmts → stmt_list | ε  
stmt_list → stmt_list ; stmt | stmt
```

- *P2.6.** Создайте набор тестовых выражений для компилятора из раздела 2.9, таких, чтобы каждая продукция использовалась в наборе по крайней мере один раз. Создайте тестовую программу, используемую в качестве тестирующего инструмента, и воспользуйтесь ею для запуска компилятора с этими выражениями.

- P2.7.** Создайте набор тестовых инструкций для компилятора из упр. P2.5, таких, чтобы каждая продукция использовалась, по меньшей мере один раз, для генерации некоторой тестовой инструкции. Для запуска компилятора с этими тестовыми выражениями воспользуйтесь программой, разработанной в упр. P2.6.

Библиографические примечания

В этой главе мы коснулись ряда вопросов, которые будут детально рассматриваться в последующих главах книги. Ссылки на литературу по конкретным вопросам будут приведены в соответствующих главах.

Контекстно-свободные грамматики были введены Хомским [79] при изучении естественных языков. Использование контекстно-свободных грамматик для определения синтаксиса языков программирования было развито независимо (см. [454], стр. 162), например, при работе над черновым вариантом Algol 60. Результирующая нотация языка представляла собой вариант контекстно-свободной грамматики. Такую же семантическую нотацию для определения правил санскритской грамматики (между 400 и 200 гг. до н.э.) использовал Панини ([205]).

В статье Д. Кнута [252] было предложено, в знак признания большого вклада Наура в разработку Algol 60 [328], читать аббревиатуру BNF как Backus–Naur Form (форма Бэкуса–Наура) (в то время как ранее она означала Backus Normal Form, нормальная форма Бэкуса).

Синтаксически управляемые определения представляют собой вид индуктивных определений, в которых индукция находится в синтаксической структуре. Как таковые они давно неформально используются в математике, а их применение к языкам программирования началось с Algol 60. Вскоре после этого Айронс [206] создал синтаксически управляемый компилятор.

Анализ методом рекурсивного спуска используется с начала 60-х годов. Бауэр [49] приписывает этот метод Лукасу [295]. Хоар [189] описывает компилятор Algol как “набор процедур, каждая из которых способна обрабатывать одну из синтаксических единиц Algol 60”. Фостер [140] обсуждает в своей работе устранение левой рекурсии из продукции, содержащих семантические действия, не влияющие на значения атрибутов.

Мак-Карти [304] утверждает, что трансляция языка должна базироваться на абстрактном синтаксисе. В своей статье он предлагает читателям самостоятельно убедиться в том, что оконечно-рекурсивная формулировка факториала эквивалентна итеративной программе.

Преимущества разбиения компилятора на начальную и заключительную стадии были представлены в отчете [414], где предложено название UNCOL (от Universal Computer Oriented Language — универсальный компьютерно-ориентированный язык) для универсального промежуточного языка. Однако это так и осталось недостижимым идеалом.

Хороший способ изучить технологии — ознакомиться с кодом существующих компиляторов. К сожалению, коды компиляторов не так уж часто публикуются. Рэнделл и Рассел [363] дали полное описание раннего компилятора Algol. Код компилятора можно также найти в работе [311]. В [45] дается подборка статей по реализации компиляторов языка программирования Pascal. В [335] описаны детали реализации компилятора Pascal P, в [31] — детали генерации кода, а в [457] — код Pascal S, подмножества Pascal, разработанного Виртом для использования в обучающих целях. Кнут [262] дал необычайно ясное и детальное описание транслятора \TeX .

Керниган и Пайк [247] детально описали построение программы настольного калькулятора с применением схемы синтаксически управляемой трансляции и доступного в UNIX инструментария для создания компиляторов. Уравнение (2.17) взято из [422].

ГЛАВА 3

Лексический анализ

В этой главе мы познакомимся с технологиями определения и реализации лексических анализаторов. Простейший способ построения лексического анализатора состоит в создании диаграммы, иллюстрирующей структуру токенов исходного языка, и трансляции ее в программу для поиска токенов. Таким образом можно создавать эффективные лексические анализаторы.

Технологии, используемые при создании лексических анализаторов, могут применяться и в других областях — таких, например, как языки запросов и информационно-поисковые системы. В основе каждого приложения лежит задача определения и создания программы, которая выполняла бы определенные действия при появлении в строке некоторых шаблонов. Поскольку такое шаблонно-управляемое программирование широко используется в различных областях, познакомимся с языком Lex для создания лексических анализаторов. Шаблоны в этом языке определяются регулярными выражениями (*regular expression*), а компилятор для Lex генерирует эффективный конечный автомат, распознающий их.

Регулярные выражения для описания шаблонов используются и в ряде других языков. Например, язык AWK использует регулярные выражения для выбора обрабатываемых входных строк, а оболочки UNIX с помощью этих выражений позволяют пользователю ссылаться на набор файлов. Например, команда UNIX `rm *.o` удаляет все файлы, имена которых заканчиваются на “`.o`”¹.

Программный инструментарий, автоматизирующий построение лексических анализаторов, позволяет программистам использовать регулярные выражения в своих приложениях. Например, Джарвис [210] использовал генератор лексических анализаторов для создания программы, распознающей дефекты печатных плат. Платы сканировались, и результат сканирования преобразовывался в “строку” отрезков. “Лексический анализатор” просматривал полученную строку в поисках шаблона, соответствующего дефектам. Главное достоинство генератора лексических анализаторов заключается в том, что он может использовать самые мощные алгоритмы поиска шаблонов и, таким образом, создавать эффективные лексические анализаторы для программистов, не являющихся экспертами в этой области.

3.1. Роль лексических анализаторов

Лексический анализатор представляет собой первую фазу компилятора. Его основная задача состоит в чтении новых символов и выдаче последовательности токенов, используемых синтаксическим анализатором в своей работе. На рис. 3.1 схематически показано взаимодействие лексического и синтаксического анализаторов, которое обычно реализуется путем создания лексического анализатора в качестве подпрограммы синтаксического анализатора (или подпрограммы, вызываемой им). При получении запроса на следующий токен лексический анализатор считывает входной поток символов до точной идентификации следующего токена.

¹ Выражение `*.o` является вариантом обычной записи регулярных выражений. В упр. 3.10 и 3.14 вы встретитесь с некоторыми часто используемыми вариантами записи регулярных выражений.

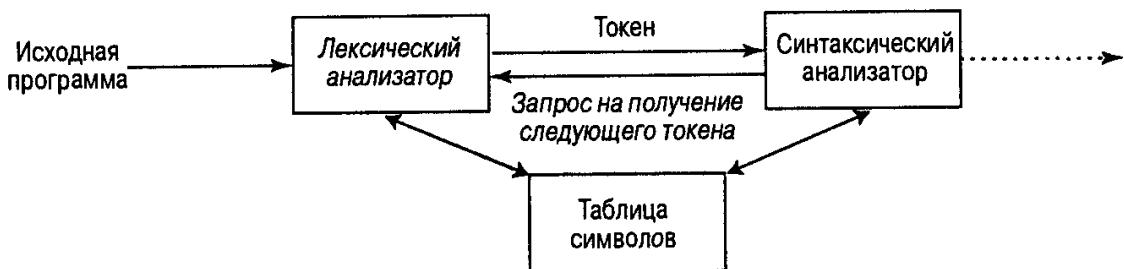


Рис. 3.1. Взаимодействие лексического и синтаксического анализаторов

Поскольку лексический анализатор является частью компилятора, которая считывает исходный текст, он может также выполнять некоторые вторичные задачи интерфейса. К ним, например, относятся удаление из текста исходной программы комментариев и не несущих смысловой нагрузки пробелов (а также символов табуляции и новой строки). Еще одна задача состоит в согласовании сообщений об ошибках компиляции и текста исходной программы. Например, лексический анализатор может подсчитывать количество считанных строк и указать строку, вызвавшую ошибку². В некоторых компиляторах лексический анализатор создает копию текста исходной программы с указанием ошибок. Если исходный язык поддерживает макросы и директивы препроцессора, то они также могут реализовываться лексическим анализатором.

Зачастую работа лексического анализатора состоит из двух фаз — сканирования и лексического анализа. Фаза сканирования отвечает за выполнение простейших задач, в то время как на фазу лексического анализа возлагаются задачи более сложные. Так, например, компилятор Fortran может использовать сканер для удаления из входного текста лишних пробелов.

Задачи лексического анализа

Имеется ряд причин, по которым фаза анализа компиляции разделяется на лексический и синтаксический анализы.

1. Пожалуй, наиболее важной причиной является упрощение разработки. Отделение лексического анализатора от синтаксического часто позволяет упростить одну из фаз анализа. Например, включение в синтаксический анализатор комментариев и пробелов существенно сложнее, чем удаление их лексическим анализатором. При создании нового языка разделение лексических и синтаксических правил может привести к более четкому и ясному построению языка.
2. Увеличивается эффективность компилятора. Отдельный лексический анализатор позволяет создать специализированный и потенциально более эффективный процессор для решения поставленной задачи. Поскольку на чтение исходной программы и разбор ее на токены тратится много времени, специализированные технологии буферизации и обработки токенов могут существенно повысить производительность компилятора.
3. Увеличивается переносимость компилятора. Особенности входного алфавита и другие специфичные характеристики используемых устройств могут ограничивать возможности лексического анализатора. Например, таким образом может быть решена

² К сожалению, особенно в сложных программах и сложных языках программирования такое указание ошибки не всегда достоверно — иногда ошибка может быть распознана только после считывания нескольких строк. Такие ситуации могут встречаться и в коммерческих компиляторах, хотя в этом случае они достаточно редки. — Прим. ред.

проблема специальных нестандартных символов, таких как \uparrow в Pascal (или “триграфы” в C — Прим. ред.).

Существует ряд специализированных инструментов, предназначенных для автоматизации построения лексических и синтаксических анализаторов (в том случае, когда они разделены). Некоторые из этих инструментов мы рассмотрим в данной книге.

Токены, шаблоны, лексемы

Говоря о лексическом анализе, мы используем термины “токен”, “шаблон” и “лексема” каждый в своем собственном смысле. Примеры их использования приведены на рис. 3.2. Вообще говоря, во входном потоке существует набор строк, для которых в качестве выхода получается один и тот же токен. Этот набор строк описывается правилом, именуемым *шаблоном* (pattern), связанным с токеном. О шаблоне говорят, что он *соответствует* (match) каждой строке в наборе. Лексема же представляет собой последовательность символов исходной программы, которая соответствует шаблону токена. Например, в инструкции Pascal

```
const pi = 3.1416;
```

подстрока `pi` представляет собой лексему токена “идентификатор”.

ТОКЕН	ПРИМЕР ЛЕКСЕМ	НЕФОРМАЛЬНОЕ ОПИСАНИЕ ШАБЛОНА
const	const	const
if	if	if
relation	<, <=, =, >, >=	< или <= или = или > или >=
id	pi, count, D2	Буква, за которой следуют буквы и цифры
num	3.1416, 0, 6.02E23	Любая числовая константа
literal	"core dumped"	Любые символы между парными кавычками, исключая сами кавычки

Рис. 3.2. Примеры токенов

Мы рассматриваем токены как терминальные символы грамматики исходного языка (используя выделение полужирным шрифтом). Лексемы, соответствующие шаблонам токенов, представляют в исходной программе строки символов, которые рассматриваются вместе как лексическая единица.

В большинстве языков программирования в качестве токенов выступают ключевые слова, операторы, идентификаторы, константы, строки литералов и символы пунктуации, например скобки, запятые и точки с запятыми. В приведенном выше примере, когда во входном потоке встречается последовательность символов `pi`, синтаксическому анализатору возвращается токен, представляющий идентификатор. Возврат токена чаще всего реализуется путем передачи целого числа, соответствующего токену (это целое число (рис. 3.2) соответствует выделенному полужирным шрифтом `id`).

Шаблон — правило, описывающее набор лексем, которые могут представлять определенный токен в исходной программе. Так, шаблон токена `const` (рис. 3.2) представляет собой просто строку `const`, являющуюся ключевым словом. Шаблон токена `relation` — набор всех шести операторов сравнения в Pascal. Для точного описания шаблонов сложных токенов типа `id` (для идентификатора) и `num` (для числа) нужно использовать регулярные выражения, которым посвящен раздел 3.3.

Языковые соглашения существенно усложняют лексический анализ. Языки типа Fortran требуют, чтобы некоторые конструкции языка располагались в определенных позициях входной строки. Таким образом, выравнивание лексем может играть существенную роль при определении корректности входной программы. Тенденция современных языков к свободному формату входной информации позволяет не обращать внимания на позиционирование и уменьшает важность и актуальность этого аспекта лексического анализа.

Обработка пробелов также сильно разнится в разных языках программирования. В некоторых языках, таких как Fortran или Algol 68, пробелы не играют роли нигде, за исключением строк-литералов, и могут быть добавлены в любом месте для повышения удобочитаемости программы. Такие соглашения могут существенно усложнить задачу идентификации токенов.

Классическим примером, иллюстрирующим потенциальную сложность распознавания токенов, является инструкция DO в Fortran. В инструкции

DO 5 I = 1.25

до тех пор, пока не встретимся с десятичной точкой, мы не можем сказать определенно, является ли DO ключевым словом или частью идентификатора DO5I. В инструкции же

DO 5 I = 1,25

мы имеем семь токенов, соответствующих ключевому слову DO, метке инструкции 5, идентификатору I, оператору =, константе 1, запятой и константе 25. В данном случае, пока не встретим запятую, мы не можем утверждать, что DO — ключевое слово. Для устранения такой неопределенности в Fortran 77 позволено использовать необязательную запятую между меткой и индексом инструкции DO. Применение этой запятой всячески приветствуется и поддерживается, поскольку она помогает сделать инструкцию DO более ясной и удобочитаемой.

Во многих языках ряд строк является зарезервированным, т.е. их смысл предопределен и не может быть изменен пользователем. Если ключевые слова не зарезервированы, лексический анализатор должен уметь определить, с чем он имеет дело — ключевым словом или идентификатором, введенным пользователем. В PL/I ключевые слова не зарезервированы, и, соответственно, правила лексического разбора в этом языке очень сложны, что можно увидеть на примере следующей инструкции PL/I:

IF THEN THEN THEN = ELSE; ELSE ELSE = THEN;

Атрибуты токенов

Если шаблону соответствует несколько лексем, лексический анализатор должен обеспечить дополнительную информацию о лексемах для последующих фаз компиляции. Например, шаблон **num** соответствует как строке 0, так и 1, и при генерации кода крайне важно знать, какая именно строка соответствует токену.

Лексический анализатор хранит информацию о токенах в связанных с ними атрибутах. Токены определяют работу синтаксического анализатора; атрибуты — трансляцию токенов. На практике токены обычно имеют единственный атрибут — указатель на запись в таблице символов, в которой хранится информация о токене. Для диагностических целей нам могут понадобиться как лексемы идентификаторов, так и номера строк, в которых они впервые появились в программе. Вся эта (и другая) информация может храниться в записях в таблице символов.

Пример 3.1

Токены и связанные с ними значения атрибутов для инструкции Fortran

E = M * C ** 2

записываются как последовательность пар:

```
<id, указатель на запись в таблице символов для E>
<assign_op, >
<id, указатель на запись в таблице символов для M>
<mult_op, >
<id, указатель на запись в таблице символов для C>
<exp_op, >
<num, целое значение 2>.
```

Заметим, что в некоторых парах нет необходимости в значении атрибута — первого компонента вполне достаточно, чтобы идентифицировать лексему. В этом небольшом примере токен num задан атрибутом с целым значением. Компилятор может хранить строку символов, составляющих число, в таблице символов, сделав, таким образом, атрибут токена num указателем на запись в таблице символов. □

Лексические ошибки

На уровне лексического анализатора определяются только некоторые ошибки, поскольку лексический анализатор рассматривает исходный текст программы в ограниченном контексте. Если в программе на языке С строка `f i` впервые встретится в контексте

`fi (a == f(x)) . . .`

лексический анализатор не сможет определить, что именно представляет собой `fi` — неверно записанное слово `i f` или необъявленный идентификатор функции. Поскольку `fi` является корректным идентификатором, лексический анализатор должен вернуть токен для идентификатора и предоставить обработку ошибки другой части компилятора.

Однако представим, что лексический анализатор не способен продолжать работу, поскольку ни один из шаблонов не соответствует оставшейся части входного потока. Вероятно, простейшим в этой ситуации будет восстановление в “режиме паники”. Мы просто пропускаем входные символы до тех пор, пока лексический анализатор не встретит распознаваемый токен. Иногда это запутывает синтаксический анализатор, но для интерактивной среды данная технология может оказаться вполне адекватной.

Существуют и другие возможные действия по восстановлению после ошибки.

1. Удаление лишних символов.
2. Вставка пропущенных символов.
3. Замена неверного символа верным.
4. Перестановка двух соседних символов.

При восстановлении корректного входного потока могут выполняться различные преобразования. Простейшая стратегия состоит в проверке, не может ли начало оставшейся части входного потока быть заменено корректной лексемой путем единственного преобразования. Эта стратегия полагает, что большинство лексических ошибок вызвано единственным неверным преобразованием (по сути, опечаткой), и это предположение подтверждается практикой.

Один из способов восстановления после ошибок в программе состоит в определении минимального количества преобразований ошибок, требующихся для превращения неправильной программы в синтаксически корректную. Мы говорим, что неверная программа содержит k ошибок, если кратчайшая последовательность преобразований ошибок, отображающая программу в синтаксически корректную, имеет длину k . Критерий минимальности коррекций ошибок представляет собой удобную теоретическую меру, однако далеко не всегда применимую на практике с учетом сложности ее определения. Однако несколько экспериментальных компиляторов использовали критерий минимального расстояния в пространстве ошибок для коррекции ошибочных программ.³

3.2. Буферизация ввода

В этом разделе поговорим об эффективности буферизации ввода. Вначале упомянем о двухбуферной схеме ввода, использование которой особенно полезно, когда при идентификации токенов требуется “заглядывание вперед” во входной поток. Затем рассмотрим некоторые технологии ускорения работы лексического анализатора, например использование ограничителей для маркировки конца буфера.

Имеется три основных способа реализации лексического анализатора.

1. Использование генератора лексических анализаторов, такого как Lex (о котором говорим в разделе 3.5), для создания лексического анализатора по спецификациям, основанным на регулярных выражениях. В этом случае генератор предоставляет функции для чтения и буферизации ввода.
2. Написание лексического анализатора на подходящем языке программирования с использованием возможностей ввода-вывода этого языка для чтения входной информации.
3. Написание лексического анализатора на языке ассемблера и явное управление процессом чтения входной информации.

Эти методы перечислены в порядке усложнения их реализации. К сожалению, более трудные для реализации подходы часто дают более быстрые лексические анализаторы. Поскольку лексический анализатор — единственная фаза компилятора, которая читает исходную программу, стоит поговорить о ней подробно, несмотря на другие, более сложные фазы компиляции. Хотя эта глава в основном посвящена первому методу создания лексических анализаторов — с помощью генераторов, мы рассмотрим также технологии, которые полезны при ручной разработке. В разделе 3.4 описываются диаграммы переходов, представляющие собой полезную концепцию при создании лексических анализаторов вручную.

Пары буферов

Для многих исходных языков часто требуется просмотреть входной поток на несколько символов за пределами лексемы, перед тем как найти соответствующий токен. Лексический анализатор в главе 2, “Простой однопроходный компилятор”, использовал функцию `ungetc` для возврата символа назад во входной поток. Поскольку на переме-

³ Следует понимать, что исправленная программа, будучи корректной синтаксически, может оказаться некорректной семантически или, что еще хуже, давать неверный результат. — Прим. ред.

щение символов может быть затрачено достаточно много времени, для уменьшения накладных расходов на обработку входного потока была разработана специальная буферизующая технология. Могут использоваться различные схемы буферизации, но поскольку технологии буферизации отчасти зависят от параметров системы, здесь мы рассмотрим только основные принципы одного класса технологий.

Мы используем буфер, разделенный на две N -символьные половины, как показано на рис. 3.3. Обычно N равно количеству символов в дисковом блоке, т.е. 1024 или 4096.

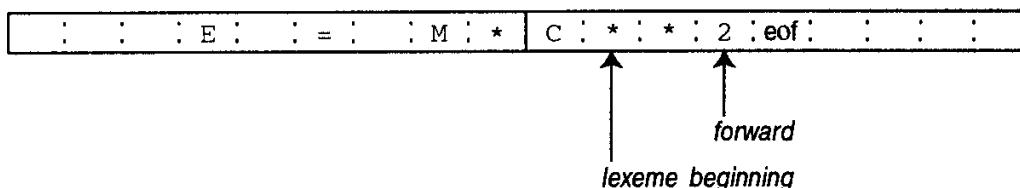


Рис. 3.3. Входной буфер, состоящий из двух частей

Мы считываем N входных символов в каждую половину буфера с помощью одной системной команды чтения (а не вызывая команду чтения для каждого входного символа отдельно). Если осталось считать меньше N символов, то в буфер вносится специальный символ `eof`, как показано на рис. 3.3. Таким образом, `eof` указывает конец исходного файла и отличается от всех прочих входных символов.

В процессе работы поддерживаются два указателя. Стока символов между этими указателями представляет собой текущую лексему. Изначально оба указателя указывают на первый символ следующей лексемы. Один указатель (*forward*) перемещается вперед, сканируя входную строку до тех пор, пока не будет обнаружено соответствие шаблону. Как только определена очередная лексема, указатель *forward* указывает на ее крайний справа символ. После обработки лексемы оба указателя указывают на символ, непосредственно следующий за лексемой. В этой схеме комментарии и пробелы могут рассматриваться как шаблоны, не порождающие токенов.

Если указатель *forward* пересекает границу между половинами буфера, правая половина заполняется N новыми символами из входного потока. Если же указатель *forward* выходит за пределы правой границы буфера, новые N символовчитываются в левую половину буфера, а указатель перемещается в начало буфера.

Эта схема буферизации в большинстве случаев работает вполне прилично, но с ограниченным просмотром. Эта ограниченность может сделать невозможным распознавание токенов, когда путь, который должен пройти указатель *forward*, превышает длину буфера. Например, если в программе на PL/I мы встретим строку

```
DECLARE ( ARG1, ARG2, . . . ARGn )
```

то не сможем определить, является ли `DECLARE` ключевым словом или именем массива, пока не встретим первый символ после правой закрывающей скобки. В любом случае лексема завершается последним символом `E` в слове `DECLARE`, но количество символов, которое следует просмотреть, пропорционально количеству аргументов (которое теоретически не ограничено).

```
if forward в конце первой половины буфера then begin
    загрузить вторую половину буфера;
    forward := forward + 1
end
else if forward в конце второй половины буфера then begin
```

```

загрузить первую половину буфера;
переместить forward к началу первой половины
end
else forward := forward + 1;

```

Рис. 3.4. Код продвижения указателя forward

Ограничители

При использовании схемы, приведенной на рис. 3.3, необходимо при каждом перемещении указателя проверять, не покидает ли он текущую половину буфера. Таким образом, код перемещения указателя *forward* должен постоянно выполнять проверки, приведенные в коде на рис. 3.4.

За исключением конечных позиций в половинах буфера, код на рис. 3.4 требует двух проверок при каждом перемещении указателя. Мы можем свести две проверки к одной, если добавить к каждой из половин буфера символ-ограничитель (*sentinel*) в конце. Ограничитель представляет собой специальный символ, который не может встретиться в исходной программе. Естественным выбором для ограничителя является *eof*. На рис. 3.5 показан тот же буфер, что и на рис. 3.3, но с дополнительными ограничителями.

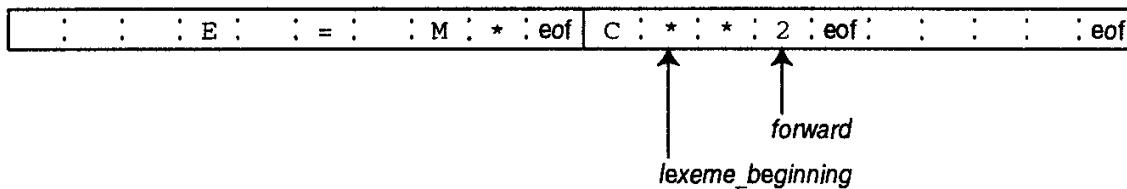


Рис. 3.5. Использование ограничителей в конце каждой половины буфера

При использовании ограничителей для перемещения указателя *forward* можно применить код, приведенный на рис. 3.6 (кроме прочего, этот код проверяет условие конца исходного файла). В большинстве случаев будет выполняться только одна проверка — не указывает ли *forward* на символ *eof*. Только при достижении конца части буфера или конца файла выполняются дополнительные проверки. Поскольку между символами *eof* располагается *N* символов исходного текста, среднее количество проверок на один входящий символ очень близко к 1.

```

forward := forward + 1;
if forward† = eof then begin
    if forward в конце первой половины буфера then begin
        загрузить вторую половину буфера;
        forward := forward + 1
    end
    else if forward в конце второй половины буфера then begin
        загрузить первую половину буфера;
        переместить forward к началу первой половины
    end
    else /* eof встретился не в конце буфера — конец ввода */
end

```

Рис. 3.6. Код продвижения указателя forward с использованием ограничителей

Требуется также решить, каким образом обрабатывать символ, сканируемый указателем *forward*, в зависимости от того, является ли он концом токена, символом ключевого слова или чем-то иным. Один из способов структурировать эти проверки — использовать инструкцию типа *case* (если язык реализации это позволяет). В таком случае проверка

```
if forward↑ = eof
```

является всего лишь одним из вариантов *case*-структур.

3.3. Определение токенов

В определении токенов важную роль играют регулярные выражения. Каждый шаблон соответствует множеству строк, так что регулярные выражения можно рассматривать как имена таких множеств. В разделе 3.5 такой способ записи будет расширен до шаблонно-управляемого языка для лексического анализа.

Строки и языки

Термин *алфавит*, или *класс символов*, обозначает любое конечное множество символов. Типичным примером символов могут служить буквы или алфавитно-цифровые символы. Множество $\{0,1\}$ представляет собой *бинарный алфавит*. ASCII и EBCDIC являются примерами компьютерных алфавитов.

Строка над некоторым алфавитом — это конечная последовательность символов, взятых из алфавита. В теории языков термин *предложение* (*sentence*) и *слово* (*word*) часто используются как синонимы термина “строка”. Длина строки s , обычно обозначаемая как $|s|$, равна количеству символов в строке. Например, длина строки *banana* равна шести. *Пустая строка*, обозначаемая как ϵ , представляет собой специальную строку нулевой длины. Некоторые общеупотребительные термины, связанные с частями строк, приведены на рис. 3.7.

ТЕРМИН	ОПРЕДЕЛЕНИЕ
<i>Префикс</i> строки s (prefix)	Строка, полученная удалением нуля или нескольких последних символов строки s ; например, <i>ba</i> n является префиксом строки <i>banana</i>
<i>Суффикс</i> строки s (suffix)	Строка, полученная удалением нуля или нескольких первых символов строки s ; например, <i>ana</i> является суффиксом строки <i>banana</i>
<i>Подстрока</i> строки s (substring)	Строка, полученная удалением префикса и суффикса строки s ; например, <i>na</i> n является подстрокой строки <i>banana</i> . Каждый префикс и каждый суффикс также являются подстроками, но не каждая подстрока есть префиксом или суффиксом. Для каждой строки s строки s и ϵ являются суффиксами, префиксами и подстроками строки s
<i>Правильные</i> префикс, суффикс и подстрока строки s (proper ...)	Любая непустая строка x , которая является соответственно префиксом, суффиксом и подстрокой строки s и не совпадает со строкой s

Рис. 3.7. Термины, описывающие части строк

ТЕРМИН	ОПРЕДЕЛЕНИЕ
Подпоследовательность строки s (subsequence)	Любая строка, сформированная удалением нуля или нескольких (не обязательно последовательных) символов из строки s . Так, $baaa$ является подпоследовательностью строки $banana$

Рис. 3.7. Термины, описывающие части строк (окончание)

Термин **язык** (*language*) обозначает любое множество строк над некоторым фиксированным алфавитом. Такое определение весьма широко. Абстрактные языки типа \emptyset (пустое множество) или $\{\epsilon\}$ (множество, содержащее только пустую строку) также подходят под это определение. Точно так же языками являются и множество всех синтаксически корректных программ на языке Pascal, и все грамматически корректные предложения английского языка — хотя два последних множества существенно сложнее определить. Заметим также, что такое определение не описывает значение (смысла) строк языка. Методы описания смысла строк обсуждаются в главе 5, “Синтаксически управляемая трансляция”. Если x и y — строки, то **конкатенация** строк x и y , которая записывается как xy , является строкой, сформированной путем дописывания y к x . Например, если $x = \text{dog}$, а $y = \text{house}$, то $xy = \text{doghouse}$. Пустая строка представляет собой единичный элемент по отношению к операции конкатенации, т.е. для любой строки s справедливо соотношение $s\epsilon = \epsilon s = s$.

Если рассматривать конкатенацию как операцию “умножения”, то можно определить операцию “возвведения в степень” следующим образом. Определим, что $s^0 = \epsilon$, а для $i > 0$ определим s^i как $s^{i-1}s$. Поскольку ϵs — просто s , то $s^1 = s$. Соответственно, $s^2 = ss$, $s^3 = sss$ и т.д.

Операции над языками

Имеется ряд важных операций, выполняемых над языками. С точки зрения лексического анализа нас в первую очередь интересуют объединение, конкатенация и замыкание (рис. 3.8). Можно также обобщить оператор возвведения в степень для языка, определив L^0 как $\{\epsilon\}$, а L^i — как $L^{i-1}L$. Таким образом, L^i представляет собой L , конкатенированный с самим собой $i-1$ раз.

ОПЕРАЦИЯ	ОПРЕДЕЛЕНИЕ
Объединение (union) L и M — $L \cup M$	$L \cup M = \{s \mid s \in L \text{ или } s \in M\}$
Конкатенация (concatenation) L и M — LM	$LM = \{st \mid s \in L \text{ и } t \in M\}$
Замыкание Клини (Kleene closure); или итерация L — L^*	$L^* = \bigcup_{i=0}^{\infty} L^i$ L^* обозначает “нуль или более конкатенаций L ”
Позитивное замыкание (positive closure) L — L^+	$L^+ = \bigcup_{i=1}^{\infty} L^i$ L^+ обозначает “одна или более конкатенаций L ”

Рис. 3.8. Определения операций над языками

Пример 3.2

Пусть L — множество $\{A, B, \dots, Z, a, b, \dots, z\}$, а D — множество $\{0, 1, 2, \dots, 9\}$. Мы можем рассматривать множества L и D двояко. С одной стороны, L представляет собой алфавит, состоящий из набора прописных и строчных букв, а D — алфавит, состоящий из цифр. С другой стороны, поскольку символ можно рассматривать как строку единичной длины, множества L и D представляют собой конечные языки. Вот несколько примеров новых языков, созданных из L и D с применением операторов, приведенных на рис. 3.8.

1. $L \cup D$ представляет собой множество букв и цифр.
2. LD — множество строк, состоящих из буквы, за которой следует цифра.
3. L^4 — множество всех четырехбуквенных строк.
4. L^* — множество всех строк из букв, включая пустую строку ϵ .
5. $L(L \cup D)^*$ — множество всех строк из букв и цифр, начинающихся с буквы.
6. D^+ — множество всех строк из одной или нескольких цифр.

□

Регулярные выражения

В Pascal идентификатор представляет собой букву, за которой следует нуль или несколько букв или цифр; следовательно, идентификатор является элементом множества, определенного в п. 5 примера 3.2. В этом разделе представлен способ записи, именуемый регулярными выражениями, который позволяет точно определять подобные множества. С помощью этого способа записи можно определить идентификаторы Pascal как

`letter (letter | digit) *`

Вертикальная черта означает “или”, скобки используются для группирования подвыражений, а непосредственное соседство `letter` с оставшейся частью выражения означает конкатенацию.

Регулярное выражение строится из более простых регулярных выражений с использованием набора правил. Каждое регулярное выражение r обозначает, или задает язык $L(r)$. Правила определяют, каким образом из языков, заданных подвыражениями r , формируется $L(r)$.

Рассмотрим правила, которые определяют *регулярные выражения над алфавитом Σ* .

1. ϵ представляет собой регулярное выражение, обозначающее $\{\epsilon\}$, т.е. множество, содержащее пустую строку.
2. Если a является символом из Σ , то a — регулярное выражение, обозначающее $\{a\}$, т.е. множество, содержащее строку a . Хотя мы используем одну и ту же запись, технически регулярное выражение a отличается от строки a и символа a . Говорим ли мы о регулярном выражении, строке или символе — становится понятно из контекста.
3. Предположим, что r и s — регулярные выражения, обозначающие языки $L(r)$ и $L(s)$. Тогда
 - $(r) | (s)$ представляет собой регулярное выражение, обозначающее $L(r) \cup L(s)$.
 - $(r)(s)$ — регулярное выражение, обозначающее $L(r)L(s)$.

- $(r)^*$ — регулярное выражение, обозначающее $(L(r))^*$.
- (r) — регулярное выражение, обозначающее $L(r)$.⁴

Язык, задаваемый регулярным выражением, называется *регулярным множеством*.

Определение регулярного выражения является примером рекурсивного определения. Правила (1) и (2) формируют базис определения; используем термин *базовый символ* для ϵ или символа из Σ , появляющихся в регулярном выражении. Правило (3) обеспечивает шаг индукции.

Излишние скобки в регулярном выражении могут быть устранины, если принять следующие соглашения.

1. Унарный оператор $*$ имеет высший приоритет и левоассоциативен.
2. Конкатенация имеет второй по значимости приоритет и левоассоциативна.
3. $|$ (объединение) имеет низший приоритет и левоассоциативно.

При этих соглашениях запись $(a)|((b)^*(c))$ эквивалентна $a|b^*c$. Оба выражения обозначают множество строк, которые представляют собой либо единственный a , либо нуль или несколько b , за которыми следует единственный c .

Пример 3.3

Пусть $\Sigma = \{a,b\}$.

1. Регулярное выражение $a|b$ обозначает множество $\{a,b\}$.
2. Регулярное выражение $(a|b)(a|b)$ — $\{aa, ab, ba, bb\}$, множество всех строк из a и b длины 2. Другое регулярное выражение для того же множества — $aa|ab|ba|bb$.
3. Регулярное выражение a^* — множество всех строк из нуля или более a , т.е. $\{\epsilon, a, aa, aaa, \dots\}$.
4. Регулярное выражение $(a|b)^*$ обозначает множество всех строк, содержащих нуль или несколько экземпляров a и b , т.е. множество всех строк из a и b . Другое регулярное выражение для этого множества — $(a^*b^*)^*$.
5. Регулярное выражение $a|a^*b$ — множество, содержащее строку a и все строки, состоящие из нуля или нескольких a , за которыми следует b . \square

Если два регулярных выражения r и s задают один и тот же язык, то r и s называются *эквивалентными*, т.е. $r = s$. Например, $(a|b)=(b|a)$.

Имеется ряд алгебраических законов, используемых для преобразования регулярных выражений в эквивалентные. На рис. 3.9 приведены некоторые из этих законов для регулярных выражений r , s и t .

АКСИОМА	ОПИСАНИЕ
$r s = s r$	Оператор $ $ коммутативен
$r (s t) = (r s) t$	Оператор $ $ ассоциативен
$(rs)t = r(st)$	Конкатенация ассоциативна

Рис. 3.9. Алгебраические свойства регулярных выражений

⁴ Это правило гласит, что при необходимости можно поместить регулярное выражение в дополнительные скобки.

АКСИОМА	ОПИСАНИЕ
$r(s \mid t) = rs \mid rt$ $(s \mid t)r = sr \mid tr$	Конкатенация дистрибутивна над \mid
$\epsilon r = r$	ϵ является единичным элементом по отношению к конкатенации
$r\epsilon = r$	
$r^* = (r \mid \epsilon)^*$	Связь между $*$ и ϵ
$r^{**} = r^*$	Оператор $*$ идемпотентен

Рис. 3.9. Алгебраические свойства регулярных выражений (окончание)

Регулярные определения

Для удобства записи регулярным выражениям можно давать имена и определять регулярные выражения с использованием этих имен так, как если бы это были символы. Если Σ является алфавитом базовых символов, то *регулярное определение* представляет собой последовательность вида

$$\begin{aligned} d_1 &\rightarrow r_1 \\ d_2 &\rightarrow r_2 \\ \dots \\ d_n &\rightarrow r_n \end{aligned}$$

где каждое d_i — индивидуальное имя, а каждое r_i — регулярное выражение над символами из $\Sigma \cup \{d_1, d_2, \dots, d_{i-1}\}$, т.е. базовыми символами и уже определенными именами. Ограничива каждое r_i символами из Σ и ранее определенными именами, можно построить регулярное выражение над Σ для любого r_i , заменяя (возможно, неоднократно) имена регулярных выражений обозначенными ими выражениями. Если r_i использует d_j для некоторого $j \geq i$, то r_i может оказаться определено рекурсивно, и подстановка никогда не завершится.

Для того чтобы отличить имена от символов, имена в регулярных выражениях выделяются полужирным шрифтом.

Пример 3.4

Как говорилось ранее, множество идентификаторов Pascal представляет собой множество строк из букв и цифр, начинающихся с буквы. Вот регулярное определение этого множества:

$$\begin{aligned} \text{letter} &\rightarrow A \mid B \mid \dots \mid Z \mid a \mid b \mid \dots \mid z \\ \text{digit} &\rightarrow 0 \mid 1 \mid \dots \mid 9 \\ \text{id} &\rightarrow \text{letter} (\text{letter} \mid \text{digit})^* \end{aligned}$$

□

Пример 3.5

Беззнаковые числа в Pascal представляют собой строки типа 5280, 39.37, 6.336E4 или 1.894E-4. Следующее регулярное определение обеспечивает точную спецификацию этого класса строк:

```

digit → 0 | 1 | . . . | 9
digits → digit digit *
optional_fraction → . digits | ε
optional_exponent → ( E ( + | - | ε ) digits ) | ε
num → digits optional_fraction optional_exponent

```

Данное определение гласит, что **optional_fraction** либо представляет собой десятичную точку, за которой следует одна или несколько цифр, либо отсутствует (является пустой строкой). В случае наличия **optional_exponent** представляет собой Е, за которым следует необязательный знак + или – и одна или несколько цифр. Заметьте, что за точкой должна следовать цифра, т.е. запись 1. некорректна, в отличие от записи 1.0. □

Сокращения

Некоторые конструкции встречаются в регулярных выражениях настолько часто, что для них введены специальные сокращения.

- Один или несколько экземпляров.* Унарный постфиксный оператор **+** означает “один или более экземпляров”. Если r — регулярное выражение, означающее язык $L(r)$, то $(r)^+$ представляет собой регулярное выражение, описывающее язык $(L(r))^+$. Таким образом, регулярное выражение a^+ описывает множество всех строк из одного или более a . Оператор **+** имеет тот же приоритет и ассоциативность, что и оператор *****. Два алгебраических тождества $r^* = r^+ \mid \epsilon$ и $r^+ = rr^*$ связывают замыкание Клини и позитивное замыкание.
- Нуль или один экземпляр.* Унарный постфиксный оператор **?** означает “нуль или один экземпляр”. Запись $r?$ представляет собой сокращенную запись $r \mid \epsilon$. Если r — регулярное выражение, то $(r)?$ — регулярное выражение, обозначающее язык $L(r) \cup \{\epsilon\}$. Например, используя операторы **+** и **?**, мы можем переписать регулярное определение для **num** из примера 3.5 как

```

digit → 0 | 1 | . . . | 9
digits → digit +
optional_fraction → (. digits)?
optional_exponent → ( E ( + | - )? digits )?
num → digits optional_fraction optional_exponent

```

- Классы символов.* Запись [abc], где a, b и c — символы алфавита, означает регулярное выражение $a \mid b \mid c$. Сокращенный класс символов типа [a-z] означает регулярное выражение $a \mid b \mid \dots \mid z$. С использованием классов символов можно описать идентификаторы как строки, заданные регулярным выражением [A-Za-z] [A-Za-z0-9]*.

Нерегулярные множества

Некоторые языки не могут быть описаны регулярными выражениями. Для иллюстрации ограниченности описательной мощи регулярных выражений приведем примеры конструкций языков программирования, которые не могут быть описаны посредством регулярных выражений. Доказательство этого утверждения можно найти в библиографии.

Регулярные выражения не могут быть использованы для описания сбалансированных или вложенных конструкций. Например, с одной стороны, множество всех строк из сбалансированных скобок не может быть описано регулярным выражением. С другой стороны, это множество может быть описано посредством контекстно-свободной грамматики.

Повторяющиеся строки также не могут быть описаны регулярными выражениями.
Множество

$\{w^* \mid w \text{ — строка из } a \text{ и } b\}$

не может быть описано ни каким-либо регулярным выражением, ни контекстно-свободной грамматикой.

Регулярные выражения могут использоваться для описания только фиксированного (или неопределенного) количества повторений данной конструкции. Два произвольных числа не могут сравниваться в контексте регулярных выражений для выяснения, равны они или нет. Таким образом, мы не можем описать строки Холлерита вида $nHa_1a_2\dots a_n$ из ранних версий Fortran посредством регулярных выражений, поскольку количество символов после H должно соответствовать десятичному числу перед H .

3.4. Распознавание токенов

В предыдущем разделе мы рассмотрели проблему определения (спецификации) токенов. В этом разделе мы зададимся вопросом, как их распознать. Здесь в качестве примера используется язык, порождаемый следующей грамматикой.

Пример 3.6

Рассмотрим следующий фрагмент грамматики:

```
stmt → if expr then stmt
      | if expr then stmt else stmt
      | ε
expr → term relop term
      | term
term → id
      | num
```

где терминалы **if**, **then**, **else**, **relop**, **id** и **num** порождают множества строк, задаваемых следующими регулярными определениями:

```
if → if
then → then
else → else
relop → < | <= | = | <> | > | >=
id → letter ( letter | digit )*
num → digit+ (. digit+)? (E(+ | -)? digit+)?
```

где **letter** и **digit** определены так же, как и ранее.

Для этого фрагмента языка лексический анализатор будет распознавать ключевые слова **if**, **then**, **else**, а также лексемы, обозначенные **relop**, **id** и **num**. Для простоты считаем, что ключевые слова зарезервированы и, таким образом, не могут использовать-

ся в качестве идентификаторов. Как и в примере 3.5, num представляет беззнаковые целые и действительные числа Pascal.

Предположим также, что лексемы разделены “пустым пространством”, состоящим из непустых последовательностей пробелов, символов табуляций и новой строки. Наш лексический анализатор будет отбрасывать эти символы путем сравнения строки с регулярным определением ws, приведенным ниже:

```
delim → blank | tab | newline
ws → delim+
```

Если находится соответствие ws, лексический анализатор не возвращает токен синтаксическому анализатору. Вместо этого происходит поиск нового токена, следующего за пробелами, и передача его синтаксическому анализатору.

Наша цель — построение лексического анализатора, который сможет выделять из входного буфера лексему для следующего токена и, используя таблицу трансляции на рис. 3.10, выдавать пары, состоящие из токена и атрибута-значения. Атрибут-значение для операторов отношения определяется символическими константами LT, LE, EQ, NE, GT, GE. □

РЕГУЛЯРНОЕ ВЫРАЖЕНИЕ	ТОКЕН	АТРИБУТ-ЗНАЧЕНИЕ
ws	—	—
if	if	—
then	then	—
else	else	—
id	id	указатель на запись в таблице
num	num	указатель на запись в таблице
<	relop	LT
<=	relop	LE
=	relop	EQ
<>	relop	NE
>	relop	GT
>=	relop	GE

Рис. 3.10. Шаблоны регулярных выражений для токенов

Диаграммы переходов

В качестве промежуточного шага при создании лексического анализатора вначале построим стилизованные блок-схемы, называемые *диаграммами переходов* (transition diagram). Диаграмма переходов изображает действия, выполняемые лексическим анализатором при вызове его синтаксическим анализатором для получения очередного токена, как показано на рис. 3.1. Предположим, что входной буфер подобен изображеному на рис. 3.3 и указатель начала лексемы указывает на символ, следующий за последней найденной лексемой. В данном случае мы используем диаграмму переходов для отслеживания информации о символах, которые были сканированы при перемещении указателя *forward*. Чтобы сделать это, мы перемещаемся при чтении символов от позиции к позиции по диаграмме переходов.

Позиции в диаграмме переходов изображаются кружками и называются *состояниями* (states). Состояния соединены стрелками, называемыми *дугами* (edges). Выходящие из

сстояния s дуги имеют метки, указывающие входные символы, которые могут появиться во входном потоке по достижении состояния s . Метка *other* означает появление любого символа, не указанного другими исходящими дугами.

Мы считаем диаграммы переходов из этого раздела детерминированными, т.е. ни один символ не может быть меткой двух исходящих из одного состояния дуг. В разделе 3.5 мы ослабим это условие, облегчая жизнь проектировщика лексических анализаторов и, при наличии хорошего инструментария, не усложняя жизнь разработчика.

Одно из состояний имеет метку *start* — это начальное состояние диаграммы переходов в момент начала распознавания токена. Некоторые состояния имеют действия, выполняемые при достижении этих состояний. При попадании в некоторое состояние мы считываем следующий входной символ и, если имеется исходящая дуга с меткой, соответствующей этому символу, перемещаемся по ней в следующее состояние. Если такой дуги нет, то входящий символ некорректен и произошла ошибка.

На рис. 3.11 показана диаграмма переходов для шаблонов $>=$ и $>$. Диаграмма работает следующим образом. Работа начинается в состоянии 0, в котором считывается следующий символ из входного потока. Дуга, помеченная $>$, ведет в состояние 6, если этот символ — “ $>$ ”. В противном случае мы не можем распознать $>$ или $>=$.

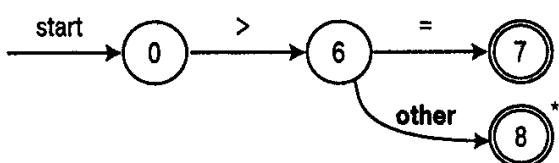


Рис. 3.11. Диаграмма переходов для $>=$

По достижении состояния 6 считываем следующий входной символ. Дуга, помеченная как $=$, приводит из состояния 6 в состояние 7, если считанный символ — “ $=$ ”. В противном случае по дуге *other* попадаем в состояние 8. Двойной кружок, который изображает состояние 7, показывает, что это — допускающее, или заключительное состояние, в котором найден токен $>=$.

Заметим, что символ $>$ с другим дополнительным символом приводит в другое допускающее состояние 8. Поскольку этот другой символ не является частью токена, он должен быть возвращен во входной поток, что и указано звездочкой около этого состояния.

Вообще говоря, может существовать ряд диаграмм переходов, каждая из которых определяет группу токенов. Если при перемещении по диаграмме переходов происходит сбой, мы возвращаем указатель *forward* к тому месту, где он был в стартовом состоянии данной диаграммы, и переходим к следующей диаграмме. Поскольку указатель на начало лексемы и указатель *forward* в стартовом состоянии указывают на одно и то же место, такой возврат осуществляется очень просто — возвратом в позицию, которая определяется указателем *lexeme_beginning*. Если сбой происходит во всех диаграммах, вызывается программа восстановления после ошибки.

Пример 3.7

Диаграмма переходов для токена *relop* показана на рис. 3.12. Обратите внимание, что рис. 3.11 представляет собой часть этой более сложной диаграммы переходов. □

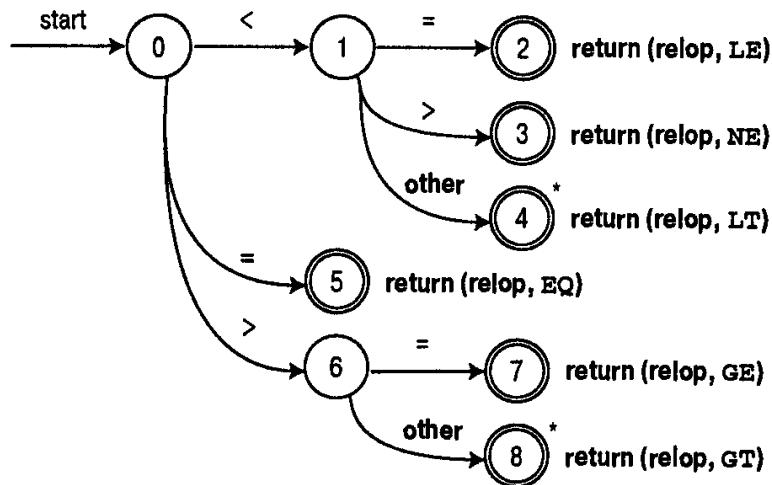


Рис. 3.12. Диаграмма переходов для операторов отношения

Пример 3.8

Поскольку ключевые слова есть не что иное, как последовательности букв, они представляют собой исключение из правила, гласящего, что последовательность букв и цифр, начинающаяся с буквы, является идентификатором. Но вместо кодирования исключений в диаграмме переходов можно использовать небольшую хитрость и рассматривать ключевые слова как специальные идентификаторы, как и в разделе 2.7. Тогда по достижении заключительного состояния на рис. 3.13 выполняется некоторый код, который определяет, чем является лексема, приведшая нас в это состояние, — ключевым словом или идентификатором.

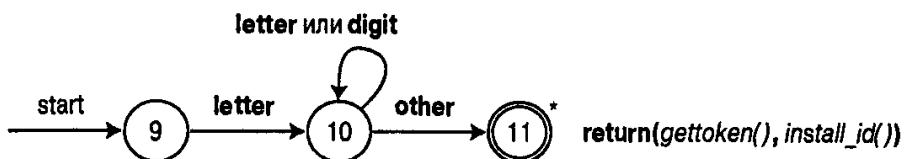


Рис. 3.13. Диаграмма переходов для идентификаторов и ключевых слов

Простейшая технология отделения ключевых слов от идентификаторов состоит в соответствующей инициализации таблицы символов, в которой хранится информация об идентификаторах. Для токенов на рис. 3.10 надо внести в таблицу символов строки `if`, `then` и `else` до начала работы с входным потоком. При распознавании такой строки возвращается токен ключевого слова. Инструкция возврата, связанная с допускающим состоянием на рис. 3.13, использует функции `gettoken()` и `install_id()` для получения токена и атрибута-значения соответственно. Процедура `install_id()` имеет доступ к буферу, где находится лексема идентификатора. Программа проверяет таблицу символов и, если лексема найдена в ней и помечена как ключевое слово, `install_id()` возвращает 0. Если лексема найдена и является переменной программы, `install_id()` возвращает указатель на запись в таблице символов. Если лексема в таблице символов отсутствует, она считается переменной, и функция возвращает указатель на вновь созданную запись в таблице символов для этой лексемы.

Процедура `gettoken()` точно так же просматривает таблицу символов в поисках лексемы. Если лексема представляет собой ключевое слово, возвращается соответствующий токен; в противном случае возвращается токен `id`.

Заметим, что диаграмма переходов не изменяется при необходимости распознавания дополнительного ключевого слова — мы просто добавляем новое ключевое слово в таблицу символов при инициализации. □

Технология размещения ключевых слов в таблице символов играет важную роль при создании лексического анализатора вручную. Без этого в лексическом анализаторе для типичного языка программирования количество состояний равно нескольким сотням, в то время как при использовании описанного способа обычно хватает менее сотни состояний.

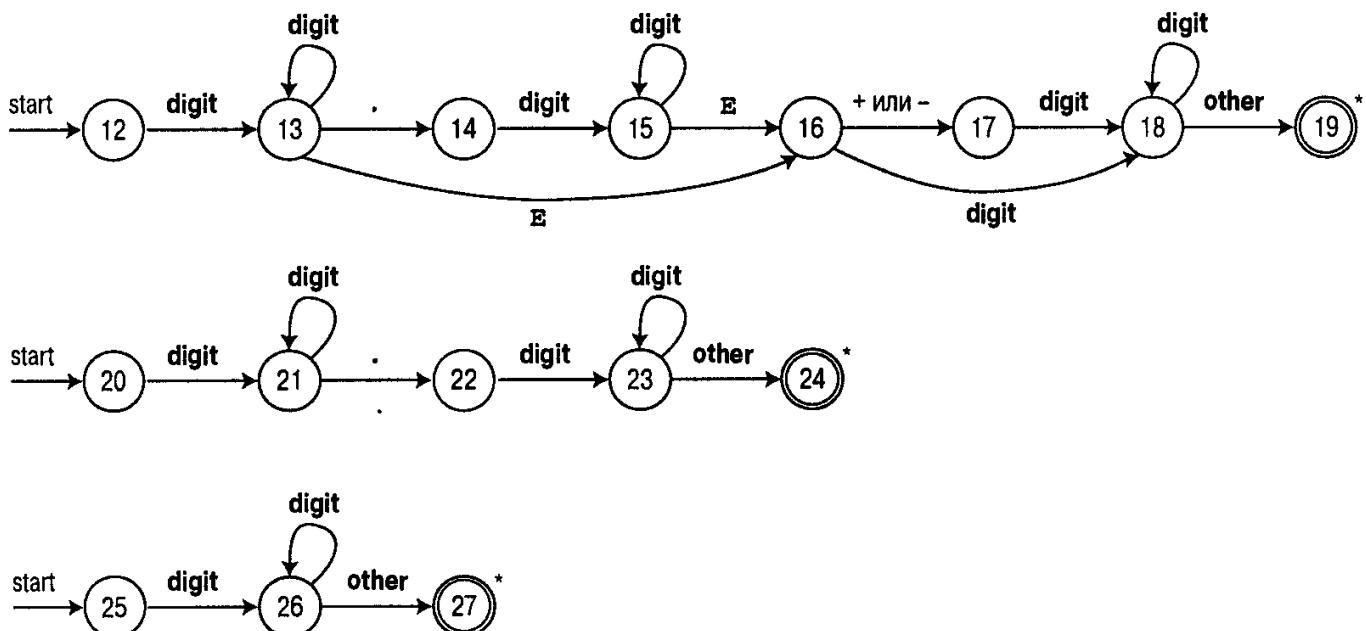


Рис. 3.14. Диаграммы переходов для беззнаковых чисел в Pascal

Пример 3.9

Число проблем возрастает при построении распознавателя беззнаковых чисел, задаваемых регулярным определением

$\text{num} \rightarrow \text{digit}^+ (. \text{ digit}^+)? (\text{E} (+ | -)?) \text{ digit}^+$

Обратите внимание, что определение имеет вид **digits fraction? exponent?**, где **fraction** и **exponent** — необязательные части.

Лексема для данного токена должна быть максимально возможной длины⁵. Например, лексический анализатор не должен останавливаться после того, как встретит 12 или даже 12.3, если введено число 12.3E4. Начиная с состояний 25, 20 и 12 на рис. 3.14, допускающие состояния при получении во входном потоке строки 12.3E4, за которой следует нецифровой символ, достигаются после считывания 12, 12.3 и 12.3E4 соответственно. Диаграммы переходов со стартовыми состояниями 25, 20 и 12 описывают соответственно **digits**, **digits fraction?** и **digits fraction? exponent**, так что испытывать их следует в обратном порядке — 12, 20, 25.

По достижении любого из допускающих состояний 19, 24 и 27 происходит вызов процедуры `install_num()`, которая вносит лексему в таблицу чисел и возвращает

⁵ Вообще говоря, правило поиска лексемы наибольшей длины — общепринятое правило работы лексического анализатора, позволяющее избежать множества неприятностей. — Прим. ред.

указатель на созданную запись. Лексический анализатор возвращает токен `num` с этим указателем в качестве лексического значения. □

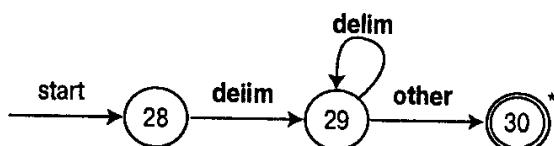
Информация о языке, содержащаяся вне регулярных определений токенов, может использоваться для точного указания ошибок во входном тексте. Например, при вводе `1.<x` произойдет в состояниях 14 и 22 на рис. 3.14 при получении очередного входного символа — «<». Вместо того, чтобы вернуть число 1, можно сообщить об ошибке и продолжить работу так, как если бы была введена строка `1. 0<x`. Таким образом, знание особенностей языка обеспечивает возможность восстановления при некоторых ошибках, которые иначе привели бы к аварийному завершению трансляции.

Имеется ряд способов избежать излишних проверок в диаграммах переходов на рис. 3.14. Один из подходов состоит в том, чтобы переписать диаграммы переходов, объединив их в одну (что в общем-то является нетривиальной задачей). Другой путь — изменить реакцию на ошибки в процессе работы с диаграммой. Подход, рассматриваемый позднее в этой главе, позволяет пройти по ряду допускающих состояний, возвращаясь назад к последнему пройденному допускающему состоянию перед сбоем.

Пример 3.10

Последовательность диаграмм переходов для всех токенов из примера 3.6 осуществляется при объединении диаграмм переходов, изображенных на рис. 3.12–3.14. Стартовые состояния с меньшими номерами испытываются до состояний с большими номерами.

В данном случае остается только проблема, связанная с пробелами. Обработка регулярного определения `ws`, представляющего пробелы, отличается от обработки шаблонов тем, что синтаксическому анализатору не возвращается какой-либо токен. Диаграмма переходов, распознающая пробелы, выглядит следующим образом.



При этом по достижении допускающего состояния синтаксическому анализатору не возвращается никакое значение; мы просто возвращаемся к стартовому состоянию первой диаграммы переходов для поиска другого шаблона.

По возможности следует начинать просмотр с часто встречающихся токенов, поскольку работа с некоторой диаграммой переходов начинается только после завершения работы с ее предшественниками. Так как пробелы встречаются весьма часто, размещение диаграммы переходов для пробелов перед другими должно повысить общую эффективность. □

Реализация диаграммы переходов

Последовательность диаграмм переходов может быть преобразована в программу поиска токенов, определяемых диаграммами. В данном случае используется систематический подход, который работает для всех диаграмм переходов и создает программы, размер которых пропорционален количеству состояний и дуг в диаграммах.

Каждое состояние дает часть кода. Если из состояния выходят дуги, то его код считывает очередной символ и выбирает дугу следования (если это возможно). Для чтения очередного символа из входного буфера используется функция `nextchar()`, которая

также перемещает указатель *forward* и возвращает считанный символ⁶. Если существует дуга, помеченная считанным символом (или классом, к которому принадлежит считанный символ), то управление передается коду состояния, на которое указывает данная дуга. Если такой дуги нет, а текущее состояние указывает, что токен не найден, вызывается функция *fail()*, которая возвращает указатель *forward* к началу лексемы и инициирует новый поиск токена с использованием следующей диаграммы переходов. Если все диаграммы опробованы, *fail()* вызывает программу обработки ошибок.

Для возвращаемых токенов используем глобальную переменную *lexical_value*, которой присваивается указатель, возвращаемый функциями *install_id()* и *install_num()*. Токен возвращается основной функцией лексического анализатора, *nexttoken()*.

Для поиска стартового состояния очередной диаграммы переходов используется инструкция *case*. В реализации на языке программирования С, приведенной на рис. 3.15, две переменные — *state* и *start* — отслеживают текущее и стартовое состояния испытываемой диаграммы переходов. Номера состояний в коде соответствуют диаграммам переходов на рис. 3.12–3.14.

```
int state = 0, start = 0;
int lexical_value; /* Второй компонент токена */

int fail()
{
    forward = token_beginning;
    switch ( start ) {
        case 0: start = 9; break;
        case 9: start = 12; break;
        case 12: start = 20; break;
        case 20: start = 25; break;
        case 25: recover(); break;
        default: /* Ошибка компиляции */
    }
    return start;
}
```

Рис. 3.15. Код С для поиска следующего стартового состояния

Дуги в диаграммах переходов проходятся путем повторяющегося выбора фрагмента кода для состояния и его выполнением для определения следующего состояния, как показано на рис. 3.16. В листинге приведен код для состояния 0, обрабатывающий пробелы, и код для двух диаграмм переходов с рис. 3.13 и 3.14. Обратите внимание, что конструкция *while(1) stmt* выполняет *stmt* “вечно”, т.е. до оператора *return*.

```
token nexttoken()
{
    while(1) {
        switch ( state ) {
            case 0: c = nextchar();
                      /* c - текущий сканируемый символ */

```

⁶ Более эффективная реализация может использовать вместо функции *nextchar()* встроенный макрос.

```

        if (c == blank || c == tab || c == newline) {
            state = 0;
            lexeme_beginning++;
            /* перемещаем указатель начала лексемы */
        }
        else if (c == '<') state = 1;
        else if (c == '=') state = 5;
        else if (c == '>') state = 6;
        else state = fail();
        break;

    ... /* cases 1-8 */

    case 9: c = nextchar();
        if (isletter(c)) state = 10;
        else state = fail();
        break;
    case 10: c = nextchar();
        if (isletter(c)) state = 10;
        else if (isdigit(c)) state = 10;
        else state = 11;
        break;
    case 11: retract(1); install_id();
        return ( gettoken() );
    ... /* cases 12-24 */

    case 25: c = nextchar();
        if (isdigit(c)) state = 26;
        else state = fail();
        break;
    case 26: c = nextchar();
        if (isdigit(c)) state = 26;
        else state = 27;
        break;
    case 27: retract(1); install_num();
        return ( NUM );
    }

}
}

```

Рис. 3.16. Код лексического анализатора на языке С

Поскольку С не позволяет вернуть одновременно и токен, и атрибут-значение, функции `install_id()` и `install_num()` устанавливают должным образом значение некоторой глобальной переменной, присваивая ей атрибут-значение, соответствующее записи в таблице для `id` или `num`.

Если язык реализации не имеет инструкции `case`, можно создать массив для каждого состояния, проиндексированный символами. Если `state1` — такой массив, то `state1[c]` — указатель на часть кода, которая должна выполняться, если текущий сканируемый символ — `c` (обычно этот код завершается переходом к коду для следующего состояния). Массив для состояния `s` можно рассматривать как таблицу переходов для `s`.

3.5. Язык спецификации лексических анализаторов

Для построения лексических анализаторов на основе спецификаций, использующих регулярные выражения, был создан ряд специальных программных инструментов. Мы уже ознакомились с использованием регулярных выражений для определения шаблонов токенов. Перед тем как рассмотреть алгоритмы компиляции регулярных выражений в программах распознавания соответствия шаблонам, познакомимся с примером инструмента, который может использовать такой алгоритм.

В этом разделе опишем инструмент под названием Lex, широко используемый для определения лексических анализаторов различных языков. В данном случае будем говорить о компиляторе Lex, который использует специальный язык Lex. Рассмотрение существующего инструмента позволит показать, каким образом можно объединить определение шаблонов с помощью регулярных выражений с действиями (например, созданием записей в таблице символов), которые должен выполнять лексический анализатор. Lex-образные спецификации могут использоваться даже при отсутствии компилятора Lex — их можно преобразовать в работающую программу вручную, воспользовавшись технологией диаграмм переходов из предыдущего раздела.

Обычно Lex используется так, как изображено на рис. 3.17. Вначале подготавливается спецификация лексического анализатора с использованием языка Lex (файл lex.1). Затем lex.1 пропускается через компилятор Lex для получения программы на языке C lex.yu.c. Эта программа содержит табличное представление диаграммы переходов, построенной по регулярным выражениям из файла lex.1, а также стандартную программу, которая использует эту таблицу для распознавания лексем. Действия, связанные с регулярными выражениями в файле lex.1, представляют собой фрагменты кода на языке C, которые переносятся в lex.yu.c. После этого программа lex.yu.c компилируется с помощью компилятора С для создания программы a.out, которая является лексическим анализатором, преобразовывающим входной поток в последовательность токенов.

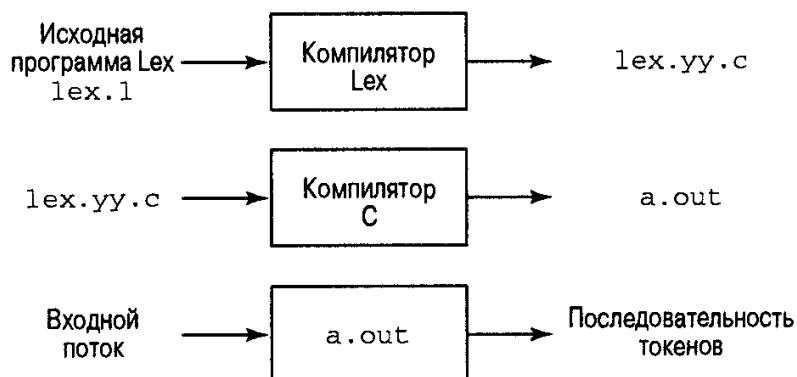


Рис. 3.17. Создание лексического анализатора с помощью Lex

Спецификации Lex

Программа Lex состоит из трех разделов.

Объявления

%%

Правила трансляции

%%

Вспомогательные процедуры

Раздел объявлений включает объявления переменных, именованные константы (идентификаторы, используемые для представления констант) и регулярные определения. Регулярные определения представляют собой инструкции, подобные приведенным в разделе 3.3, и используются в качестве компонентов регулярных выражений в правилах трансляции.

Правила трансляции Lex-программы являются инструкциями типа

```
p1 { action1 }
p2 { action2 }
...
pn { actionn }
```

где каждое p_i — регулярное выражение, а $action_i$ — фрагмент кода, описывающий выполняемые лексическим анализатором действия, если лексема соответствует шаблону p_i . В Lex действия записываются с использованием языка программирования С; однако, вообще говоря, они могут быть выражены на любом языке⁷.

Третий раздел содержит различные вспомогательные процедуры, которые могут быть скомпилированы отдельно и загружены вместе с лексическим анализатором.

Лексический анализатор, созданный Lex, работает в связке с синтаксическим анализатором следующим образом. При активизации синтаксическим анализатором лексический анализатор начинает чтение оставшейся части входного потока по одному символу до тех пор, пока не будет найден самый длинный префикс, соответствующий одному из регулярных выражений p_i . Затем выполняется действие $action_i$. Обычно это действие возвращает управление синтаксическому анализатору. Однако если это не так, лексический анализатор продолжает поиск лексем, пока управление не будет передано синтаксическому анализатору. Такой поиск лексем до явного выхода из лексического анализатора позволяет легко осуществить обработку пробелов и комментариев.

Лексический анализатор возвращает синтаксическому анализатору единственную величину — найденный токен. Для передачи значения атрибута с информацией о лексеме используется глобальная переменная `yyval`.

Пример 3.11

На рис. 3.18 приведена программа, которая распознает токены с рис. 3.10 и возвращает найденный токен. Подробное рассмотрение кода поможет понять важные особенности Lex.

```
% {
    /* Определения именованных констант
     * LT, LE, EQ, NE, GT, GE, IF, THEN,
     * ELSE, ID, NUMBER, RELOP
     */
}

/* Регулярные определения */

delim      [ \t\n]
ws         {delim}+
letter     [A-Za-z]
digit      [0-9]
```

⁷ Существуют реализации Lex для различных языков программирования, например Pascal. — Прим. ред.

```

id      {letter}({letter}|{digit})*
number  {digit}+(\.{digit}+)?(E[+\-]?{digit}+)??

%%
{ws}    { /* Ни действий, ни возврата */ }
if     { return (IF); }
then   { return (THEN); }
else   { return (ELSE); }
{id}    { yyval = install_id(); return (ID); }
{number} { yyval = install_num(); return (NUMBER); }
"<"    { yyval = LT; return (RELOP); }
"<="   { yyval = LE; return (RELOP); }
"="    { yyval = EQ; return (RELOP); }
"<>"  { yyval = NE; return (RELOP); }
">"    { yyval = GT; return (RELOP); }
">="   { yyval = GE; return (RELOP); }

%%
install_id() {
/*
 * Процедура вставки лексемы, на первый символ которой
 * указывает yytext и длина которой равна yylen, в таблицу
 * символов. Возвращает указатель на запись.
 */
}

install_num() {
/*
 * Аналогичная процедура для лексемы,
 * представляющей число.
*/
}

```

Рис. 3.18. Программа Lex для токенов с рис. 3.10

В разделе объявлений мы видим место для объявления некоторых констант, используемых правилами трансляции⁸. Эти объявления взяты в специальные скобки — % { и % }. Все, что находится в этих скобках, непосредственно копируется в лексический анализатор `lex.yu.c` и не рассматривается как часть регулярных определений или правил трансляции. Точно так же обрабатываются и вспомогательные процедуры из третьего раздела. На рис. 3.18 представлены две процедуры — `install_id` и `install_num`, которые используются правилами трансляции; эти процедуры будут дословно скопированы в `lex.yu.c`.

В разделе объявлений имеются также некоторые регулярные определения. Каждое такое определение состоит из имени и регулярного выражения, обозначаемого этим именем. Например, первое определенное имя — `delim`, которое означает класс симво-

⁸ Обычно программа `lex.yu.c` используется как подпрограмма синтаксического анализатора, генерируемого Yacc, генератором синтаксических анализаторов, с которым мы познакомимся в главе 4, “Синтаксический анализ”. В этом случае объявления констант могут предоставляться синтаксическим анализатором в результате его компиляции с программой `lex.yu.c`.

лов [\t\n], т.е. пробел, символ табуляции или новой строки. Второе определение дает имя ws последовательностям из одного или нескольких символов класса delim. Заметим, что слово delim в Lex должно находиться в фигурных скобках для отличия его от шаблона, состоящего из пяти символов delim.

В определении letter используется класс символов. Сокращение [A-Za-z] означает любую из строчных или прописных букв от A до Z. Пятое определение, id, для группирования использует скобки (представляющие собой метасимволы Lex). Вертикальная черта в Lex — метасимвол, представляющий объединение.

В последнем регулярном определении, pumber, в качестве метасимвола используется вопросительный знак ?, означающий “нуль или один экземпляр”. Кроме того, здесь применяется обратная косая черта \ как специальный символ, изменяющий значение следующего за ним escape-символа. В частности, десятичная точка в определении pumber выражена как \., поскольку сама по себе точка в Lex (как и во многих программах UNIX, работающих с регулярными выражениями) представляет собой класс символов, состоящий из любых символов, кроме символа новой строки. В классе [+\\-] используется обратная косая черта, чтобы показать, что это знак минус, а не указатель диапазона, как, например, в классе [A-Z]⁹.

Имеется и иной способ представления символов в качестве знаков (а не метасимволов Lex) — применение кавычек. Этот метод использован в разделе правил трансляции при описании шести операторов сравнения¹⁰. Теперь рассмотрим правила трансляции в разделе, следующем за первым разделителем %. Первое правило гласит, что если обнаружено ws, т.е. последовательность пробелов, символов табуляций и начала новой строки максимальной длины, не нужно выполнять никаких действий, в частности, не надо ничего возвращать синтаксическому анализатору. Вспомните — структура лексического анализатора такова, что он продолжает распознавание токенов, пока не встретит явный оператор возврата из функции.

Второе правило заставляет, встретив строку if, вернуть значение IF, представляющее собой константу, равную некоторому целому числу (это говорит синтаксическому анализатору о том, что найден токен if). Следующие два правила точно так же обрабатывают строки then и else.

В правиле для id используются две инструкции. Вначале переменной yyval присваивается значение, возвращаемое функцией install_id() (определение которой содержится в третьем разделе программы). yyval — переменная, определение которой появится в выходном файле Lex lex.yy.c и которая доступна синтаксическому анализатору. Цель yyval — содержать возвращаемое лексическое значение, поскольку вторая инструкция, return(ID), может вернуть только код класса токена.

Мы не приводим детальный код функции install_id(), а только подразумеваем, что она просматривает таблицу символов в поисках лексемы, соответствующей шаблону id. Lex делает лексемы доступными для подпрограмм из третьего раздела с помощью двух переменных — yytext и yylen. Первая соответствует переменной, о которой ранее говорили как об указателе lexeme_beginning, т.е. указателе на первый символ лексемы. Вторая, yylen, является целым числом, указывающим длину лексемы. Напри-

⁹ В действительности Lex корректно отработает определение [+ -], поскольку в нем знак минус находится в конце и не может представлять диапазон.

¹⁰ Мы поступаем таким образом с < и >, поскольку они являются метасимволами Lex. В случае знака равенства кавычки не нужны, однако ничто не запрещает их использование и в этом случае.

мер, если функция `install_id` не может найти лексему в таблице символов, она создает для нее новую запись. При этом в массив символов, о котором говорилось в разделе 2.7, можно скопировать `yytext` символов, начиная с `yytext` и завершая символом конца строки. Новая запись в таблице символов будет указывать на начало полученной таким образом копии.

Числа обрабатываются аналогично, в соответствии со следующим правилом в списке. В последних шести правилах (для операторов отношения) для возврата кода найденного оператора используется `yyval`, в то время как возвращаемое функцией лексического анализатора значение представляет собой код токена `relop` для всех случаев.

Предположим, что лексический анализатор, сгенерированный из программы на рис. 3.18, получает входной поток, состоящий из двух символов табуляции, букв `if` и пробела. Два символа табуляции — наибольший префикс, соответствующий шаблону, а именно шаблону `ws`. Действие для `ws` не делает ничего, поэтому лексический анализатор перемещает указатель начала лексемы `yytext` так, что он указывает на символ `i`, и приступает к поиску очередного токена.

Следующая соответствующая шаблону лексема — `if`. Заметим, что лексеме соответствует два шаблона — `if` и `{ id }`, и никакой из шаблонов не соответствует более длинной строке. Поскольку в листинге на рис. 3.18 шаблон для ключевого слова `if` предшествует шаблону для идентификатора, конфликт разрешается в пользу ключевого слова. В целом такая стратегия разрешения неоднозначностей упрощает резервирование ключевых слов путем расположения их до шаблона идентификаторов.

В качестве другого примера рассмотрим оператор `<=`, состоящий из двух считанных из входного потока символов. Хотя шаблон `<` и соответствует первому считанному символу, длина соответствия меньше, чем у шаблона `<=`, а потому выбор, естественно, падает на последний токен. □

Прогностический оператор

Как видно из раздела 3.1, лексические анализаторы для некоторых конструкций языков программирования должны “заглянуть вперед” (за лексему), перед тем как определить токен. Вспомним пример из Fortran с парой инструкций

```
DO 5 I = 1.25
DO 5 I = 1,25
```

В Fortran пробелы не имеют значения вне комментариев и строк Холлерита. Предположим, мы удалили все пробелы перед началом лексического анализа. Тогда лексический анализатор получит следующий входной поток

```
DO5I=1.25
DO5I=1,25
```

В первой инструкции до появления десятичной точки невозможно определить, является ли `DO` частью идентификатора `DO5I`. Во второй строке `DO` представляет собой ключевое слово.

В Lex можно записать шаблон в виде r_1/r_2 (где r_1 и r_2 — регулярные выражения). Это означает, что шаблон соответствует строке r_1 только в том случае, если за ней следует строка, соответствующая r_2 . Регулярное выражение r_2 после прогностического оператора / указывает правый контекст соответствия, который используется только для огра-

ничения и не является частью шаблона. Например, спецификация Lex, распознающая ключевое слово DO в контексте, приведенном выше, имеет вид

```
DO/({letter}|{digit})*=( {letter}|{digit})*,
```

Используя эту спецификацию, лексический анализатор приступает к просмотру входного буфера (после DO) в поисках последовательности букв и цифр, за которой следует знак “=” и еще одна такая последовательность с запятой в конце. Если за DO действительно следует данная последовательность, то только символы D и O, предшествующие оператору /, будут частью лексемы, соответствующей спецификации. При этом утилита `yytext` указывает на D, а `yylen` равно 2. Заметим, что такое определение позволяет распознать DO, даже если за ним следует мусор типа “Z4=6Q,”, но никогда не распознает DO в качестве составной части идентификатора.

Пример 3.12

Прогностический оператор может также использоваться для распознавания ключевых слов и идентификаторов в Fortran. Например, строка

```
IF(I,J) = 3
```

представляет собой обычный оператор присвоения Fortran, а не условный оператор IF. Один из способов определения ключевого слова IF в Lex состоит в указании его правого контекста с помощью прогностического оператора. Простая форма логического оператора IF выглядит как

```
IF ( condition ) statement
```

В Fortran 77 вводится другой вид логического оператора IF:

```
IF ( condition ) THEN
    then_block
ELSE
    else_block
END IF
```

Заметим, что каждая непомеченная инструкция Fortran начинается с буквы и за каждой правой скобкой, используемой для индексации или группирования, следует оператор типа =, +, запятая, другая правая скобка или конец инструкции (но не буква). В этой ситуации, чтобы считать IF ключевым словом, а не именем массива, следует сканировать входной поток в поисках правой скобки, за которой следует буква, до появления символа новой строки. Шаблон для ключевого слова IF может быть записан как

```
IF / \(.*\) {letter}
```

Точка означает любой символ, кроме символа новой строки, а обратная косая черта перед скобками говорит Lex о том, что эти скобки следует рассматривать как обычные символы, а не метасимволы для группирования в регулярных выражениях (см. упр. 3.10). □

Другой способ решения проблемы состоит в том, чтобы при появлении выражения IF (выяснить, не был ли ранее объявлен массив IF. Рассмотренный выше шаблон тестируется, только если был объявлен массив IF; иначе IF трактуется как ключевое слово. Это усложняет автоматическую реализацию лексического анализатора по спецификации Lex и может стоить дополнительного времени работы в связи с частыми дополнительными проверками. Зачастую оказывается проще написать специализированный лексический анализатор для Fortran, чем использовать генератор типа Lex.

3.6. Конечные автоматы

Распознавателем (recognizer) языка называется программа, которая получает на вход строку x и отвечает “да”, если x — предложение языка, или в противном случае “нет”. Мы компилируем регулярное выражение в распознаватель путем построения обобщенной диаграммы переходов, называемой конечным автоматом. Такой автомат может быть детерминированным или недетерминированным (недетерминированный автомат может иметь более одного перехода из состояния при одном и том же входном символе).

Как детерминированные, так и недетерминированные конечные автоматы способны к распознаванию точных регулярных множеств. Таким образом, они могут распознавать все, что могут обозначать регулярные выражения. Однако детерминированные конечные автоматы, которые приводят к более быстрому распознаванию, обычно больше по размеру, чем эквивалентные недетерминированные. В следующем разделе будут рассмотрены методы преобразования регулярных выражений в оба типа конечных автоматов. Преобразование в недетерминированный автомат более очевидно, поэтому вначале обсудим именно этот тип автоматов.

В приведенных ниже примерах используется в первую очередь язык, заданный регулярным выражением $(a \mid b)^*abb$, состоящим из множества всех строк из a и b , которые заканчиваются на abb . Подобные языки встречаются на практике. Например, регулярное выражение для имен всех файлов, заканчивающихся на $.o$, имеет вид $(. \mid o \mid c)^* . o$, где c — символ, отличающийся от точки и o . Другой пример — комментарий в языке С, начинающийся с $/*$. Этот комментарий представляет собой любую последовательность символов, заканчивающуюся на $*/$, с дополнительным требованием, чтобы ни один собственный префикс не заканчивался на $*/$.

Недетерминированные конечные автоматы

Недетерминированный конечный автомат (nondeterministic finite automaton, NFA¹¹) представляет собой математическую модель, состоящую из

- множества состояний S ;
- множества входных символов Σ (символов входного алфавита);
- функции переходов *move*, которая отображает пары символ-состояние на множество состояний;
- состояния s_0 , известного как *стартовое* (начальное);
- множества состояний F , известных как *допускающие* (конечные);

НКА может использоваться в виде помеченного ориентированного графа, так называемого *графа переходов*, узлы которого представляют собой состояния, а помеченные дуги составляют функцию переходов. Такой график похож на диаграмму переходов, однако один и тот же символ может помечать два и более переходов из одного состояния, а некоторые переходы могут быть помечены специальным символом ϵ , как обычным входным символом (ϵ -переходы).

Граф переходов для НКА, распознающего язык $(a \mid b)^*abb$, показан на рис. 3.19. Множество состояний НКА — $\{0, 1, 2, 3\}$, а входной алфавит — $\{a, b\}$. Состояние 0 — стартовое, а заключительное состояние 3 представлено двойным кружком.

¹¹ Далее в тексте используется сокращение НКА. — Прим. ред.

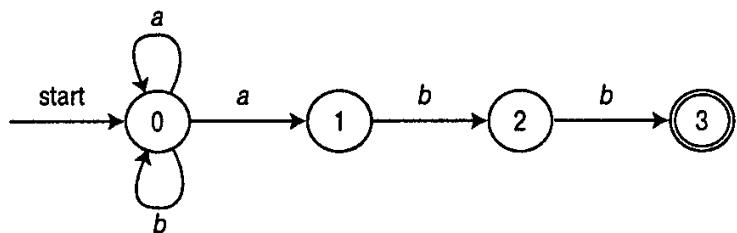


Рис. 3.19. Недетерминированный конечный автомат

При описании НКА воспользуемся графом переходов. Как мы увидим далее, функция переходов НКА может быть реализована различными способами. Простейший из них — таблица переходов, в которой строки представляют состояния, а столбцы — входные символы (и, при необходимости, ϵ). Запись в строке i для символа a является множеством состояний (или — на практике — чаще всего указателем на множество состояний), которые могут быть достигнуты переходом из состояния i при входном символе a . Таблица переходов для НКА показана на рис. 3.20.

СОСТОЯНИЕ	ВХОДНОЙ СИМВОЛ	
	a	b
0	{0, 1}	{0}
1	—	{2}
2	—	{3}

Рис. 3.20. Таблица переходов для конечного автомата на рис. 3.19

Представление автомата таблицей переходов хорошо тем, что обеспечивает быстрый доступ к переходам из данного состояния по данному символу. Вместе с тем таблица занимает слишком много места, когда входной алфавит велик и большинство переходов ведут в пустое множество состояний. Представление функции переходов в виде списка более компактно, правда, в данном случае замедляется доступ к переходам. Очевидно, что каждое из этих представлений легко преобразовать в другое.

НКА допускает, или принимает (accepts) входную строку x (а эта строка является допустимой) тогда и только тогда, когда в графе переходов существует некоторый путь от начального состояния к какому-либо из заключительных, такой, что метки дуг этого пути соответствуют строке x . НКА на рис. 3.19 допускает входные строки abb , $aabb$, $babb$, $aaabb$, Например, $aabb$ допускается по пути из 0 вдоль дуги a в состояние 0, затем в состояния 1, 2 и 3 вдоль дуг, помеченных соответственно a , b и b .

Путь может быть представлен в виде последовательности переходов состояний, так называемых *перемещений*, или *ходов* (moves). Следующая диаграмма показывает перемещения, выполненные для входной строки $aabb$:

$$0 \xrightarrow{a} 0 \xrightarrow{a} 1 \xrightarrow{b} 2 \xrightarrow{b} 3$$

Вообще говоря, в заключительное состояние может приводить более чем одна последовательность перемещений. Заметьте, для входной строки $aabb$ могут быть осуществлены и некоторые другие последовательности перемещений, не приводящие в заключительное состояние. Например, для той же входной строки $aabb$ может быть выполнена следующая последовательность перемещений, оставляющая нас в состоянии 0.

$$0 \xrightarrow{a} 0 \xrightarrow{a} 0 \xrightarrow{b} 0 \xrightarrow{b} 0$$

Язык, определяемый НКА, представляет собой множество допускаемых им входных строк. Не так сложно показать, что НКА на рис. 3.19 допускает строки $(a|b)^*abb$.

Пример 3.13

На рис. 3.21 показан НКА, распознающий строки $aa^*|bb^*$. Стока aaa допускается перемещениями по состояниям 0, 1, 2, 2 и 2. Метками соответствующих дуг являются ϵ , a и a , конкатенация которых дает строку aaa . Заметьте, что ϵ в результирующей строке отсутствует. \square

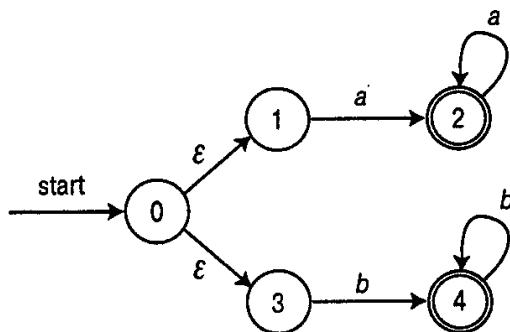


Рис. 3.21. Недетерминированный конечный автомат, допускающий строки $aa^*|bb^*$

Детерминированный конечный автомат

Детерминированный конечный автомат (deterministic finite automaton, DFA¹²) является специальным случаем недетерминированного конечного автомата, в котором

- отсутствуют состояния, имеющие ϵ -переходы;
- для каждого состояния s и входного символа a существует не более одной дуги, исходящей из s и помеченной как a .

Детерминированный конечный автомат имеет для любого входного символа не более одного перехода из каждого состояния. Если для представления функции переходов ДКА используется таблица, то каждая запись в ней представляет собой единственное состояние. Следовательно, очень просто проверить, допускает ли данный ДКА некоторую строку, поскольку имеется не более одного пути от стартового состояния, помеченного этой строкой. Следующий алгоритм имитирует поведение ДКА при обработке входной строки.

Алгоритм 3.1. Моделирование ДКА

Вход. Входная строка x , завершаемая символом конца файла eof, и ДКА D со стартовым состоянием s_0 и множеством заключительных состояний F .

Выход. “Да”, если D допускает x , и “нет” в противном случае.

Метод. Ко входной строке x применяется алгоритм, приведенный на рис. 3.22. Функция $move(s, c)$ дает состояние, в которое происходит переход из состояния s при входном символе c . Функция $nextchar$ возвращает очередной символ строки x . \square

```

s := s0;
c := nextchar;
  
```

¹² Далее в тексте используется сокращение ДКА. — Прим. ред.

```

while c ≠ eof do begin
    s := move(s, c);
    c := nextchar;
end;
if s ∈ F then
    return "да"
else return "нет"

```

Рис. 3.22. Имитационное моделирование ДКА

Пример 3.14

На рис. 3.23 показан граф переходов детерминированного конечного автомата, допускающего тот же язык $(a \mid b)^*abb$, что и НКА на рис. 3.19. При работе с этим ДКА и входной строкой *ababb* алгоритм 3.1 пройдет последовательность состояний 0, 1, 2, 1, 2, 3 и ответит “да”. \square

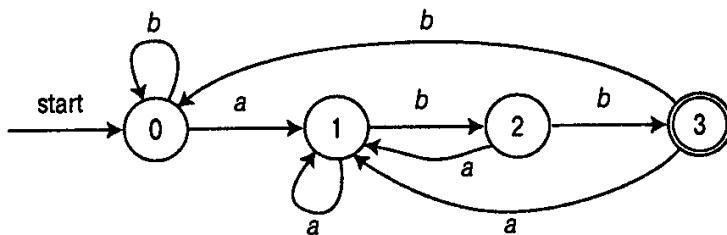


Рис. 3.23. ДКА, допускающий строку $(a \mid b)^*abb$

Преобразование НКА в ДКА

Обратите внимание, что НКА на рис. 3.19 имеет два перехода из состояния 0 для входного символа *a* — в состояние 0 или 1. Точно так же НКА на рис. 3.21 имеет два перехода для ϵ из состояния 0. Хотя это и не показано на примере, ситуация, когда можно выбрать переход для ϵ или реально введенного символа, также неоднозначна. Ситуации, в которых функция переходов многозначна, делают моделирование НКА с помощью компьютерной программы весьма сложной задачей. Определение допустимости утверждает только, что должен существовать некоторый путь, помеченный входной строкой и ведущий от начального к заключительному состоянию. Однако когда имеется много путей для одной и той же входной строки, возможно, придется рассматривать их все, чтобы найти путь к заключительному состоянию или выяснить, что такого пути не существует.

Сейчас мы рассмотрим алгоритм для преобразования НКА в ДКА, распознающий тот же язык, что и НКА. Этот алгоритм, часто называемый *построением подмножества* (subset construction), полезен при моделировании НКА компьютерной программой. Тесно связанный с ним алгоритм играет важную роль в построении LR-анализаторов, рассматриваемых в следующей главе.

В таблице переходов НКА каждая запись представляет собой множество состояний; в таблице переходов ДКА — единственное состояние. Общая идея преобразования НКА в ДКА состоит в том, что каждое состояние ДКА соответствует множеству состояний НКА. ДКА использует свои состояния для отслеживания всех возможных состояний, в которых НКА может находиться после чтения очередного входного символа. Таким образом, после чтения входного потока $a_1a_2\dots a_n$ ДКА находится в состоянии, которое пред-

ставляет подмножество T состояний НКА, достижимых из стартового состояния НКА по пути, помеченному как $a_1a_2\dots a_n$. Количество состояний ДКА может оказаться экспоненциально зависящим от количества состояний НКА, но на практике этот наихудший случай встречается крайне редко.

Алгоритм 3.2. Построение ДКА из НКА (построение подмножества)

Вход. НКА N .

Выход. ДКА D , допускающий тот же язык.

Метод. Данный алгоритм строит таблицу переходов $Dtran$ для D . Каждое состояние ДКА является множеством состояний НКА, и мы строим $Dtran$ так, чтобы D “параллельно” моделировал все возможные перемещения N по данной входной строке.

Для отслеживания множеств состояний НКА используем операции, приведенные на рис. 3.24 (s представляет состояние НКА, а T — множество состояний НКА).

ОПЕРАЦИЯ	ОПИСАНИЕ
$\epsilon\text{-closure}(s)$ (ϵ -замыкание(s))	Множество состояний НКА, достижимых из состояния s только по ϵ -переходам
$\epsilon\text{-closure}(T)$ (ϵ -замыкание(T))	Множество состояний НКА, достижимых из какого-либо состояния s из T только по ϵ -переходам
$move(T, a)$	Множество состояний НКА, в которые имеется переход из какого-либо состояния s из T по входному символу a

Рис. 3.24. Операции над состояниями НКА

Перед тем как рассматривать первый входной символ, N может быть в любом из состояний множества $\epsilon\text{-closure}(s_0)$, где s_0 — стартовое состояние N . Предположим, что состояния множества T , и только они, достижимы из s_0 после прочтения данной последовательности входных символов, и пусть a — следующий входной символ. Получив a , N может переместиться в любое состояние из множества $move(T, a)$. Совершив из этих состояний все возможные ϵ -переходы, N после обработки a может быть в любом состоянии из $\epsilon\text{-closure}(move(T, a))$.

Изначально $\epsilon\text{-closure}(s_0)$ является единственным состоянием в $Dstates$ и непомечено;
while в $Dstates$ имеется непомеченное состояние T **do begin**
 пометить T ;
 for каждый входной символ a **do begin**
 $U := \epsilon\text{-closure}(move(T, a));$
 if $U \notin Dstates$ **then**
 Добавить U как непомеченное состояние в $Dstates$;
 $Dtran[T, a] := U$
 end
end

Рис. 3.25. Построение подмножества

Множество состояний $Dstates$ автомата D и таблицу его переходов $Dtran$ можно создать следующим образом. Каждое состояние D соответствует множеству состояний

НКА, в которых может находиться N после чтения некоторой последовательности входных символов, включая все возможные ϵ -переходы до и после считанных символов. Стартовое состояние D — ϵ -closure(s_0). Состояния и переходы добавляются в D согласно алгоритму, приведенному на рис. 3.25. Состояние D является допускающим, если оно представляет собой множество состояний НКА, содержащих как минимум одно допускающее состояние N .

```

Внести все состояния множества  $T$  в стек  $stack$ ;
инициализировать  $\epsilon$ -closure( $T$ ) множеством  $T$ ;
while  $stack$  не пуст do begin
    снять со стека верхний элемент  $t$ 
    for каждое состояние  $u$  с дугой
        от  $t$  к  $u$ , помеченной  $\epsilon$  do
        if  $u \notin \epsilon$ -closure( $T$ ) then begin
            добавить  $u$  к  $\epsilon$ -closure( $T$ );
            поместить  $u$  в  $stack$ 
        end
    end
end
```

Рис. 3.26. Вычисление ϵ -замыкания

Вычисление ϵ -closure(T) является типичным процессом поиска графа для узлов, достижимых из данного множества узлов. В этом случае состояния T представляют данное множество узлов, а граф состоит только из дуг НКА, помеченных ϵ . Простой алгоритм вычисления ϵ -closure(T) использует стек для хранения состояний, дуги которых не были проверены на наличие ϵ -переходов. Такая процедура показана на рис. 3.26. \square

Пример 3.15

На рис. 3.27 показан еще один НКА N , допускающий язык $(a \mid b)^*abb$ (он встретится нам в следующем разделе, где будет построен по регулярному выражению). Применим к N алгоритм 3.2. Стартовое состояние эквивалентного ДКА представляет собой ϵ -closure(0), т.е. $A = \{0, 1, 2, 4, 7\}$, поскольку именно эти состояния достижимы из состояния 0 путями, в которых каждая дуга помечена ϵ . Заметим, что путь может и не иметь дуги, так что состояние 0 также достигается из состояния 0 , а значит, входит в исходное множество.

Алфавит входных символов представляет собой $\{a, b\}$. С помощью алгоритма на рис. 3.25 можно пометить A и вычислить ϵ -closure(move(A, a)). Вычислим move(A, a), множество состояний N , имеющих переходы по символу a для элементов множества A . Из состояний $0, 1, 2, 4$ и 7 только 2 и 7 имеют такие переходы к состояниям 3 и 8 , поэтому

$$\epsilon\text{-closure}(\text{move}(\{0,1,2,4,7\}, a)) = \epsilon\text{-closure}(\{3,8\}) = \{1,2,3,4,6,7,8\}$$

Назовем это множество B . Таким образом, $Dtran[A, a] = B$.

Среди состояний в A только состояние 4 имеет переход по b в состояние 5 , так что ДКА имеет переход по b из A в состояние

$$C = \epsilon\text{-closure}(\{5\}) = \{1,2,4,5,6,7\}$$

Таким образом, $Dtran[A, b] = C$.

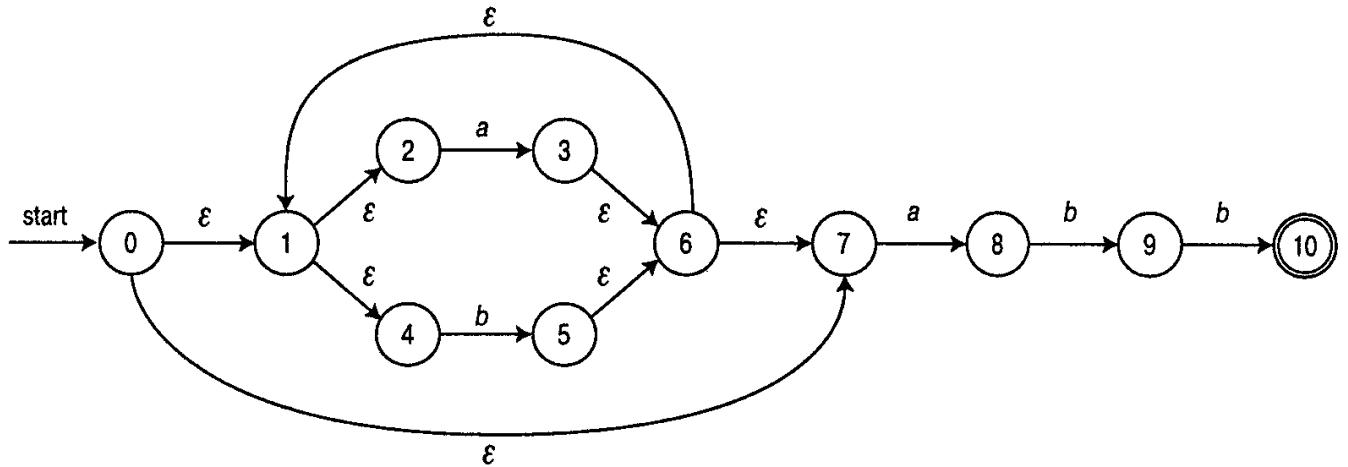


Рис. 3.27. НКА N для $(a|b)^*abb$

Если продолжить этот процесс с непомеченными в настоящий момент множествами B и C , в конечном итоге все множества-состояния ДКА окажутся помеченными. Это бесспорно, поскольку всего имеется 2^{11} различных подмножеств множества из одиннадцати состояний, а однажды помеченное множество остается таковым навсегда. В действительности мы создаем пять различных множеств состояний:

$$\begin{aligned} A &= \{0, 1, 2, 4, 7\} \\ B &= \{1, 2, 3, 4, 6, 7, 8\} \\ C &= \{1, 2, 4, 5, 6, 7\} \end{aligned}$$

$$\begin{aligned} D &= \{1, 2, 4, 5, 6, 7, 9\} \\ E &= \{1, 2, 4, 5, 6, 7, 10\} \end{aligned}$$

Состояние A является начальным, а E — единственным заключительным состоянием. Полностью таблица переходов $Dtran$ показана на рис. 3.28.

Состояние	Входной символ	
	a	b
A	B	C
B	B	D
C	B	C
D	B	E
E	B	C

Рис. 3.28. Таблица переходов $Dtran$ для ДКА

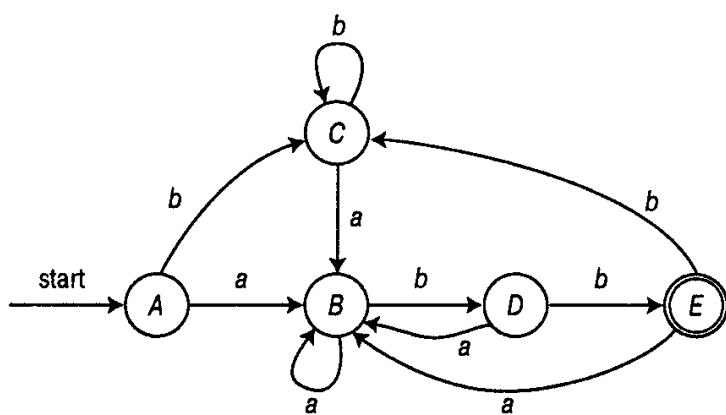


Рис. 3.29. Результат применения построения подмножества к рис. 3.27

Граф переходов полученного в результате ДКА показан на рис. 3.29. Следует заметить, что ДКА на рис. 3.23 также допускает язык $(a|b)^*abb$ и имеет на одно состояния меньше. Вопрос минимизации количества состояний ДКА обсудим в разделе 3.9. \square

3.7. От регулярного выражения к НКА

Имеется немало стратегий построения распознавателей на основе регулярных выражений; каждая из них имеет свои достоинства и недостатки. Одна из стратегий, используемая во многих текстовых редакторах, состоит в построении НКА на основе регулярного выражения и последующем моделировании его поведения с применением алгоритмов 3.3 и 3.4. Если существенна скорость работы, НКА можно преобразовать в ДКА с использованием алгоритма построения подмножества из предыдущего раздела. В разделе 3.9 рассмотрим альтернативные реализации ДКА по регулярным выражениям, в которых НКА явным образом не строится. Этот раздел завершается обсуждением вопросов скорости работы и размера памяти при реализации распознавателей на основе НКА и ДКА.

Построение НКА по регулярному выражению

Сейчас мы рассмотрим алгоритм построения НКА на основе регулярного выражения. Имеется множество вариаций этого алгоритма, и здесь будет представлен простейший вариант, легкий в реализации. Алгоритм синтаксически управляем, поскольку использует в процессе построения синтаксическую структуру регулярного выражения. Ветвления алгоритма соответствуют ветвлению определения регулярного выражения. Вначале покажем, как строится автомат для распознавания ϵ и любого символа из алфавита; затем определим, как конструируется автомат для выражений, содержащих операторы объединения, конкатенации или замыкания Клини. Например, для выражения $r|s$ индуктивно построим НКА из автоматов для r и s .

В процессе создания НКА каждый шаг вносит не более двух новых состояний. В результате НКА, построенный по регулярному выражению, имеет количество состояний, превышающее число символов и операторов в регулярном выражении не более чем в два раза.

Алгоритм 3.3. Построение НКА на основе регулярного выражения (построение Томпсона)

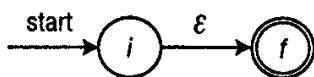
Вход. Регулярное выражение r над алфавитом Σ .

Выход. НКА N , допускающий $L(r)$.

Метод. Вначале мы разбираем r на составляющие подвыражения. Затем, используя приведенные ниже правила (1) и (2), строим НКА для каждого базового символа в r (которые представляют собой символы алфавита либо ϵ). Базовые символы соответствуют частям (1) и (2) в определении регулярного выражения. Важно понимать, что если в регулярном выражении r некоторый символ a встречается несколько раз, для каждого его вхождения создается отдельный НКА.

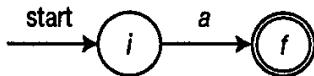
Затем, следуя синтаксической структуре регулярного выражения r , индуктивно комбинируем эти НКА с использованием правила (3), пока не будет получен НКА для всего выражения. Каждый промежуточный НКА соответствует своему подвыражению r и имеет несколько важных свойств: такой автомат имеет ровно одно заключительное состояние, ни одна дуга не входит в начальное состояние и ни одна не выходит из заключительного.

1. Для ϵ строим НКА.



Здесь i — новое начальное состояние, а f — новое заключительное. Очевидно, что этот НКА распознает $\{\epsilon\}$.

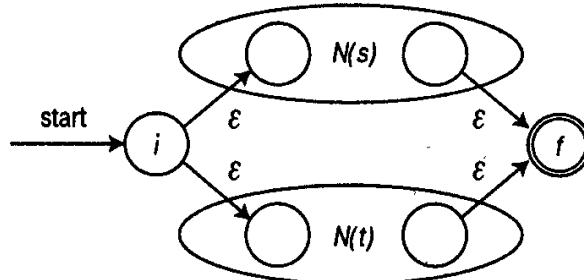
2. Для $a \in \Sigma$ строим НКА



Здесь также i — новое начальное состояние, а f — новое заключительное. Этот НКА распознает $\{a\}$.

3. Пусть $N(s)$ и $N(t)$ представляют собой НКА для регулярных выражений s и t .

a) Для регулярного выражения $s | r$ строим следующий составной НКА $N(s | r)$:



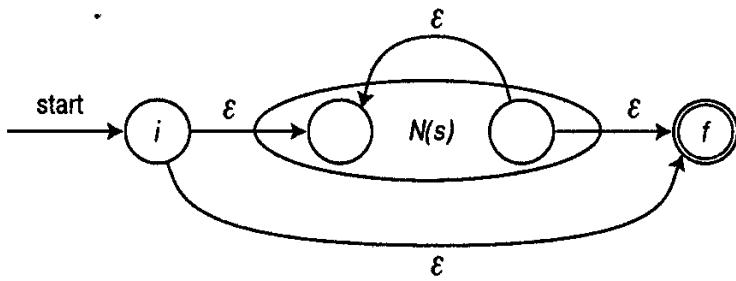
Здесь i — новое начальное состояние, а f — новое заключительное. Имеются переходы по ϵ от i к стартовым состояниям $N(s)$ и $N(t)$, а также переходы по ϵ из заключительных состояний $N(s)$ и $N(t)$ в новое заключительное состояние f . Начальные и заключительные состояния $N(s)$ и $N(t)$ не совпадают с начальными и заключительными состояниями $N(s | t)$. Заметим, что любой путь от i к f должен проходить либо только через $N(s)$, либо только через $N(t)$. В соответствии со сказанным составной НКА распознает $L(s) \cup L(t)$.

b) Для регулярного выражения st строится составной НКА $N(st)$:



Начальное состояние $N(s)$ становится начальным состоянием составного НКА, а заключительное состояние $N(t)$ — заключительным состоянием составного автомата. Заключительное состояние $N(s)$ сливаются с начальным $N(t)$; таким образом, все переходы из начального состояния $N(t)$ преобразуются в переходы из заключительного состояния $N(s)$. Новое объединенное состояние теряет свой статус начального/заключительного в комбинированном автомате. Путь от i к f должен пройти вначале через $N(s)$, а затем — через $N(t)$, так что метки на этом пути представляют собой строку из $L(s)L(t)$. Поскольку нет дуг, которые входят в начальное состояние $N(t)$ или покидают заключительное $N(s)$, не существует пути от i к f , который бы возвращался от $N(t)$ к $N(s)$. Следовательно, составной НКА распознает $L(s)L(t)$.

c) Для регулярного выражения s^* построим составной НКА $N(s^*)$:



Здесь i — новое начальное состояние, а f — новое заключительное. В составном НКА можем пройти от i к f либо напрямую, вдоль дуги ϵ , которая представляет тот факт, что $\epsilon \in (L(s))^*$, либо через $N(s)$ один или более раз. Очевидно, что составной НКА распознает $(L(s))^*$.

d) Для регулярного выражения в скобках (s) в качестве НКА используем $N(s)$.

Каждое новое состояние получает индивидуальное имя. Таким образом, никакие состояния любого из компонентов НКА не могут иметь одно и то же имя. Даже если в r несколько раз встречается один и тот же символ, для каждого экземпляра этого символа создается отдельный НКА с собственными состояниями. \square

Мы можем убедиться, что каждый шаг построения в алгоритме 3.3 дает НКА, который распознает корректный язык. Кроме того, построение приводит к НКА $N(r)$ со следующими свойствами.

1. $N(r)$ имеет количество состояний, превышающее количество символов и операторов в r не более чем в два раза. Это вытекает из того, что на каждом этапе построения создается не более двух новых состояний.
2. $N(r)$ имеет ровно одно начальное и одно заключительное состояния. Заключительное состояние не имеет исходящих переходов. Это свойство верно и для всех составляющих автоматов.
3. Каждое состояние $N(r)$ имеет либо один исходящий переход для символа из Σ , либо не более двух исходящих ϵ -переходов.

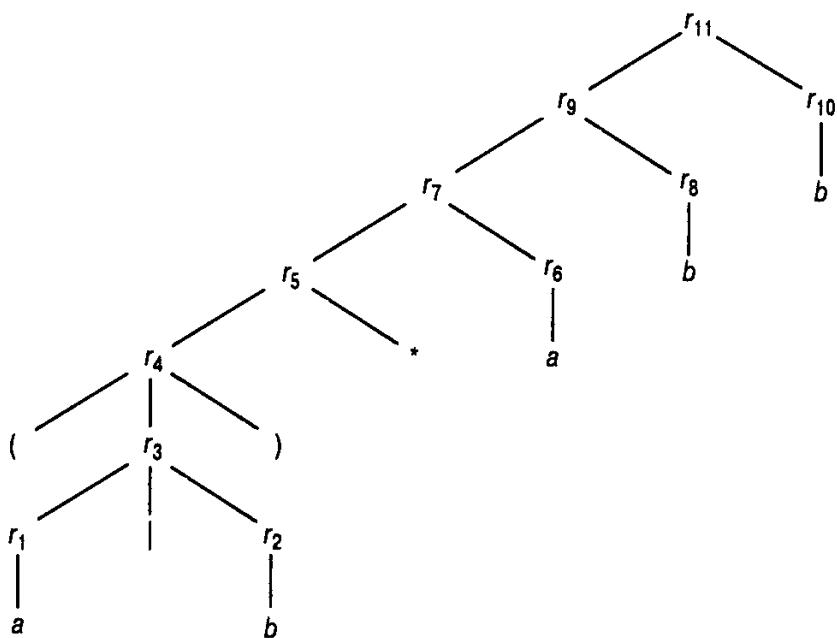
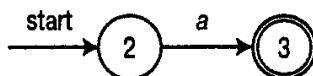


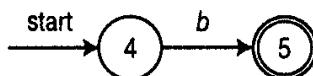
Рис. 3.30. Разбор $(a|b)^*abb$

Пример 3.16

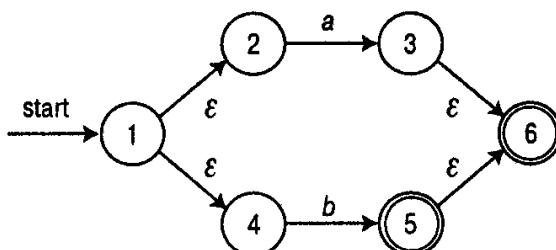
Применим алгоритм 3.3 к построению $N(r)$ для регулярного выражения $r = (a \mid b)^*abb$. На рис. 3.30 показано дерево разбора для r , аналогичное деревьям, созданным для арифметических выражений в разделе 2.2. Для составляющей r_1 , первого a , строим НКА



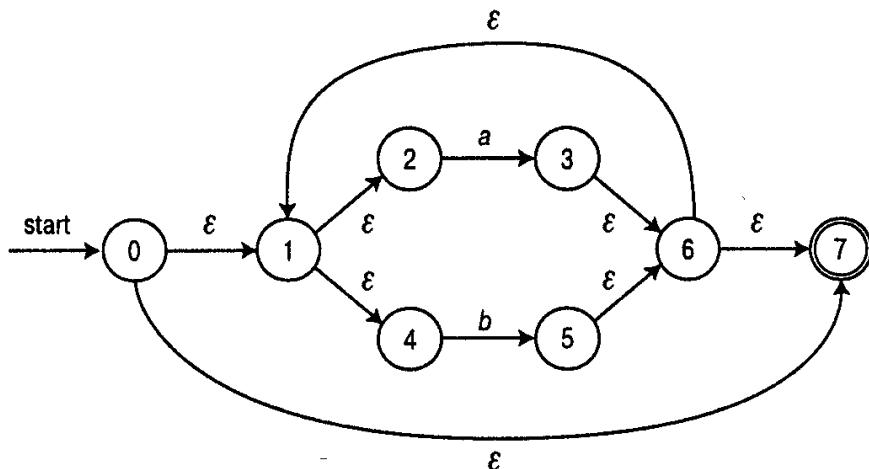
Для r_2 НКА выглядит как



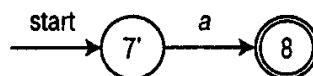
Теперь можно соединить $N(r_1)$ и $N(r_2)$ с использованием правила объединения, получив при этом НКА для $r_3 = r_1 \mid r_2$



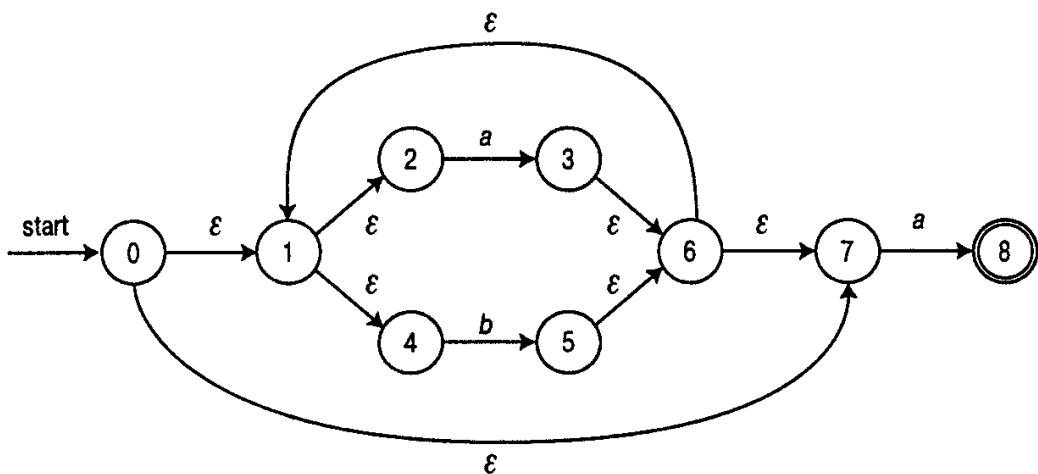
НКА для (r_3) тот же, что и для r_3 , а для $(r_3)^*$ он выглядит следующим образом:



НКА для $r_6 = a$ представляет собой



Чтобы получить автомат для r_5r_6 , нужно объединить состояния 7 и $7'$ в единое состояние 7 . В результате получаем НКА



Продолжая работу, получим НКА для $r_{11} = (a|b)^*abb$, который показан на рис. 3.27. \square

Двустековое моделирование НКА

Теперь мы рассмотрим алгоритм, который по НКА N , построенному по алгоритму 3.3, и входной строке x определяет, допускается ли x автоматом N . Алгоритм считывает строку посимвольно и вычисляет полное множество состояний, в которых N может находиться после прочтения каждого входного префикса. Для эффективного построения множества недетерминированных состояний алгоритм использует свойства НКА, получаемые при построении последнего по алгоритму 3.3. Алгоритм может быть выполнен за время, пропорциональное $|N| \times |x|$, где $|N|$ — количество состояний N , а $|x|$ — длина строки x .

Алгоритм 3.4. Моделирование НКА

Вход. НКА N , построенный по алгоритму 3.3, и входная строка x . Мы считаем, что строка x заканчивается символом конца файла `eof`. Стартовое состояние НКА — s_0 , а множество заключительных состояний — F .

Выход. Ответ “да”, если N допускает x , и “нет” в противном случае.

Метод. Применим алгоритм, схематически изображенный на рис. 3.31, ко входной строке x . Алгоритм, по сути, выполняет построение подмножества. Он вычисляет переход из текущего множества состояний S в следующее множество состояний в два этапа. Вначале определяется $\text{move}(S, a)$, все состояния, которые могут быть достигнуты из состояния S переходом по a (текущему входному символу). Затем вычисляется ϵ -замыкание этого множества, т.е. все состояния, которые могут быть достигнуты из $\text{move}(S, a)$ несколькими ϵ -переходами (включая нуль ϵ -переходов). Для считывания очередного символа из входной строки используется функция nextchar . Когда считаны все символы, алгоритм проверяет наличие заключительного состояния в множестве текущих состояний S и отвечает “да”, если оно там есть, и “нет” в противном случае. \square

```

S := ε-closure({s₀});
a := nextchar;
while a ≠ eof do begin
    S := ε-closure(move(S, a));
    a := nextchar

```

```

end;
if  $S \cap F \neq \emptyset$  then
    return "да"
else return "нет"

```

Рис. 3.31. Моделирование НКА, построенного по алгоритму 3.3

Алгоритм 3.4 может быть эффективно реализован с использованием двух стеков и битового вектора, индексированного состояниями НКА. Один стек используется для отслеживания текущего множества недетерминированных состояний, а второй — для вычисления множества состояний после перехода. Для вычисления ϵ -closure применяется алгоритм, приведенный на рис. 3.26. Битовый вектор может использоваться для определения, имеется ли некоторое недетерминированное состояние в стеке (чтобы не внести его в стек дважды), за время $O(1)$. Как только будут определены и внесены во второй стек новые состояния, можно просто поменять стеки местами (исходным стеком становится второй, а внесение новых состояний производится в первый стек). Поскольку каждое недетерминированное состояние имеет не более двух выходящих переходов, оно может привести не более чем к двум новым состояниям. Обозначим через $|N|$ количество состояний N . Поскольку в стеке может находиться не более $|N|$ состояний, вычисление следующего множества состояний из текущего может быть выполнено за время, пропорциональное $|N|$. Следовательно, время, необходимое для моделирования поведения N при входной строке x , пропорционально $|N| \times |x|$.

Пример 3.17

Пусть N — НКА, представленный на рис. 3.27, а x — строка, состоящая из одного символа a . Начальное состояние представляет собой ϵ -closure($\{0\}$) = {0, 1, 2, 4, 7}. Для входного символа a имеются переходы от 2 к 3 и от 7 к 8. Таким образом, $T = \{3, 8\}$. ϵ -closure(T) дает следующее состояние {1, 2, 3, 4, 6, 7, 8}. Поскольку ни одно из этих недетерминированных состояний не является заключительным, алгоритм возвращает “нет”.

Заметим, что алгоритм 3.4 выполняет построение подмножества в процессе работы. Например, сравним описанные выше переходы с состояниями ДКА на рис. 3.29, построенного из НКА на рис. 3.27. Начальное и следующее за ним для входного символа a множества состояний соответствуют состояниям A и B ДКА. \square

Скорость работы

Для данного регулярного выражения r и входной строки x существует два метода определения, входит ли x в $L(r)$. Один подход состоит в использовании алгоритма 3.3 для построения НКА N из r . Это построение может быть выполнено за время $O(|r|)$, где $|r|$ — длина r . N имеет количество состояний, не более чем в два раза превосходящее $|r|$, и не более двух переходов для каждого состояния. В результате таблица переходов N занимает объем $O(|r|)$. Затем с помощью алгоритма 3.4 за время $O(|r| \times |x|)$ можно определить, допускает ли N строку x . Таким образом, используя описанный подход, можно узнать, содержится ли x в $L(r)$, за время, пропорциональное длине r , умноженной на дли-

ну x . Такой подход применяется во многих текстовых редакторах для поиска шаблонов регулярных выражений¹³, когда строка x не очень длинная.

Второй подход заключается в создании ДКА из регулярного выражения r с применением построения Томпсона, а затем — построения подмножества (алгоритм 3.2) к полученному НКА (метод, позволяющий избежать явного построения промежуточного НКА, дан в разделе 3.9). Реализуя функцию переходов с помощью таблицы переходов, можно использовать алгоритм 3.1 для моделирования работы ДКА с входной строкой x за время, пропорциональное длине x (независимо от количества состояний ДКА). Такой подход чаще используется в программах поиска по шаблону, которые просматривают текстовые файлы в поисках шаблонов регулярных выражений. После того как ДКА построен, поиск осуществляется очень быстро, поэтому такой подход в особенности применим к очень длинным строкам¹⁴.

Однако существует ряд регулярных выражений, для которых наименьшие ДКА имеют количество состояний, представляющее собой экспоненту от размера регулярного выражения. Например, регулярное выражение $(a|b)^*a(a|b)(a|b)\dots(a|b)$, завершающееся $n-1$ $(a|b)$, не имеет ДКА с менее чем 2^n состояниями. Это регулярное выражение описывает строку из a и b , на n -м месте от правого конца которой находится символ a . Нетрудно доказать, что любой ДКА для этого выражения должен отслеживать как минимум n символов входного потока; в противном случае может быть дан неверный ответ. Ясно, что для отслеживания всех возможных последовательностей из n символов a и b требуется по крайней мере 2^n состояния. К счастью, такого рода выражения встречаются в приложениях лексического анализа не так уж часто, хотя и вполне реальны.

Третий подход предлагает использовать ДКА, но при этом избежать построения всей таблицы переходов с помощью технологии, называемой “отложенным вычислением переходов”. При таком методе переходы вычисляются в процессе работы программы, но только в тот момент, когда они действительно нужны для работы. Вычисленные таким образом переходы хранятся в кэше. Каждый раз при необходимости вычислить некоторый переход происходит обращение к кэшу. Если искомого перехода в нем нет, он вычисляется и помещается в кэш. Если кэш переполняется, можно удалить часть ранее вычисленных переходов и освободить место для нового.

На рис. 3.32 показаны память и время, необходимые в наихудшем случае при работе со строкой x и языком, определяемым регулярным выражением r с использованием распознавателей на базе недетерминированных и детерминированных конечных автоматов. “Отложенная” технология комбинирует требования к пространству для НКА и время работы с ДКА. Требования к памяти сводятся к размещению регулярного выражения и кэша, а скорость работы почти так же высока, как и при обычной работе с ДКА. В ряде приложений такая технология по скорости превосходит работу с ДКА, поскольку при этом не вычисляются переходы, которые никогда не используются.

АВТОМАТ	ПАМЯТЬ	ВРЕМЯ
НКА	$O(r)$	$O(r \times x)$
ДКА	$O(2^{ r })$	$O(x)$

Рис. 3.32. Память и время, требуемые для распознавания регулярных выражений

¹³ Точнее, для поиска подстрок строки x , задаваемых регулярным выражением r . — Прим. ред.

¹⁴ А также для многократного поиска с одним и тем же регулярным выражением, а значит, с однажды построенным ДКА. — Прим. ред.

3.8. Построение генератора лексических анализаторов

В этом разделе рассматривается создание программного инструмента, который автоматически строит лексический анализатор по программе на языке Lex (несмотря на то, что из рассмотренных нами методов ни один не соответствует в точности используемому в UNIX-программе Lex).

Предположим, что у нас есть спецификация лексического анализатора в виде

```
p1 { action1 }  
p2 { action2 }  
...  
pn { actionn }
```

где, как и в разделе 3.5, каждый шаблон p_i представляет собой регулярное выражение, а каждое действие $action_i$ — программный фрагмент, который будет выполняться при нахождении во входном потоке лексемы, соответствующей p_i .

Наша задача состоит в построении распознавателя, который ищет лексемы во входном буфере. Если находится соответствие более чем одному шаблону, то выбирается более длинная лексема. Если самой длинной лексеме соответствует несколько шаблонов, то выбирается первый подходящий из списка шаблонов.

Конечный автомат — естественная модель, на основе которой строится лексический анализатор, в нашем случае имеющий вид, показанный на рис. 3.33б. В данном случае имеется входной буфер, рассмотренный в разделе 3.2, с двумя указателями — указателем на начало лексемы и указателем просмотра. Компилятор Lex строит таблицу переходов для конечного автомата на основе шаблонов регулярных выражений в спецификации Lex. Сам по себе лексический анализатор состоит из имитатора конечного автомата, который использует полученную таблицу переходов для поиска шаблонов регулярных выражений во входном буфере.

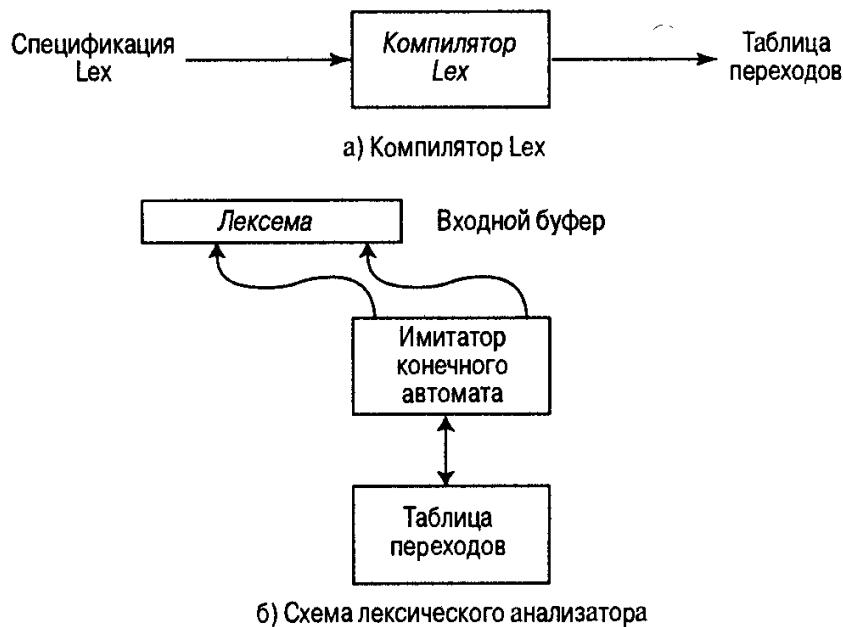


Рис. 3.33. Модель компилятора Lex

Как показано в оставшейся части этого раздела, реализация компилятора Lex может базироваться как на недетерминированных, так и детерминированных автоматах. В конце последнего раздела мы увидим, что таблица переходов НКА для шаблона регулярного

выражения может быть значительно меньше, чем таковая для случая ДКА. Однако решающим фактором при выборе того или иного метода может оказаться более быстрая работа ДКА.

Поиск по шаблону на основе НКА

Один из методов заключается в построении таблицы переходов недетерминированного конечного автомата N для составного шаблона $p_1|p_2|\dots|p_n$. Это можно сделать, вначале создав НКА $N(p_i)$ для каждого шаблона p_i , с использованием алгоритма 3.3, а затем добавив новое начальное состояние s_0 , и связав его с начальными состояниями каждого из автоматов $N(p_i)$ с помощью ϵ -перехода, как показано на рис. 3.34.

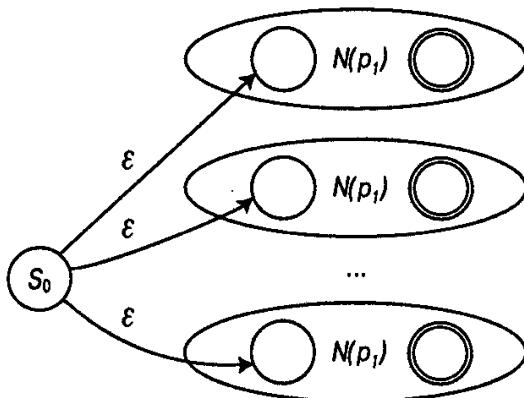


Рис. 3.34. НКА, построенный на основе спецификации Lex

Для моделирования этого НКА можно воспользоваться модификацией алгоритма 3.4. Данная модификация гарантирует, что комбинированный автомат распознает самый длинный префикс из входного потока, соответствующий шаблону. В полученном НКА для каждого шаблона p_i имеется заключительное состояние. При моделировании работы НКА с помощью алгоритма 3.4 создается последовательность множеств состояний, в которых мог находиться объединенный конечный автомат после считывания очередного символа из входного потока. Даже если есть множество состояний, содержащее заключительное состояние, для поиска наиболее длинного соответствия нужно продолжать моделирование НКА, пока не будет достигнуто *завершение* (termination), т.е. множество состояний, из которого нет переходов для данного входного символа.

Полагаем, что спецификация Lex разработана так, чтобы была невозможной ситуация, когда НКА был бы не в состоянии завершить работу при полностью заполненном буфере. Например, обычно компиляторы имеют ограничения на длину идентификаторов, и нарушения этого ограничения отслеживаются при переполнении входного буфера (если не ранее).

Для поиска корректного соответствия внесем в алгоритм 3.4 два изменения. Во-первых, когда мы добавляем заключительное состояние в текущее множество состояний, запоминаем текущую позицию во входном потоке и шаблон p_i , соответствующий этому заключительному состоянию. Если текущее множество состояний уже содержит заключительное состояние, то записывается только шаблон, появляющийся первым в спецификации Lex. Во-вторых, продолжаем выполнять переходы до тех пор, пока не достигнем завершения, после чего возвращаем указатель просмотра *forward* в позицию, где было обнаружено последнее соответствие. Шаблон этого соответствия идентифицирует найденный токен, а соответствующая лексема располагается между указателем начала лексемы и указателем просмотра.

Обычно спецификация Lex такова, что всегда обнаруживается соответствие некоторому шаблону (возможно, шаблону ошибки). Если не найдено соответствие никакому из шаблонов, возникает непредусмотренная ошибка и лексический анализатор должен передать управление программе обработки ошибок.

Пример 3.18

Изложенное выше иллюстрирует простой пример. Предположим, что имеется программа Lex, состоящая из трех регулярных выражений и не использующая регулярных определений.

```
a { } /* Действия в этом примере опущены */
abb { }
a*b+ { }
```

Три приведенных токена распознаются автоматом, изображенным на рис. 3.35а. Мы в какой-то степени упростили третий автомат по сравнению с полученным при помощи алгоритма 3.3. Как было указано выше, можно преобразовывать автоматы рис. 3.35а в один комбинированный НКА, показанный на рис. 3.35б.

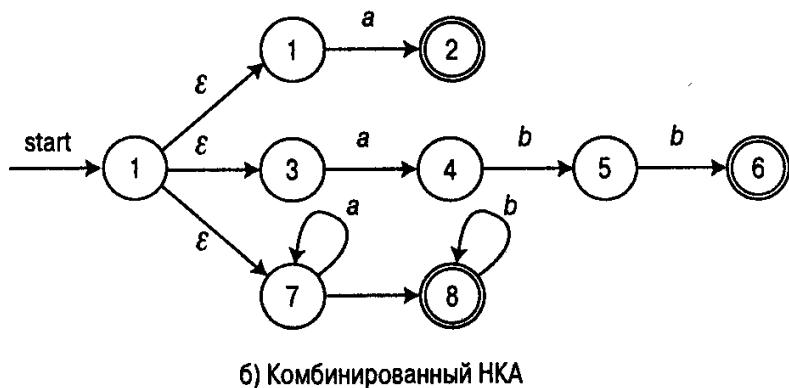
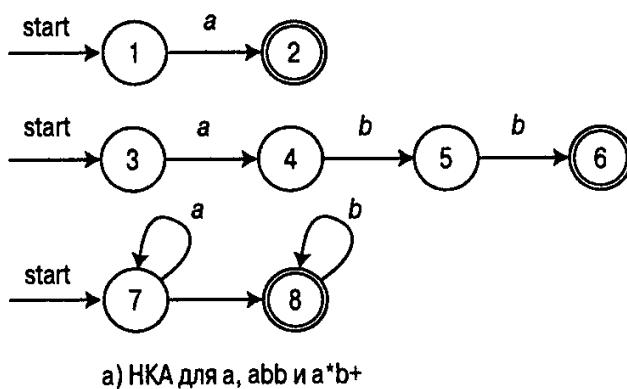


Рис. 3.35. НКА, распознающий три различных шаблона

Теперь рассмотрим поведение автомата N при работе с входной строкой $aaba$ с использованием модифицированного алгоритма 3.4. На рис. 3.36 показано множество состояний и шаблоны, соответствующие каждому входному символу при обработке строки $aaba$. Из рисунка видно, что начальное множество состояний — $\{0, 1, 3, 7\}$. Состояния 1, 3 и 7 имеют переходы по a в состояния, соответственно, 2, 4 и 7. Поскольку состояние 2 является заключительным для первого шаблона, определяем, что первый шаблон соответствует входной строке после чтения первого символа.

Однако имеется переход из состояния 7 в 7 по второму входному символу, а значит, нужно продолжить выполнение переходов. По следующему входному символу — b — выполняется переход из состояния 7 в 8. Состояние 8 является заключительным для третьего шаблона. По достижении состояния 8 по следующему входному символу a переходов нет, следовательно, мы достигли завершения. Поскольку последнее соответствие было достигнуто после прочтения третьего символа из входного потока, можно заключить, что третий шаблон соответствует лексеме abb . \square

Роль действия $action_i$, связанного с шаблоном p_i в спецификации Lex, заключается в следующем. Когда распознается экземпляр p_i , лексический анализатор выполняет связанное с ним действие $action_i$. Заметим, что действие не выполняется, когда НКА попадает в состояние, содержащее заключительное состояние для p_i — выполнение происходит только тогда, когда определяется, что p_i соответствует самая длинная лексема.

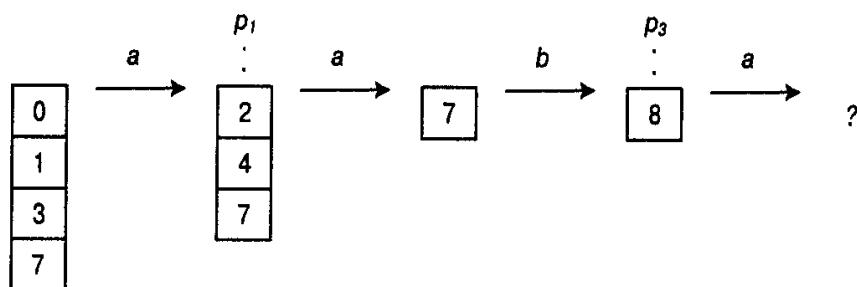


Рис. 3.36. Последовательность множеств состояний, проходимых при обработке входной строки aab

ДКА для лексического анализа

Другой подход к построению лексического анализатора на основе Lex-спецификации состоит в использовании ДКА для выполнения проверки на соответствие шаблону. Единственный нюанс — необходимо убедиться, что найдено корректное соответствие. Использование ДКА полностью аналогично только что описанному применению НКА. При конвертации НКА в ДКА с применением алгоритма 3.2 в данном подмножестве недетерминированных состояний может оказаться несколько заключительных состояний. В этой ситуации приоритет имеет заключительное состояние, которое соответствует первому шаблону в спецификации. Как и в случае моделирования НКА, модификация алгоритма состоит в выполнении переходов до тех пор, пока не будет достигнуто состояние, не имеющее следующего состояния для очередного входного символа. Для определения найденной лексемы возвращаемся к последнему входному символу, приведшему ДКА в заключительное состояние.

СОСТОЯНИЕ	ВВЕДЕНИЙ СИМВОЛ		ОБЪЯВЛЕННЫЙ ШАБЛОН
	<i>a</i>	<i>b</i>	
0137	247	8	нет
247	7	58	<i>a</i>
8	—	8	a^*b^+
7	7	8	нет
58	—	68	a^*b^+
68	—	8	<i>abb</i>

Рис. 3.37. Таблица переходов ДКА

Пример 3.19

При преобразовании НКА на рис. 3.35 в ДКА получим таблицу состояний (рис. 3.37), в которой состояния ДКА названы списками состояний НКА. Последний столбец на рис. 3.37 показывает один из шаблонов, распознанных при входе ДКА в соответствующее состояние. Например, среди состояний НКА 2, 4 и 7 только состояние 2 — заключительное, соответствующее шаблону a (см. рис. 3.35a). Таким образом, состояние ДКА 247 распознает шаблон a .

Заметим, что строка abb соответствует двум шаблонам, abb и a^*b^+ , распознаваемым в НКА состояниями 6 и 8. Состояние ДКА 68, приведенное в нижней строке таблицы переходов, включает два заключительных состояния НКА. Поскольку abb находится в спецификации Lex перед a^*b^+ , считается, что в состоянии 68 находится соответствие шаблону abb .

При входной строке $aaba$ ДКА входит в состояния, полученные при моделировании работы НКА, показанного на рис. 3.36. Рассмотрим другой пример, а именно строку aba . ДКА на рис. 3.37 начинает работу из состояния 0137. При входном символе a осуществляется переход в состояние 247, затем по символу b автомат переходит в состояние 58, из которого для символа a новый переход отсутствует. Таким образом, достигнуто завершение переходов 0137—247—58; последнее из этих состояний включает заключительное состояние 8 НКА на рис. 3.35a. Следовательно, в состоянии 58 ДКА может объявить о распознавании шаблона a^*b^+ и выбрать в качестве лексемы строку ab , приведшую в это состояние. \square

Реализация прогностического оператора

Вспомним описанный в разделе 3.4 прогностический оператор $/$, необходимый в тех случаях, когда шаблону, означающему некоторый токен, нужно описать замыкающий контекст для текущей лексемы. При преобразовании шаблона $c /$ в НКА можно рассматривать $/$ как ϵ и не искать этот символ во входном потоке. Однако если во входном буфере распознается строка, описываемая этим регулярным выражением, конец лексемы оказывается не в заключительном состоянии НКА, а в последнем состоянии перед переходом по (воображаемому) символу $/$.

Пример 3.20

НКА, распознающий шаблон для IF (пример 3.12), показан на рис. 3.38. Состояние 6 указывает на наличие ключевого слова IF ; однако токен IF определяется состоянием 2. \square

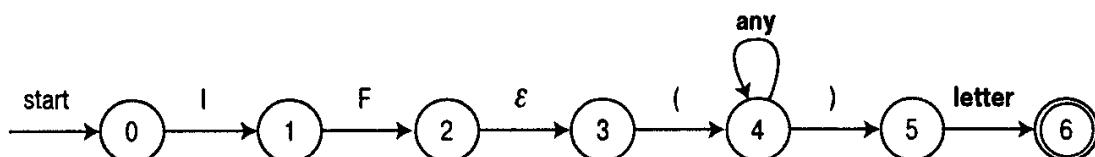


Рис. 3.38. НКА, распознающий ключевое слово IF языка программирования Fortran

3.9. Оптимизация поиска соответствия шаблону на базе ДКА

В этом разделе мы представим три алгоритма, используемые для реализации и оптимизации поиска соответствия шаблону на базе регулярных выражений. Первый алгоритм подходит для включения в компилятор Lex, поскольку он строит ДКА непосредственно по регулярному выражению, без построения промежуточного НКА.

Второй алгоритм минимизирует количество состояний любого ДКА; он может применяться для уменьшения размера программы поиска, основанной на ДКА. Алгоритм весьма эффективен, его время работы — $O(n \log n)$, где n — количество состояний ДКА. Третий алгоритм может использоваться для создания быстрого и более компактного представления таблицы переходов ДКА по сравнению с обычной двухмерной таблицей.

Важные состояния в НКА

Назовем состояние НКА *важным* (*important*), если оно имеет исходящий не- ϵ -переход. Построение подмножества (рис. 3.25) использует при определении ϵ -closure(*move*(T, a)) (множества состояний, достижимых из T при входном символе a) только важные состояния в подмножестве T . Множество *move*(s, a) является непустым только в том случае, когда состояние s — важное. В процессе построения два подмножества могут считаться идентичными, если они содержат одни и те же важные состояния и либо оба содержат, либо оба не содержат заключительные состояния НКА.

При применении алгоритма построения подмножества к НКА, полученному по регулярному выражению алгоритмом 3.3, можно воспользоваться особыми свойствами НКА для комбинирования двух построений. Комбинированное построение связывает важные состояния НКА с символами в регулярных выражениях. Построение Томпсона создает важное состояние при появлении символа алфавита в регулярном выражении. Например, важные состояния будут построены для каждого a и b в $(a|b)^*abb$.

Кроме того, полученный в результате автомат имеет ровно одно заключительное состояние, но оно не является важным, поскольку нет переходов, выходящих из него. Добавляя в качестве маркера правого конца регулярного выражения r специальный символ $\#$ ¹⁵, мы добавляем к заключительному состоянию r переход по $\#$, делая это состояние важным в НКА для $r\#$. Другими словами, используя расширенное регулярное выражение $(r)\#$, можно забыть о заключительных состояниях в процессе построения подмножества; по окончании построения любое состояние ДКА с переходом по $\#$ должно быть заключительным.

Представим расширенное регулярное выражение синтаксическим деревом с базовыми символами в листьях и операторами в промежуточных узлах. Будем говорить о внутренних узлах как о *cat*-узлах, *or*-узлах или *star*-узлах, если они помечены соответственно оператором конкатенации, | или *. На рис. 3.39a показано синтаксическое дерево для расширенного регулярного выражения (*cat*-узлы дерева помечены точками). Синтаксическое дерево для регулярного выражения может быть построено так же, как и синтаксическое дерево для арифметического выражения (см. главу 2, “Простой однопроходный компилятор”).

¹⁵ Этот символ уникален в том смысле, что не встречается во входном потоке. — Прим. ред.

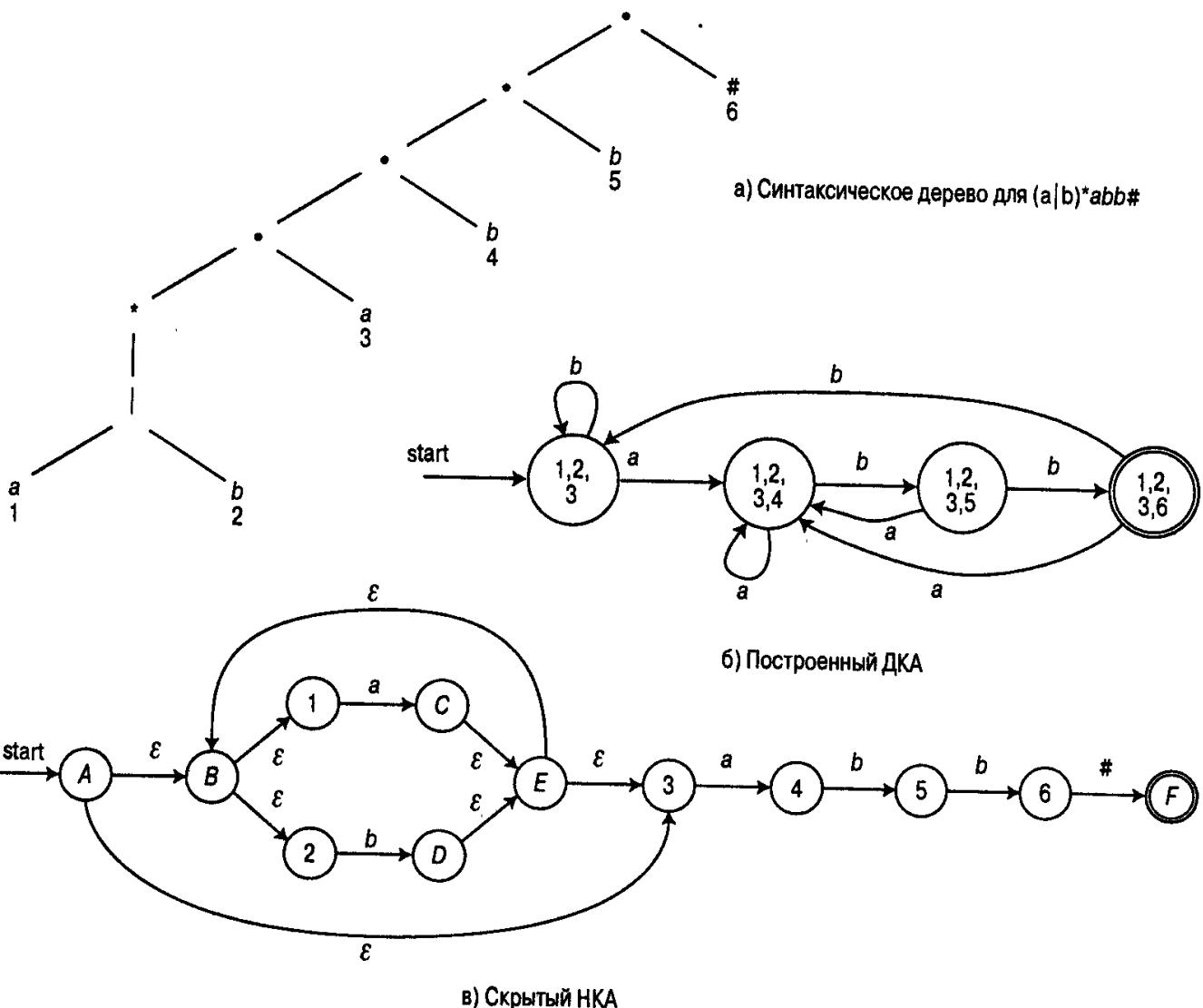


Рис. 3.39. ДКА и НКА, построенные по $(a|b)^*abb\#$

Листья синтаксического дерева для регулярного выражения помечены символами алфавита или ϵ . Каждому листу, не помеченному ϵ , приписывается целое значение, уникальное в пределах дерева, и оно используется, как *позиция* листа (а также его символа). Повторяющийся символ, таким образом, имеет несколько позиций. На рис. 3.39а позиции указаны числами, размещенными под соответствующими символами. Пронумерованные состояния НКА на рис. 3.39в соответствуют позициям листьев в синтаксическом дереве на рис. 3.39а. То, что эти состояния представляют собой важные состояния НКА, не совпадение. Не важные состояния на рис. 3.39в именуются с помощью прописных букв.

ДКА на рис. 3.39б может быть получен из НКА на рис. 3.39в, если мы воспользуемся построением подмножества и будем отождествлять подмножества, содержащие одни и те же важные состояния. В результате получается автомат с количеством состояний, на единицу меньшим, чем на рис. 3.29.

От регулярного выражения к ДКА

Здесь мы рассмотрим, каким образом можно построить ДКА непосредственно по расширенному регулярному выражению $(r)\#$. Начнем с построения синтаксического дерева T для $(r)\#$, а затем вычислим четыре функции: *nullable*, *firstpos*, *lastpos* и *followpos*. И

наконец, построим ДКА на основе *followpos*. Функции *nullable*, *firstpos* и *lastpos* определены в узлах синтаксического дерева и используются для вычисления функции *followpos*, которая определена на множестве позиций.

Вспоминая об эквивалентности важных состояний НКА и позиций листьев в синтаксическом дереве регулярного выражения, можно обойти создание НКА построением ДКА, состояния которого соответствуют множествам позиций в дереве. ϵ -переходы НКА представляют довольно сложную структуру позиций; в частности, в ней закодирована информация о том, когда одна позиция может следовать за другой. Таким образом, каждый символ входной строки может соответствовать определенным позициям. Входной символ *c* может соответствовать только позициям, содержащим *c*; однако не каждая из них обязательно соответствует некоторому *c* из входного потока.

Понятие позиции, соответствующей входному символу, определяется в терминах функции *followpos* от позиции в синтаксическом дереве. Если *i* представляет собой некоторую позицию, то *followpos(i)* является множеством положений *j*, для которых существует некоторая входная строка ...*cd...*, обладающая тем свойством, что *i* соответствует указанному в строке символу *c*, а *j* — символу *d*.

Пример 3.21

На рис. 3.39а $followpos(1) = \{1, 2, 3\}$. Доказательство этого факта заключается в следующем. Если мы встречаем *a*, соответствующее позиции 1, то это означает, что мы встретились с первым шаблоном *a | b* в замыкании $(a | b)^*$. Далее может встретиться либо символ, соответствующий шаблону *a | b*, что соответствует значениям 1 и 2 в *followpos(1)*, либо первый символ из части регулярного выражения, следующей за шаблоном, т.е. символ *a* в позиции 3. \square

Для вычисления функции *followpos* необходимо знать, какие позиции могут соответствовать первому или последнему символам строки, генерируемой данным подвыражением регулярного выражения (такая информация использовалась в примере 3.21). Если *r** представляет собой такое подвыражение, то каждая позиция, которая может быть первой в *r*, следует за каждой позицией, которая может быть последней в *r*. Аналогично, если *rs* — подвыражение, то каждая первая позиция *s* следует за каждой последней позицией *r*.

Пусть подвыражению регулярного выражения соответствует поддерево с корнем *n* — узлом в синтаксическом дереве выражения. Функция *firstpos(n)* определяется как множество позиций, которые могут соответствовать первым символам строк, задаваемых этим подвыражением. Аналогично определяем функцию *lastpos(n)*, дающую множество позиций, которые могут соответствовать последним символам в таких строках. Например, если *n* — корень всего дерева на рис. 3.39а, то *firstpos(n) = {1, 2, 3}*, а *lastpos(n) = {6}*. Алгоритм для вычисления этих функций приведен далее.

Для вычисления функций *firstpos* и *lastpos* необходимо знать, какие узлы являются корнями подвыражений, порождающих языки, содержащие пустую строку. Такие узлы будем называть ϵ -порождающими¹⁶, а функцию *nullable(n)* определим как возвращающую значение “истина”, если *n* — ϵ -порождающий узел, и “ложь” в противном случае.

Теперь можем дать правила для вычисления функций *nullable*, *firstpos*, *lastpos* и *followpos*. Для первых трех функций существует базовое правило для выражений из основных символов и три индуктивных правила, позволяющие определить значения функций

¹⁶ В оригинале — *nullable*. — Прим. перев.

при прохождении по синтаксическому дереву снизу вверх. Эти три правила соответствуют трем операторам — объединения, конкатенации и замыкания. Правила для функций *nullable* и *firstpos* приведены на рис. 3.40. Правила для функции *lastpos* те же, что и для функции *firstpos*, но с взаимозаменой c_1 и c_2 , и потому не приведены.

Первое правило для *nullable* гласит, если n — лист, помеченный ϵ , то значение $nullable(n)$ истинно. Второе правило говорит о том, что, если n является листом, помеченным символом алфавита, значение $nullable(n)$ ложно. В этом случае каждый лист соответствует единственному входному символу и, таким образом, не может порождать ϵ . Последнее правило утверждает, что если n — *star*-узел с дочерним c_1 , то $nullable(n)$ истинно, поскольку замыкание выражения порождает язык, содержащий ϵ .

Четвертое правило для *firstpos* гласит, что если n — *cat*-узел с левым наследником c_1 и правым c_2 и если значение $nullable(c_1)$ истинно, то $firstpos(n) = firstpos(c_1) \cup firstpos(c_2)$, а в противном случае $firstpos(n) = firstpos(c_1)$. Это правило утверждает, если в выражении rs подвыражение r генерирует ϵ , то первая позиция s , как и первая позиция r , может выступать в роли первой позиции rs ; в противном случае только первая позиция r может оказаться первой позицией rs .

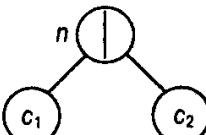
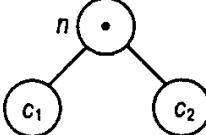
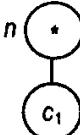
УЗЕЛ n	$nullable(n)$	$firstpos(n)$
n — узел, помеченный ϵ	true	\emptyset
n — узел, помеченный позицией i	false	$\{i\}$
	$nullable(c_1) \text{ or } nullable(c_2)$	$firstpos(c_1) \cup firstpos(c_2)$
	$nullable(c_1) \text{ and } nullable(c_2)$	$\text{if } nullable(c_1) \text{ then } firstpos(c_1) \cup firstpos(c_2) \text{ else } firstpos(c_1)$
	true	$firstpos(c_1)$

Рис. 3.40. Правила вычисления *nullable* и *firstpos*

Функция *followpos*(i) говорит, какая позиция может следовать за позицией i в синтаксическом дереве. Два приведенных ниже правила определяют все способы, которыми одна позиция может следовать за другой.

- Если n — *cat*-узел с левым потомком c_1 и правым c_2 и i является позицией в $lastpos(c_1)$, то все позиции из $firstpos(c_2)$ содержатся в *followpos*(i).
- Если n — *star*-узел и i — позиция в $lastpos(n)$, то все позиции из $firstpos(c_2)$ содержатся в *followpos*(i).

Если для каждого узла дерева вычислены *firstpos* и *lastpos*, то *followpos* для каждого положения может быть вычислена за один обход синтаксического дерева в глубину (см. главу 2, “Простой однопроходный компилятор”).

Пример 3.22

На рис. 3.41 показаны значения функций *firstpos* и *lastpos* во всех узлах синтаксического дерева на рис. 3.39 a ; значение *firstpos*(n) показано слева от узла, а *lastpos*(n) — справа. Так, значения *firstpos* и *lastpos* в крайнем левом листе дерева совпадают и равны {1} (поскольку этот узел помечен положением 1), во втором листе значения функций равны {2}. Согласно третьему правилу, *firstpos* в родительском по отношению к ним узле имеет значение {1, 2}.

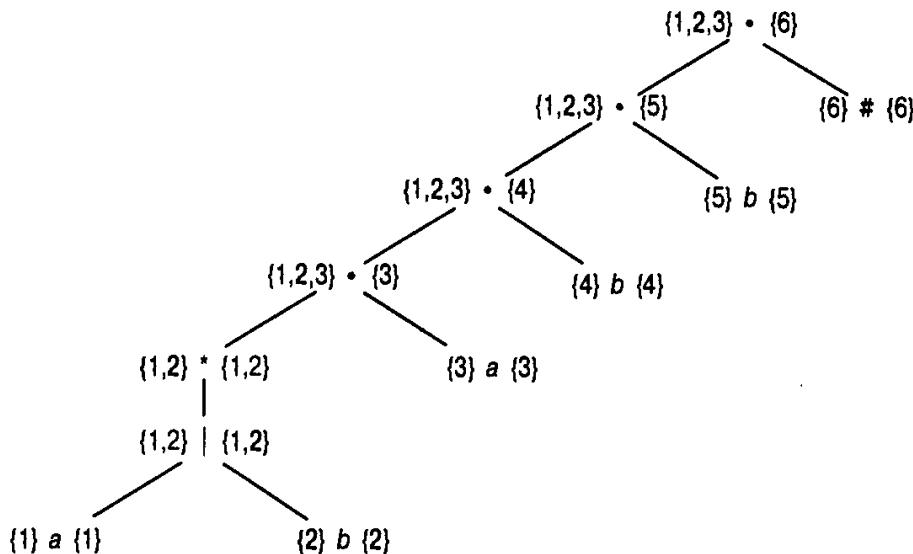


Рис. 3.41. Значения функций *firstpos* и *lastpos* в узлах синтаксического дерева для $(a|b)^*abb\#$

Узел, помеченный *, является единственным ϵ -порождающим. Таким образом, с одной стороны, согласно четвертому правилу, значение *firstpos* в родительском по отношению к нему узле (соответствующем подвыражению $(a|b)^*a$) представляет собой объединение {1, 2} и {3}, являющихся значениями *firstpos* дочерних узлов. С другой стороны, то же правило для *lastpos* утверждает, что поскольку лист в позиции 3 не ϵ -порождающий, родительский по отношению к *star*-узлу узел имеет значение *lastpos*, равное {3}.

Теперь вычислим значения функции *followpos* снизу вверх для каждого узла синтаксического дерева, представленного на рис. 3.41. В *star*-узле к *followpos*(1) и *followpos*(2) добавляем 1 и 2, в соответствии с правилом 2. В узле, родительском по отношению к *star*-узлу, согласно правилу 1 к значению *followpos*(3) добавляется 4. В следующих двух *cat*-узлах, согласно тому же правилу, к *followpos*(4) добавляется 5, а к *followpos*(5) — 6. Этим завершается построение функции *followpos*, значения которой приведены на рис. 3.42.

УЗЕЛ	<i>followpos</i>
1	{1, 2, 3}
2	{1, 2, 3}
3	{4}
4	{5}
5	{6}
6	—

Рис. 3.42. Значения функции *followpos*

Проиллюстрировать функцию *followpos* можно с помощью ориентированного графа, имеющего узел для каждой позиции и ориентированную дугу из узла i в узел j , если j содержится в $\text{followpos}(i)$. Граф для функции *followpos* приведен на рис. 3.43.

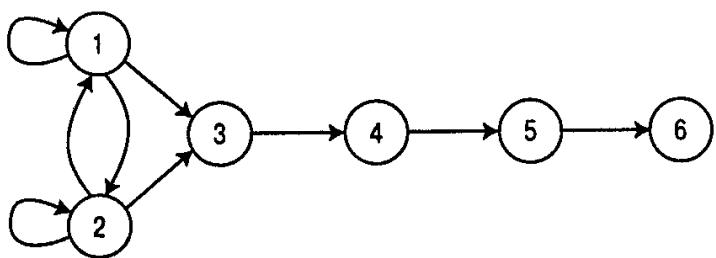


Рис. 3.43. Ориентированный граф для функции *followpos*

Интересно заметить, что эта диаграмма может стать НКА без ϵ -переходов для рассматриваемого регулярного выражения, если выполнить следующее.

1. Сделайте все позиции в *firstpos* корневого узла начальными состояниями.
2. Пометьте каждую ориентированную дугу (i, j) символом в позиции j .
3. Сделайте позицию, связанную с символом $\#$, единственным заключительным состоянием.

Таким образом, неудивительно, что мы можем преобразовать граф *followpos* в ДКА с использованием технологии построения подмножества. Полное построение может быть проведено по регулярному выражению с помощью приведенного ниже алгоритма. \square

Алгоритм 3.5. Построение ДКА по регулярному выражению r

Вход. Регулярное выражение r .

Выход. ДКА D , распознающий $L(r)$.

Метод.

1. Построить синтаксическое дерево для расширенного регулярного выражения $(r)\#$, где $\#$ — уникальный ограничитель, добавленный к (r) .
2. Создать функции *nullable*, *firstpos*, *lastpos* и *followpos* с помощью рекурсивных обходов синтаксического дерева T в глубину.
3. Построить *Dstates*, множество состояний D , и *Dtran*, таблицу переходов D , с помощью приведенной на рис. 3.44 процедуры. Состояния в *Dstates* представляют собой множество позиций; изначально каждое состояние не помечено и становится помеченным непосредственно перед рассмотрением выходящих из него переходов. Стартовое состояние D — *firstpos(root)*, а заключительные — те, в которых содержится позиция, связанная с символом $\#$. \square

Изначально единственное (непомеченное) состояние в *Dstates* — *firstpos(root)*, где $root$ — корень синтаксического дерева для $(r)\#$;

while существует непомеченное состояние $T \in Dstates$ **do begin**
 пометить T ;

for каждый входной символ a **do begin**

 Пусть U — множество позиций в *followpos(p)*
 для некоторой позиции p в T , такой, что
 символом в позиции p является a ;

```

if  $U$  не пусто и не находится в  $Dstates$  then
    добавить  $U$  как непомеченное состояние в  $Dstates$ ;
     $Dtran[T, a] := U$ 
end
end

```

Рис. 3.44. Построение ДКА

Пример 3.23

Построим ДКА для регулярного выражения $(a|b)^*abb$. Синтаксическое дерево для $((a|b)^*abb)\#$ показано на рис. 3.39а. Функция *nullable* имеет значение *true* только в узле, помеченном *. Функции *firstpos* и *lastpos* показаны на рис. 3.41, а *followpos* — на рис. 3.42.

Как видно из рис. 3.41, значение функции *firstpos* в корне равно $\{1, 2, 3\}$. Обозначим это множество через A и рассмотрим входной символ a . Символу a соответствуют позиции 1 и 3. Обозначим

$$B = \text{followpos}(1) \cup \text{followpos}(3) = \{1, 2, 3, 4\}$$

Поскольку это множество еще не было просмотрено, устанавливаем $Dtran[A, a]:=B$.

При рассмотрении входного символа b замечаем, что из позиций во множестве A с символом b связана только позиция 2, поэтому нужно рассматривать множество $\text{followpos}(2)=\{1, 2, 3\}$. Так как это множество уже рассматривалось, мы не добавляем его в $Dstates$, но добавляем переход $Dtran[A, b]:=A$.

Теперь продолжаем работу с множеством $B=\{1, 2, 3, 4\}$. Окончательные полученные состояния и переходы будут те же, что и на рис. 3.39б. \square

Минимизация количества состояний ДКА

Один важный теоретический результат состоит в том, что каждое регулярное множество распознается единственным для данного множества ДКА с минимальным числом состояний. Сейчас мы покажем, как построить такой ДКА путем уменьшения числа состояний данного ДКА до минимального без влияния на распознаваемый язык. Предположим, что есть ДКА M с множеством состояний S и алфавитом входных символов Σ ; считаем, что каждое состояние имеет переход для каждого входного символа. Если это не так, можем искусственно ввести “мертвое” состояние d с переходами из d в d для всех входных символов и добавить переход из состояния s в d для входного символа a , если s не имеет переходов для этого входного символа.

Мы говорим, что строка w различает (distinguishes) состояния s и t , если, начиная работу с ДКА M в состоянии s и получая на вход строку w , мы заканчиваем ее в заключительном состоянии, а начиная с t — в незаключительном состоянии (и наоборот). Например, ϵ отличает любое заключительное состояние от незаключительного. В ДКА на рис. 3.29 состояния A и B различаются с помощью входной строки bb , поскольку из A эта входная строка приводит в состояние C , не являющееся заключительным, в то время как из B — в заключительное состояние E .

Данный алгоритм минимизации количества состояний ДКА работает путем поиска всех групп состояний, различимых посредством некоторой входной строки. Каждая группа состояний, которые не отличаются друг от друга, сливается в одно состояние. В алгоритме используется метод работы с разбиениями множеств состояний. Каждая групп-

на состояний в разбиении содержит состояния, которые еще не были отличены друг от друга, и в процессе работы выбираются пары состояний из разных групп, различаемые некоторой входной строкой.

Изначально разбиение состоит из двух групп: заключительные состояния и незаключительные. Основной шаг состоит в том, чтобы взять некоторую группу состояний, скажем $A = \{s_1, s_2, \dots, s_k\}$, и входной символ a и посмотреть, какие переходы имеют состояния s_1, s_2, \dots, s_k для этого входного символа. Если эти переходы представляют собой переходы в состояния, попадающие в две или более различных групп текущего разбиения, нужно разбить группу A на подгруппы так, чтобы для каждого из подмножеств A эти переходы ограничивались одной группой текущего разбиения. Предположим, что s_1 и s_2 переходят при входном символе a в состояния t_1 и t_2 , которые находятся в разных группах разбиения. Тогда необходимо разбить A , как минимум, на два подмножества, чтобы одно из них содержало s_1 , а второе — s_2 . Заметим, что состояния t_1 и t_2 различаются при помощи некоторой строки w (поскольку находятся в разных группах), а значит, состояния s_1 и s_2 различаются строкой aw .

Этот процесс разбиения повторяется до тех пор, пока не останется групп, которые следует разбить. Хотя мы и доказали, что состояния, попавшие в две разные группы, действительно различимы, мы не показали, что состояния, которые остались в одной группе, не могут быть различены никакими входными строками. Тем не менее это так, и интересующиеся теорией читатели могут обратиться за доказательством, например, к работе [194]. Оставим читателям и доказательство того факта, что ДКА, построенный из состояний, взятых по одному из каждой группы последнего разбиения (из которых затем выброшены “мертвые” состояния и состояния, недостижимые из начального), имеет состояний не больше, чем любой ДКА, допускающий тот же язык.

Алгоритм 3.6. Минимизация количества состояний ДКА

Вход. ДКА M с множеством состояний S ; множеством входных символов Σ ; переходами, определенными для всех состояний и входных символов; стартовым состоянием s_0 и множеством заключительных состояний F .

Выход. ДКА M' , допускающий тот же язык, что и M , и имеющий наименьшее возможное количество состояний.

Метод.

1. Построить начальное разбиение Π множества состояний с двумя группами: заключительные состояния F и незаключительные состояния $S - F$.
2. Применить процедуру, приведенную на рис. 3.45, к разбиению Π для построения нового разбиения Π_{new} .
3. Если $\Pi_{new} = \Pi$, то $\Pi_{final} = \Pi$, и перейти к шагу (4). В противном случае повторить шаг (2) с $\Pi := \Pi_{new}$.
4. Выбрать одно состояние в каждой группе разбиения Π_{final} в качестве представителя этой группы. Представители будут состояниями ДКА M' . Пусть s является представителем. Предположим, что для входного символа a в M существует переход из s в t . Пусть r — представитель группы, в которой находится t (r может являться t). Тогда M' имеет переход из s в r по a . Стартовым состоянием M' сделать представителя группы, содержащей стартовое состояние s_0 автомата M , а заключительными со-

- стояниями M' — представителей в F . Заметим, что каждая группа из Π_{final} либо состоит только из состояний из F , либо вовсе не имеет состояний из F .
5. Если M' имеет “мертвое” состояние, т.е. состояние d , которое не является заключительным и имеет переходы в себя для всех входных символов, удалим его из M' (все переходы в d из других состояний становятся неопределенными). Удалим также все состояния, не достижимые из стартового. \square

```

for каждая группа  $G \in \Pi$  do begin
    разделить  $G$  на подгруппы, такие, что два состояния
     $s$  и  $t$  из  $G$  находятся в одной и той же подгруппе
    тогда и только тогда, когда для всех входных
    символов  $a$  состояния  $s$  и  $t$  имеют переходы по  $a$  в
    состояния из одной и той же группы  $\Pi$ 
    /* В худшем случае состояние будет
       единственным в подгруппе */
    Заменить  $G$  в  $\Pi_{new}$  множеством всех созданных подгрупп
end

```

Рис. 3.45. Построение Π_{new}

Пример 3.24

Вернемся к ДКА, представленному на рис. 3.29. Начальное разбиение Π состоит из двух групп: (E) , заключительного состояния, и $(ABCD)$, незаключительных состояний. Для построения Π_{new} алгоритм на рис. 3.45 вначале рассматривает (E) . Поскольку эта группа состоит из единственного состояния и не может быть разделена в дальнейшем, (E) помещается в Π_{new} . Затем алгоритм рассматривает группу $(ABCD)$. Для входного символа a каждое из этих состояний имеет переход в B , так что при рассмотрении этого символа все состояния остаются в одной группе. Однако если на вход поступает символ b , то переходы из A , B и C ведут к членам группы $(ABCD)$, в то время как из D переход по b ведет к E , являющемуся членом другой группы. Таким образом, в Π_{new} группа $(ABCD)$ должна быть разбита на две новые группы — (ABC) и (D) , и Π_{new} принимает вид $(ABC)(D)(E)$.

При следующем проходе алгоритма, представленного на рис. 3.45, мы вновь не получим разбиений при входном символе a , но входной символ b заставляет разбить группу (ABC) на две новые группы, $(AC)(B)$, поскольку A и C при входном символе b переходят в C , а B — в состояние D , группа которого отлична от группы C . Поэтому после этого прохода алгоритма Π принимает вид $(AC)(B)(D)(E)$.

При очередном проходе алгоритма нельзя разбить ни одну группу, в которой находится только одно состояние. Следовательно, единственная возможность разбиения — группы (AC) . Но при входном символе a переходы из A и C ведут в одну и ту же группу B , а при входном символе b — в состояние C . Следовательно, после этого прохода $\Pi_{new} = \Pi$ и Π_{final} имеет вид $(AC)(B)(D)(E)$. Если в качестве представителя группы (AC) выберем A , а для остальных групп, естественно, — B , D и E , то получим автомат с минимальным числом состояний, таблица переходов которого приведена на рис. 3.46. Начальным является состояние A , а единственным заключительным — состояние E .

СОСТОЯНИЕ	ВХОДНОЙ СИМВОЛ	
	<i>a</i>	<i>b</i>
<i>A</i>	<i>B</i>	<i>A</i>
<i>B</i>	<i>B</i>	<i>D</i>
<i>D</i>	<i>B</i>	<i>E</i>
<i>E</i>	<i>B</i>	<i>A</i>

Рис. 3.46. Таблица переходов оптимизированного автомата

Например, в полученном автомате состояние *E* имеет переход в состояние *A* для входного символа *b*, поскольку *A* является представителем группы *C* и в исходном автомате имеется переход для входного символа *b* из состояния *E* в состояние *C*. Такое же изменение сделано и для перехода из состояния *A* для входного символа *b*; все остальные переходы скопированы с рис. 3.29. На рис. 3.46 нет “мертвых” состояний, и из стартового состояния *A* достижимы все остальные. \square

Минимизация состояний в лексических анализаторах

Чтобы применить к ДКА, построенному в разделе 3.7, процедуру минимизации состояний, необходим алгоритм 3.5 с начальным разбиением, в котором все состояния, указывающие различные токены, должны быть размещены в разных группах.

Пример 3.25

При использовании ДКА, представленного на рис. 3.37, начальное разбиение должно сгруппировать состояния 0137 и 7, поскольку они не указывают на наличие какого-либо токена; должны группироваться состояния 8 и 58, указывающие на токены a^*b^+ . Остальные состояния представляют собой группы из одного состояния. Однако обнаруживаем, что состояния 0137 и 7 относятся к разным группам, поскольку различаются посредством входного символа *a*; то же происходит и с состояниями 8 и 58 при входном символе *b*. Следовательно, ДКА на рис. 3.37 представляет собой автомат с минимальным количеством состояний. \square

Методы сжатия таблиц

Как уже говорилось, имеется множество путей реализации функции переходов конечного автомата. Процесс лексического анализа занимает значительную часть времени компиляции, поскольку это единственный процесс, который последовательно, символ за символом, считывает входную программу. Таким образом, для повышения эффективности следует уменьшить среднее количество операций, выполняемых при обработке одного символа. Если при реализации лексического анализатора используется ДКА, требуется эффективное представление функции переходов. Двухмерный массив, индексы которого представляют собой состояния и входные символы, обеспечивает наиболее быстрый доступ за счет неэкономного использования пространства (скажем, несколько сотен состояний на 128 входных символов¹⁷). Более компактной и в то же время более медленной схемой является использование связанных списков для хранения переходов

¹⁷ Авторы имеют в виду использование только ASCII-символов, без использования символов национальных алфавитов (о Unicode речь не идет вовсе). — Прим. ред.

из каждого состояния, с переходом “по умолчанию” в конце списка. Наиболее часто выполняемый переход является очевидным кандидатом в конец списка.¹⁸

Имеется и более остроумная реализация, которая сочетает в себе быстрый доступ представления в виде двухмерного массива и компактность списков. В ней мы используем структуру данных, состоящих из четырех массивов с состояниями в качестве индексов, как показано на рис. 3.47¹⁹. Массив *base* используется для определения расположения записей каждого состояния, хранящихся в массивах *next* и *check*. Массив *default* применяется для определения альтернативной позиции, если текущее положение не соответствует исследуемому состоянию.

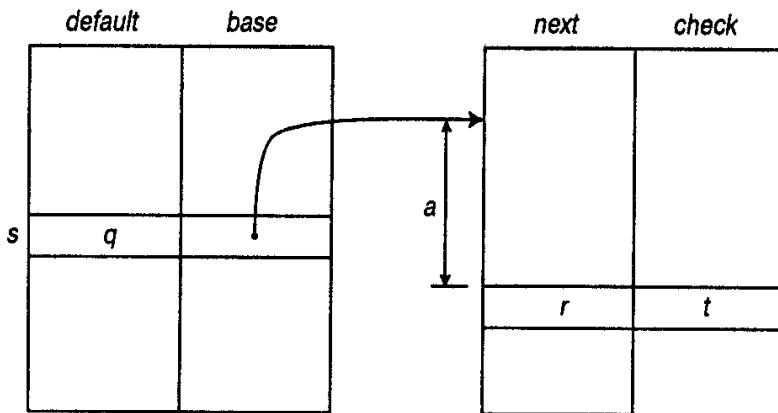


Рис. 3.47. Структура данных для представления таблиц переходов

Для вычисления *nextstate(s, a)*, перехода из состояния *s* для входного символа *a*, вначале обращаемся к массивам *next* и *check*, в частности, находим их записи для состояния *s* по индексу *l=base[s]+a*, где символ *a* рассматривается как целое число. *next[l]* представляет целевое состояние перехода по символу *a*, если *check[l]=s*. Если же это не так, находим *q=default[s]* и рекурсивно повторяем процедуру с использованием *q* вместо *s*. Процедура выглядит следующим образом.

```
procedure nextstate(s, a);
  if check[base[s] + a] = s then
    return next[base[s] + a]
  else
    return nextstate(default[s], a)
```

Цель применения структуры (рис. 3.47) состоит в том, чтобы сделать короткими массивы *next/check* благодаря использованию подобия состояний. Например, *q*, являющееся состоянием по умолчанию для *s*, может быть состоянием, которое говорит, что мы работаем с идентификатором (как, например, состояние 10 на рис. 3.13). Возможно, в состояние *s* мы попали после входной строки *th*, которая может служить как префиксом идентификатора, так и ключевого слова *then*. Тогда, если очередной входной символ — *e*, необходимо перейти в отдельное состояние, запоминающее, что мы считали входную строку *the*; в противном случае состояние *s* ведет себя так же, как и состояние *q*. Следо-

¹⁸ Имеется множество способов ускорить обращения к спискам, например технология “самоорганизации” (см. раздел 6.1 книги Кнут Д. *Искусство программирования. Т.3*. — М.: Издательский дом “Вильямс”, 2000). — Прим. ред.

¹⁹ На практике имеется еще один массив, индексированный состояниями *s* и предоставляющий шаблон поиска при входе в состояние *s*. Эта информация получается из состояний НКА, создающих состояние *s* ДКА.

вательно, устанавливаем $check[base[s]+e]$ равным s , а $next[base[s]+e]$ равным состоянию, соответствующему входной строке the .

Мы можем оказаться неспособны выбрать значения $base$ так, чтобы не оставалось неиспользованных записей $next/check$. Однако опыт показывает, что простейшая стратегия присвоения $base$ минимального значения, такого, что специальные записи могут заполняться без конфликтов с существующими, достаточно хороша. Более того, она использует объем памяти, лишь ненамного превышающий минимально возможный.

Можно сократить массив $check$ до массива, индексированного состояниями, если в ДКА входящие в каждое состояние i дуги имеют одну и ту же метку a . Для реализации этой схемы устанавливаем $check[i]=a$ и заменяем проверку во второй строке процедуры $nextstate$ проверкой

```
if check[next[base[s]+a]] = a then
```

Упражнения

3.1. Что представляет собой входной алфавит для каждого из перечисленных языков?

- a) Pascal
- b) C
- c) Fortran 77
- d) Ada
- e) Lisp

3.2. Каковы соглашения по использованию пробелов в каждом из языков из упр. 3.1?

3.3. В следующей программе определите лексемы, образующие токены. Дайте токенам обоснованные атрибуты.

- a) Pascal

```
function max (i, j : integer) : integer;
{ Возвращает максимальное из целых чисел i и j }
begin
    if i > j then max := i
    else max := j;
end;
```

- b) C

```
int max(i,j) int i, j;
/* Возвращает максимальное из целых чисел i и j */
{
    return i>j?i:j;
}
```

- c) Fortran 77

```
FUNCTION MAX (I, J)
C Возвращает максимальное из целых чисел i и j
    IF (I .GT. J) THEN
        MAX = I
    ELSE
        MAX = J
    END IF
RETURN
```

- 3.4.** Напишите программу для функции `nextchar()` из раздела 3.4 с использованием схемы буферизации с ограничителями, описанной в разделе 3.2.
- 3.5.** Данна строка длиной n . Укажите количество
- префиксов,
 - суффиксов,
 - подстрок,
 - собственных префиксов,
 - подпоследовательностей.
- *3.6.** Опишите языки, определяемые следующими регулярными выражениями.
- $0(0|1)^*0$
 - $((\epsilon|0)1^*)^*$
 - $(0|1)^*0(0|1)(0|1)$
 - $0^*10^*10^*10^*$
 - $(00|11)^*((01|10)(00|11)^*(01|10)(00|11)^*)^*$
- *3.7.** Напишите регулярные выражения для следующих языков.
- Все строки из букв, содержащие пять гласных в алфавитном порядке.
 - Все строки из букв, в которых буквы находятся в прямом лексикографическом порядке.
 - Комментарии, состоящие из строк, окруженных /* и */ и не содержащих /*, кроме как в двойных кавычках "".
 - Все строки из неповторяющихся цифр.
 - Все строки из цифр, в которых имеется не более одной повторяющейся цифры.
 - Все строки из нулей и единиц, с четным числом нулей и нечетным числом единиц.
 - Множество шахматных ходов, таких как e2-e4 или kp:f2x.
 - Все строки из нулей и единиц, не содержащие подстроки 011.
 - Все строки из нулей и единиц, не содержащие подпоследовательности 011.
- 3.8.** Определите лексическую форму числовых констант в языках из упр. 3.1.
- 3.9.** Определите лексическую форму идентификаторов и ключевых слов в языках из упр. 3.1.
- 3.10.** Конструкции регулярных выражений, разрешенные в Lex, представлены на рис. 3.48 в порядке убывания приоритета. Здесь c означает любой отдельный символ, r — регулярное выражение, а s — строку.

ВЫРАЖЕНИЕ	СООТВЕТСТВИЕ	ПРИМЕР
c	Любой символ c , не являющийся знаком операции	a
$\backslash c$	Символ c буквально	*
$"s"$	Строка s буквально	"***"
.	Любой символ, кроме символа новой строки	a.*b
$^$	Начало строки	^abc
$$$	Конец строки	abc\$

Рис. 3.48. Регулярные выражения Lex

ВЫРАЖЕНИЕ	СООТВЕТСТВИЕ	ПРИМЕР
$[s]$	Любой символ из s	$[abc]$
$[^s]$	Любой символ, не входящий в s	$[^abc]$
r^*	Нуль или более r	a^*
r^+	Одно или более r	a^+
$r^?$	Нуль или одно r	$a^?$
$r\{m,n\}$	От m до n повторений r	$a\{1,5\}$
r_1r_2	r_1 , затем r_2	ab
$r_1 r_2$	r_1 или r_2	$a b$
(r)	r	$(a b)$
r_1/r_2	r_1 , если за ним следует r_2	$abc/123$

Рис. 3.48. Регулярные выражения Lex (окончание)

a) Специальное значение знаков операций

\backslash " . ^ \$ [] * + ? { } | /

должно быть отменено при рассмотрении такого символа как обычного, что достигается либо помещением данного символа в двойные кавычки (выражение "s" означает s буквально, без включения в строку s кавычек; так, "/*" соответствует строке $**$), либо использованием символа \ (аналог "***" в этом случае записывается как $*\backslash*$). Заметим, сам по себе символ * в регулярных выражениях представляет замыкание Клини. Напишите регулярное выражение Lex, соответствующее строке "\.

- b) В Lex дополнением класса символов является класс, в котором первым символом является ^ . Класс-дополнение соответствует любому символу, не входящему в основной класс. Таким образом, $[^a]$ соответствует любому символу, не являющемуся а, $[^A-Za-z]$ — любому символу, не являющемуся буквой и т.д. Покажите, что для каждого регулярного определения с классами-дополнениями существует эквивалентное регулярное выражение без использования дополнений.
- c) Регулярное выражение $r\{m,n\}$ представляет от m до n повторений шаблона r . Например, $a\{1,5\}$ соответствует строкам, состоящим из символов а — от одного до пяти. Покажите, что для каждого регулярного выражения с таким оператором повторения существует эквивалентное регулярное выражение без этого оператора.
- d) Оператор ^ соответствует левому концу строки. Это тот же оператор, что и используемый в классе-дополнении. Какой именно смысл вкладывается в этот оператор, определяется контекстом его применения. Оператор \$ соответствует правому концу строки. Например, $^[^aeiou]*$$ соответствует любой строке, не содержащей гласной в нижнем регистре. Имеется ли для каждого регулярного выражения, содержащего операторы ^ и \$, эквивалентное регулярное выражение без этих операторов?

- 3.11. Напишите Lex-программу, которая копирует файл, заменяя каждую непустую последовательность пробельных символов (пробелов, табуляций) одним символом пробела.

- 3.12. Напишите Lex-программу, которая копирует программу на языке Fortran, заменяя все вхождения строки DOUBLE PRECISION на REAL.
- 3.13. Используя ваши спецификации для ключевых слов и идентификаторов Fortran 77 из упр. 3.9, укажите токены в следующих инструкциях.

```
IF(I) = TOKEN
IF(I) ASSIGN5TOKEN
IF(I) 10,20,30
IF(I) GOTO15
IF(I) THEN
```

Можете ли вы записать ваши спецификации для ключевых слов и идентификаторов на языке Lex?

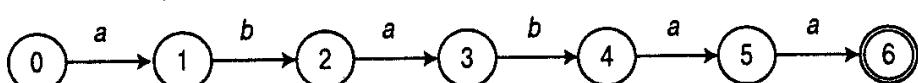
- 3.14. В операционной системе UNIX оболочка *sh* использует операторы, приведенные на рис. 3.49, при указании имен файлов для обозначения множества файлов. Например, выражение *.*o* означает все файлы, имена которых заканчиваются на *.o*; *sort.?* соответствует всем именам вида *sort.c*, где *c* — произвольный символ. Классы символов могут записываться как, например, в случае [a-z]. Покажите, как выражения для имен файлов могут быть представлены регулярными выражениями.

ВЫРАЖЕНИЕ	СООТВЕТСТВУЕТ	ПРИМЕР
' <i>s</i> '	строке <i>s</i> буквально	'\'
\ <i>c</i>	символу <i>c</i> буквально	'\'
*	любой строке	*. <i>o</i>
?	любому символу	<i>sort1.?</i>
[<i>s</i>]	любому символу из <i>s</i>	<i>sort.[cso]</i>

Рис. 3.49. Выражения для имен файлов в оболочке *sh*

- 3.15. Модифицируйте алгоритм 3.1 для поиска во входном потоке наиболее длинного префикса, допустимого ДКА.
- 3.16. Постройте недетерминированные конечные автоматы для приведенных ниже регулярных выражений с использованием алгоритма 3.3. Покажите последовательность перемещений для каждого автомата при разборе входной строки *ababbab*.
- (a|b)*
 - (a*|b*)*
 - ((ε|a)b*)*
 - (a|b)*abb(a|b)*
- 3.17. Преобразуйте недетерминированные конечные автоматы из упр. 3.16 в детерминированные с использованием алгоритма 3.2. Покажите последовательность перемещений для каждого автомата при разборе входной строки *ababbab*.
- 3.18. Постройте детерминированные конечные автоматы для регулярных выражений из упр. 3.16 с использованием алгоритма 3.5. Сравните размеры полученных автоматов с размерами автоматов из упр. 3.17.
- 3.19. Постройте детерминированный конечный автомат по диаграмме переходов для токенов, приведенных на рис. 3.10.

- 3.20. Расширьте таблицу на рис. 3.40, включив в нее операторы регулярных выражений ? и +.
- 3.21. Минимизируйте число состояний в автоматах из упр. 3.19 с использованием алгоритма 3.6.
- 3.22. Можно доказать, что два регулярных выражения эквивалентны, показав, что их ДКА с минимальным числом состояний одинаковы с точностью до имен состояний. Используя этот метод, докажите, что все приведенные ниже регулярные выражения эквивалентны.
- $(a|b)^*$
 - $(a^*|b^*)^*$
 - $((\epsilon|a)b^*)^*$
- 3.23. Постройте детерминированные конечные автоматы с минимальным числом состояний для следующих регулярных выражений.
- $(a|b)^*a(a|b)$
 - $(a|b)^*a(a|b)(a|b)$
 - $(a|b)^*a(a|b)(a|b)(a|b)$
 - **d) Докажите, что любой детерминированный конечный автомат для регулярного выражения $(a|b)^*a(a|b)(a|b)\dots(a|b)$, завершающегося $n-1$ подвыражением $(a|b)$, должен иметь как минимум 2^n состояний.
- 3.24. Постройте представление, показанное на рис. 3.47, для таблицы переходов из упр. 3.19. Отберите состояния по умолчанию и испытайте два следующих метода построения массива *next*; сравните объем памяти, используемый при каждом методе.
- Размещаем записи в массиве *next*, начиная с состояний с большим количеством записей, отличающихся от состояний по умолчанию.
 - Размещаем записи состояний в массиве *next* в случайном порядке.
- 3.25. Вариант схемы сжатия таблицы из раздела 3.9 можно использовать без рекурсивной процедуры *nextstate* путем использования фиксированных положений по умолчанию для каждого состояния. Постройте представление, показанное на рис. 3.47, для таблицы переходов из упр. 3.19 с помощью такой нерекурсивной технологии. Сравните количество требуемой при этом памяти с количеством памяти, израсходованной в упр. 3.24.
- 3.26. Пусть $b_1b_2\dots b_m$ — строка шаблона, называемая *ключевым словом*. Лучом²⁰ ключевого слова является диаграмма переходов с $m+1$ состоянием, в которой каждое состояние соответствует префиксу ключевого слова. Для $1 \leq s \leq m$ существует переход из состояния $s-1$ в состояние s по символу b_s . Начальное и заключительное состояния соответствуют пустой строке и полному ключевому слову соответственно. Луч для ключевого слова *ababa* выглядит следующим образом.



²⁰ В оригинале — *trie*. Здесь использован перевод этого термина из книги Кнут Д. *Искусство программирования. Т.3.* — М.: Издательский дом “Вильямс”, 2000 (раздел 6.3, с. 527). — Прим. перев.

Теперь в каждом состоянии диаграммы переходов определим *функцию отказа* f . Предположим, что состояния s и t представляют префиксы u и v ключевого слова. Тогда можно определить, что $f(s) = t$ тогда и только тогда, когда v — наилдлиннейший собственный суффикс u , который одновременно является префиксом ключевого слова. Для приведенного выше луча функция отказа выглядит следующим образом.

s	1	2	3	4	5	6
$f(s)$	0	0	1	2	3	1

Например, состояния 3 и 1 представляют префиксы aba и a ключевого слова $ababaa$. $f(3) = 1$, поскольку a — наилдлиннейший собственный суффикс aba , являющийся префиксом ключевого слова.

- a) Постройте функцию отказа для ключевого слова $abababaab$.
- *b) Пусть состояния луча — $0, 1, \dots, m$, причем стартовым является состояние 0. Покажите, что алгоритм, приведенный на рис. 3.50, корректно вычисляет функцию отказа.
- *c) Покажите, что в процессе работы алгоритма (рис. 3.50) инструкция присвоения $t := f(t)$ во внутреннем цикле выполняется не более m раз.
- *d) Покажите, что время работы алгоритма — $O(m)$.

```
/* Вычисление функции отказа  $f$  для  $b_1b_2\dots b_m$  */
t:=0; f(1):=0;
for s:=1 to m-1 do begin
    while t > 0 и  $b_{s+1} \neq b_{t+1}$  do t := f(t);
    if  $b_{s+1} = b_{t+1}$  then begin t:=t+1; f(s+1):=t end;
    else f(s+1) := 0
end
```

Рис. 3.50. Алгоритм вычисления функции отказа из упр. 3.26

- 3.27. Алгоритм КМП (Кнута–Морриса–Пратта) (рис. 3.51) использует функцию отказа f из упражнения 3.26 для определения, является ли ключевое слово $b_1\dots b_m$ подстрокой целевой строки $a_1\dots a_n$. Состояния в луче для $b_1\dots b_m$ пронумерованы от 0 до m , как в упр. 3.26b.

```
/* Содержит ли  $a_1\dots a_n$  в качестве подстроки  $b_1\dots b_m$  */
s := 0;
for i := 1 to n do begin
    while s > 0 и  $a_i \neq b_{s+1}$  do s := f(s);
    if  $a_i = b_{s+1}$  then s := s+1;
    if s = m then return "да"
end;
return "нет"
```

Рис. 3.51. Алгоритм КМП

- a) Примените алгоритм КМП для определения, является ли $ababaa$ подстрокой $abababaab$.

- *b) Докажите, что алгоритм КМП возвращает “да” только тогда, когда $b_1 \dots b_m$ является подстрокой $a_1 \dots a_n$.
- *c) Покажите, что время работы алгоритма КМП — $O(m+n)$.
- *d) Для данного ключевого слова y покажите, что функция отказа может использоваться для построения ДКА с $|y|+1$ состояниями для регулярного выражения $. * y . *$ за время $O(|y|)$, где $.$ означает любой входной символ.

****3.28.** Определим *период* строки s как целое число p , такое, что s может быть выражено в виде $(uv)^k$ и для некоторого $k \geq 0$, где $|uv| = p$ и v не является пустой строкой. Например, 2 и 4 — периоды строки $abababa$.

- a) Покажите, что p — период строки s тогда и только тогда, когда $st = us$ для некоторых строк t и u длины p .
- b) Пусть p и q — периоды строки s и $p+q \leq |s| + \gcd(p, q)$. Покажите, что $\gcd(p, q)$ является периодом s ($\gcd(p, q)$ — наибольший общий делитель (greatest common divisor) p и q).
- c) Пусть $sp(s_i)$ — наименьший период префикса длины i строки s . Покажите, что функция отказа f обладает тем свойством, что $f(j) = j - sp(s_{j-1})$.

***3.29.** Пусть *кратчайший повторяющийся префикс* строки s — это кратчайший префикс u строки s , такой, что $s = u^k$ для некоторого $k \geq 1$. Например, ab является кратчайшим повторяющимся префиксом $abababab$, а aba — кратчайшим повторяющимся префиксом aba . Разработайте алгоритм, который находит кратчайший повторяющийся префикс строки s за время $O(|s|)$. Указание: воспользуйтесь функцией отказа из упр. 3.26.

3.30. Стока *Фибоначчи* определяется следующим образом.

$$s_1 = b$$

$$s_2 = a$$

$$s_k = s_{k-1}s_{k-2} \text{ для } k > 2.$$

Например, $s_3 = ab$, $s_4 = aba$, $s_5 = abaab$.

- a) Чему равна длина s_n ?
- **b) Чему равен минимальный период s_n ?
- c) Постройте функцию отказа для s_6 .
- *d) Используя индукцию, покажите, что функция отказа для s_n может быть выражена как $f(j) = j - |s_{k-1}|$, где k — такое, что $|s_k| \leq j + 1 < |s_{k+1}|$ для $1 \leq j \leq |s_n|$.
- e) Примените алгоритм КМП для определения, является ли s_6 подстрокой строки s_7 .
- f) Постройте ДКА для регулярного выражения $. * s_6 . *$.
- **g) Чему равно максимальное число последовательных применений функции отказа в алгоритме КМП при определении, является ли строка s_k подстрокой s_{k+1} ?

- 3.31. Можно расширить концепцию луча и функции отказа из упр. 3.26 от одного ключевого слова до множества ключевых слов. Каждое состояние в луче соответствует префиксу одного или нескольких ключевых слов. Начальное состояние соответствует пустой строке, а состояние, соответствующее полному ключевому слову, является заключительным. Дополнительные состояния могут быть сделаны заключительными в процессе вычисления функции отказа. Диаграмма переходов для набора ключевых слов {he, she, his, hers} показана на рис. 3.52.

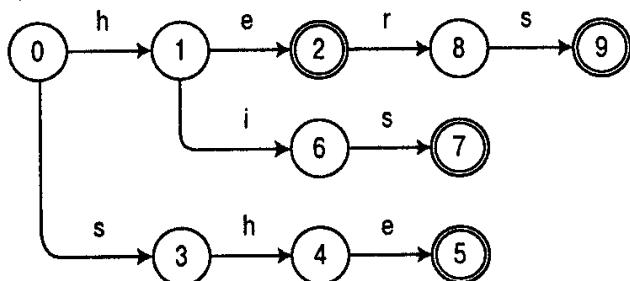


Рис. 3.52. Луч для ключевых слов {he, she, his, hers}

Определим для луча функцию переходов g (отображающую пары состояния-символ в состояния) как $g(s, b_{j+1}) = s'$, если состояние s соответствует префиксу $b_1 \dots b_j$ некоторого ключевого слова, а s' — префиксу $b_1 \dots b_j b_{j+1}$. Если s_0 — начальное состояние, определим $g(s_0, a) = s_0$ для всех входных символов a , которые не являются начальными символами ни одного из ключевых слов. Затем определим, что $g(s, a) = \text{fail}$ для любого не определенного перехода. Заметьте, что для старто-вого состояния *fail*-переходов не имеется.

Предположим, что состояния s и t представляют префиксы u и v некоторых ключевых слов. $f(s) = t$ тогда и только тогда, когда v — наидлиннейший собственный суффикс u , который является префиксом некоторого ключевого слова. Функция отказа f для приведенной выше диаграммы переходов выглядит следующим образом.

s	1	2	3	4	5	6	7	8	9
$f(s)$	0	0	0	1	2	0	3	0	3

Например, состояния 4 и 1 представляют префиксы sh и h. $f(4) = 1$, поскольку h — наидлиннейший собственный суффикс sh, который одновременно является префиксом некоторого ключевого слова. Функция отказа f может быть вычислена для состояний возрастающей глубины с использованием алгоритма, приведенного на рис. 3.53. Глубина состояния представляет собой его расстояние от начального состояния.

```

for каждое состояние  $s$  глубины 1 do
   $f(s) := s_0$ ;
  for каждое состояние  $s_d$  глубины  $d$  и символ  $a$ , такой,
    что  $g(s_d, a) = s'$  do begin
       $s := f(s_d)$ ;
      while  $g(s, a) = \text{fail}$  do  $s := f(s)$ ;
  
```

```

    f(s') := g(s, a);
end

```

Рис. 3.53. Алгоритм вычисления функции отказа для луча ключевых слов

Заметим, поскольку $g(s_0, c) \neq \text{fail}$ для любого символа c , цикл `while` на рис. 3.53 гарантированно завершается. После присвоения $f(s') := g(s, a)$, если $g(t, a)$ — заключительное состояние, мы также делаем состояние s' заключительным (если, конечно, оно еще не является таковым).

- Постройте функцию отказа для множества ключевых слов $\{aaa, abaaa, aba-baaa\}$.
 - * Покажите, что алгоритм, приведенный на рис. 3.53, правильно вычисляет функцию отказа.
 - * Покажите, что функция отказа может быть вычислена за время, пропорциональное сумме длин ключевых слов.
- 3.32. Пусть g — функция переходов, а f — функция отказа из упр. 3.31 для множества ключевых слов $K = \{y_1, y_2, \dots, y_k\}$. Алгоритм АК (Ахо–Корасика), приведенный на рис. 3.54, использует g и f для определения, содержит ли строка $a_1\dots a_n$ подстроку, являющуюся ключевым словом. Состояние s_0 — стартовое состояние диаграммы переходов K , а F — множество заключительных состояний.

```

/* Содержит ли строка a1...an в качестве
подстроки ключевое слово */
s := s0;
for i := 1 to n do begin
    while g(s, a) = fail do s := f(s);
    s := g(s, a);
    if s ∈ F then return "да"
end;
return "нет"

```

Рис. 3.54. Алгоритм АК

- Примените описанный алгоритм ко входной строке `ushers` с использованием функций переходов и отказа из упр. 3.31.
- * Докажите, что алгоритм возвращает “да” тогда и только тогда, когда некоторое ключевое слово y_i является подстрокой $a_1\dots a_n$.
- * Покажите, что в процессе работы с входной строкой длины n алгоритм выполняет не более $2n$ переходов между состояниями.
- * Покажите, что на основании диаграммы переходов и функции отказа для множества ключевых слов $\{y_1, y_2, \dots, y_k\}$ для регулярного выражения $\cdot \cdot^*(y_1|y_2|\dots|y_k) \cdot^*$ за время $O(|k|)$ может быть построен ДКА с не более чем $\sum_{i=1}^k |y_i| + 1$ состояниями.
- Модифицируйте алгоритм для вывода каждого ключевого слова, найденного в строке.

- 3.33. Воспользуйтесь алгоритмом из упр. 3.32 для построения лексического анализатора, работающего с ключевыми словами Pascal.
- 3.34. Определим $lcs(x, y)$ — *наи длиннейшую общую подпоследовательность* (longest common subsequence) двух строк x и y — как строку, которая является подпоследовательностью как x , так и y , и имеет наибольшую длину среди всех таких подпоследовательностей. Так, $lcs(\text{striped}, \text{tiger}) = \text{tie}$. Определим $d(x, y)$, *расстояние* между x и y как минимальное число вставок и удалений, требующихся для преобразования x в y (например, $d(\text{striped}, \text{tiger}) = 6$).
- Покажите, что для любых двух строк x и y расстояние между ними и длина их наибольшей общей подпоследовательности связаны соотношением $d(x, y) = |x| + |y| - (2 * |lcs(x, y)|)$.
 - * Разработайте алгоритм, который в качестве входа получает две строки, x и y , и выдает наибольшую общую подпоследовательность этих строк.
- 3.35. Определим *расстояние редактирования* $e(x, y)$ между двумя строками x и y как минимальное число вставок, удалений и замещений символов, требующихся для преобразования x в y . Пусть $x = a_1 \dots a_m$ и $y = b_1 \dots b_n$. $e(x, y)$ может быть вычислено с помощью алгоритма динамического программирования с использованием массива расстояний $d[0..m, 0..n]$, в котором $d[i, j]$ — расстояние редактирования между $a_1 \dots a_i$ и $b_1 \dots b_j$. Алгоритм, приведенный на рис. 3.55, используется для вычисления матрицы d . Функция $repl(a_i, b_j)$ представляет собой стоимость замещения символов:
- $$repl(a_i, b_j) = \begin{cases} 0, & \text{если } a_i = b_j \\ 1, & \text{если } a_i \neq b_j \end{cases}$$
- ```

for i:=0 to m do d[i, 0]:=i;
for j:=0 to n do d[0, j]:=j;
for i:=1 to m do
 for j:=1 to n do
 D[i, j]:=min(d[i-1, j-1] + repl(a_i, b_j),
 d[i-1, j] + 1,
 d[i, j-1] + 1)

```

Рис. 3.55. Алгоритм для вычисления расстояния редактирования между двумя строками

- Как связаны между собой расстояние из упр. 3.34 и расстояние редактирования?
  - Используйте алгоритм, приведенный на рис. 3.55, для вычисления расстояния редактирования между  $ababb$  и  $babaaa$ .
  - Разработайте алгоритм, выводящий минимальную последовательность действий, необходимую для преобразования  $x$  в  $y$ .
- 3.36. Разработайте алгоритм, который получает в качестве входа строку  $x$  и регулярное выражение  $r$  и выдает строку  $y \in L(r)$ , такую, что  $d(x, y)$  имеет минимально возможное значение (здесь  $d$  — расстояние, определенное в упр. 3.34).

# Программные упражнения

- P3.1.** Напишите лексический анализатор на языке программирования Pascal или С для токенов, показанных на рис. 3.10.
- P3.2.** Напишите спецификацию для токенов Pascal и постройте диаграммы переходов на основе этой спецификации. Воспользуйтесь этими диаграммами для реализации лексического анализатора для Pascal на языке программирования типа Pascal или С.
- P3.3.** Завершите программу Lex, приведенную на рис. 3.18. Сравните размер и скорость работы полученного посредством Lex лексического анализатора с программой, написанной в упр. P3.1.
- P3.4.** Напишите спецификацию Lex для токенов Pascal и используйте компилятор Lex для построения лексического анализатора Pascal.
- P3.5.** Напишите программу, которая получает на вход регулярное выражение и имя файла, а также выводит все строки из указанного файла, содержащие подстроки, описываемые данным регулярным выражением.
- P3.6.** Добавьте к программе Lex (рис. 3.18) систему восстановления после ошибок, позволяющую продолжать работу после обнаружения ошибок.
- P3.7.** Напишите лексический анализатор на основе ДКА, построенного в упр. 3.18, и сравните его с построенным в упр. P3.1 и P3.3.
- P3.8.** Создайте инструмент, который строит лексический анализатор на основе регулярного выражения, описывающего множество токенов.

## Библиографические примечания

Ограничения, налагаемые на лексические аспекты языка, часто определяются средой, в которой создается язык. В 1954 году, когда разрабатывался Fortran, обычным способом ввода данных являлись перфокарты. Игнорирование пробелов в Fortran было вызвано, в частности, тем, что операторы, подготовливавшие перфокарты на основе рукописного текста, часто ошибались при подсчете пробелов [40]. Разделение аппаратного представления Algol 58 и официального описания языка оказалось компромиссом, достигнутым в результате настойчивости одного из членов комитета, заявившего, что он никогда не использует точку в записи действительного числа [450].

Методы буферизации входного потока представлены Кнутом в [257]. В работе Фельдмана [131] обсуждаются сложности распознавания токенов Fortran 77 на практике.

Первоначально регулярные выражения изучались Клини [250], которого интересовало описание событий, представленных конечно-автоматной моделью нервной деятельности (Мак-Каллок и Питтс, [308]). Первыми вопросы минимизации конечных автоматов изучали Хаффман [199] и Мур [320]. Эквивалентность детерминированных и недетерминированных автоматов, а также их применимость к распознаванию языков были показаны Рабином и Скоттом [358]. Мак-Нотон и Я마다 [312] описали алгоритм построения детерминированного конечного автомата непосредственно по регулярному выражению. Дополнительную информацию о теории регулярных выражений можно найти в работе Хопкрофта и Ульмана [194].

Разработчики компиляторов быстро оценили пользу от инструментария для создания лексических анализаторов по регулярным выражениям. Одной из первых работ, описы-

вающих такие системы, стала [221]. Язык Lex, рассмотренный в этой главе, разработан Леском [282] и использовался при разработке множества компиляторов, работающих в UNIX. Схема компактного представления таблиц переходов из раздела 3.9 разработана Джонсоном, использовавшим ее в реализации генератора синтаксических анализаторов Yacc [215]. Другие схемы сжатия таблиц рассматриваются в работе [110].

Проблема компактной реализации таблиц переходов теоретически изучалась, в частности, Таржаном и Яо [428], а также Фредманом, Комлёсом и Семереди [144]. На основе этой работы Кормак, Хорспул и Кайзерверт [97] представили совершенный алгоритм хеширования.

Регулярные выражения и конечные автоматы использовались и во множестве других программ, а не только в компиляторах. Так, различные текстовые редакторы часто используют регулярные выражения для контекстного поиска в тексте. Томпсон [431], например, описал построение недетерминированного конечного автомата по регулярному выражению (алгоритм 3.3) в контексте текстового редактора QED. В UNIX имеется три программы поиска общего назначения, использующие регулярные выражения, — grep, egrep и fgrep. Программа grep не позволяет использовать объединения или скобки для группирования в регулярных выражениях; в ней используются алгоритмы 3.3 и 3.4. Регулярные выражения в egrep подобны используемым в Lex, за исключением итераций и прогностического оператора. egrep применяет детерминированные конечные автоматы с отложенным построением состояний, описанным в разделе 3.7. fgrep производит поиск шаблона, состоящего из множеств ключевых слов, с использованием алгоритма Ахо–Корасика [5], рассмотренного в упр. 3.31 и 3.32. В работе Ахо [4] рассматривается относительная производительность перечисленных программ.

Регулярные выражения широко использовались в системах обработки текстов, языках запросов баз данных и языках обработки файлов типа AWK [14]. Джарвис [210] использовал регулярные выражения для описания несовершенств в печатных схемах, а Черри [78] применил алгоритм, описанный в упр. 3.32, для поиска редких слов в манускриптах.

Алгоритм, описанный в упр. 3.26 и 3.27, разработан Кнутом, Моррисом и Праттом [263]. Другой эффективный алгоритм поиска в строках был изобретен Бойером и Муром [61], которые показали, что в процессе поиска соответствия можно не рассматривать все символы в целевой строке. Как было доказано Харрисоном [174], хеширование также может использоваться в качестве эффективной технологии поиска строк.

Понятие наилиннейшей общей подпоследовательности, рассмотренное в упр. 3.34, использовалось при разработке системной программы сравнения файлов UNIX diff (Хант и Мак-Илрой [200]). Эффективный алгоритм вычисления наилиннейшей общей подпоследовательности описан Хантом и Шимански [201]. Алгоритм для вычисления минимального расстояния редактирования из упр. 3.36 был разработан Вагнером и Фишером [440]. Работа Вагнера [439] содержит решение упр. 3.36. В труде Санкова и Краскала [389] увлекательно обсуждаются многочисленные применения алгоритмов распознавания минимальных расстояний, от изучения генетических кодов до задач обработки речи.

## ГЛАВА 4

# Синтаксический анализ

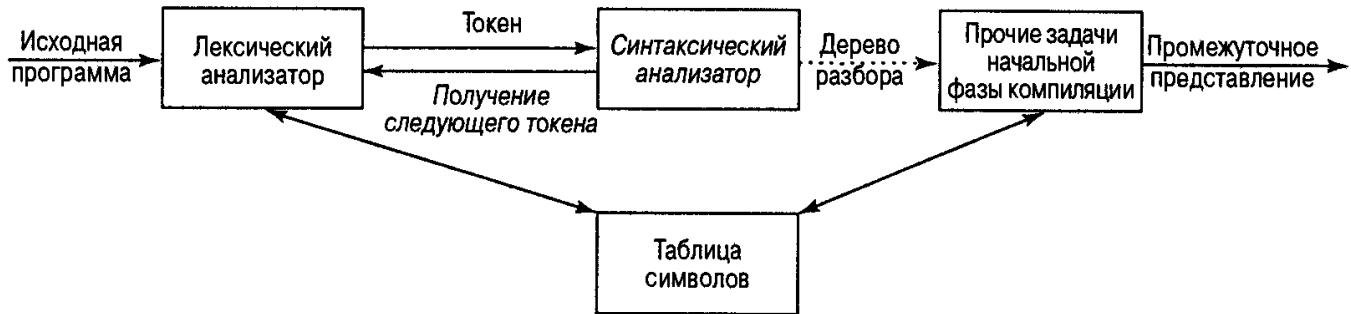
Каждый язык программирования имеет правила, которые предписывают синтаксическую структуру корректных программ. В Pascal, например, программа создается из блоков, блок — из инструкций, инструкции — из выражений, выражения — из токенов и т.д. Синтаксис конструкций языка программирования может быть описан с помощью контекстно-свободных грамматик или нотации БНФ (Backus-Naur Form, форма Бэкуса-Наура), о которой шла речь в разделе 2.2. Грамматики обеспечивают значительные преимущества разработчикам языков программирования и создателям компиляторов.

- Грамматика дает точную и при этом простую для понимания синтаксическую спецификацию языка программирования.
- Для некоторых классов грамматик мы можем автоматически построить эффективный синтаксический анализатор, который определяет, корректна ли структура исходной программы. Дополнительным преимуществом автоматического создания анализатора является возможность обнаружения синтаксических неоднозначностей и других сложных для распознавания конструкций языка, которые иначе могли бы остаться незамеченными на начальных фазах создания языка и его компилятора.
- Правильно построенная грамматика придает языку программирования структуру, которая способствует облегчению трансляции исходной программы в объектный код и выявлению ошибок. Для преобразования описаний трансляции, основанных на грамматике языка, в рабочие программы имеется соответствующий программный инструментарий.
- Со временем языки эволюционируют, обогащаясь новыми конструкциями и выполняя новые задачи. Добавление конструкций в язык окажется более простой задачей, если существующая реализация языка основана на его грамматическом описании.

Материал этой главы посвящен методам разбора, обычно используемым в компиляторах. Вначале мы представим основные концепции, затем — технологии, пригодные для реализации вручную, и, наконец, алгоритмы, используемые средствами автоматизации. Поскольку программы могут содержать синтаксические ошибки, рассматриваемые методы разбора будут включать технологии восстановления после часто встречающихся ошибок.

## 4.1. Роль синтаксического анализатора

В нашей модели компилятора синтаксический анализатор получает строку токенов от лексического анализатора, как показано на рис. 4.1, и проверяет, может ли эта строка порождаться грамматикой исходного языка. Он также сообщает о всех выявленных ошибках, причем достаточно внятно и полно. Кроме того, он должен уметь обрабатывать обычные, часто встречающиеся ошибки и продолжать работу с оставшейся частью программы.



*Рис. 4.1. Место синтаксического анализатора в модели компилятора*

Имеется три основных типа синтаксических анализаторов грамматик. Универсальные методы разбора, такие как алгоритмы Кока-Янгера-Касами или Эрли, могут работать с любой грамматикой (см. библиографические примечания). Однако эти методы слишком неэффективны для использования в промышленных компиляторах. Методы, обычно применяемые в компиляторах, классифицируются как нисходящие (сверху вниз, top-down) или восходящие (снизу вверх, bottom-up). Как явствует из названий, нисходящие синтаксические анализаторы строят дерево разбора сверху (от корня) вниз (к листьям), тогда как восходящие начинают с листьев и идут к корню. В обоих случаях входной поток синтаксического анализатора сканируется посимвольно слева направо.

Наиболее эффективные нисходящие и восходящие методы работают только с подклассами грамматик, однако некоторые из этих подклассов, такие как LL- и LR-грамматики, достаточно выразительны для описания большинства синтаксических конструкций языков программирования. Реализованные вручную синтаксические анализаторы чаще работают с LL-грамматиками; так, подход, описанный в разделе 2.4, создает синтаксические анализаторы для LL-грамматик. Синтаксические анализаторы для несколько большего класса LR-грамматик обычно создаются с помощью автоматизированных инструментов.

В этой главе мы полагаем, что выход синтаксического анализатора является некоторым представлением дерева разбора входного потока токенов, выданного лексическим анализатором. На практике имеется множество задач, которые могут сопровождать процесс разбора, — например, сбор информации о различных токенах в таблице символов, выполнение проверки типов и других видов семантического анализа, а также создание промежуточного кода, как описано в главе 2, “Простой однопроходный компилятор”. Все эти задачи представлены одним блоком “Прочие задачи начальной фазы компиляции” на рис. 4.1. Детальнее обсудим их в следующих трех главах.

В оставшейся части этого раздела мы рассмотрим природу синтаксических ошибок и стратегии восстановления после их обнаружения. Две из стратегий, известные как “режим паники” (panic-mode) и восстановление на уровне фразы, более подробно обсуждаются вместе с соответствующими методами разбора. Реализация каждой стратегии определяется автором компилятора, но мы все же дадим несколько советов по этому вопросу.

## Обработка синтаксических ошибок

Если компилятор будет иметь дело исключительно с корректными программами, его разработка и реализация существенно упрощаются. Однако слишком часто программисты пишут программы с ошибками, и хороший компилятор должен помочь программисту обнаружить их и локализовать. Примечательно, что хотя ошибки — явление чрезвы-

чайно распространенное, лишь в нескольких языках вопрос обработки ошибок рассматривался еще на фазе создания языка. Наша цивилизация существенно отличалась бы от своего нынешнего состояния, если бы в естественных языках были такие же требования к синтаксической точности, как и в языках программирования. Большинство спецификаций языков программирования, тем не менее, не определяет реакции компилятора на ошибки — этот вопрос отдается на откуп разработчикам компилятора. Однако планирование системы обработки ошибок с самого начала работы над компилятором может как упростить его структуру, так и улучшить его реакцию на ошибки.

Мы знаем, что любая программа потенциально содержит множество ошибок самого разного уровня. Например, ошибки могут быть

- лексическими, такими как неверно записанные идентификаторы, ключевые слова или операторы;
- синтаксическими, например арифметические выражения с несбалансированными скобками;
- семантическими, такими как операторы, применяемые к несовместимым с ними операндам;
- логическими, например бесконечная рекурсия.

Зачастую основные действия по выявлению ошибок и восстановлению после них концентрируются в фазе синтаксического анализа. Одна из причин этого состоит в том, что многие ошибки по природе своей являются синтаксическими или проявляются, когда поток токенов, идущий от лексического анализатора, нарушает определяющие язык программирования грамматические правила. Вторая причина заключается в точности современных методов разбора; они очень эффективно выявляют синтаксические ошибки в программе. Определение присутствия в программе семантических или логических ошибок — задача неизмеримо более сложная. В этом разделе мы представим несколько основных методов восстановления после синтаксических ошибок; их реализация рассматривается в этой главе вместе с методами разбора.

Обработчик ошибок синтаксического анализатора имеет очень просто формулируемые цели:

- он должен ясно и точно сообщать о наличии ошибок;
- он должен обеспечивать быстрое восстановление после ошибки, чтобы продолжить поиск последующих ошибок;
- он не должен существенно замедлять обработку корректной программы.

Эффективная реализация этих целей представляет собой весьма сложную задачу.

К счастью, обычные ошибки достаточно просты, и для их обработки часто достаточно относительно простых механизмов обработки ошибок. В некоторых случаях, однако, ошибка может произойти задолго до момента ее выявления (и за много строк кода до места ее выявления), и определить ее природу весьма непросто. В сложных ситуациях обработчик ошибок, по сути, должен попросту догадаться, что именно имел в виду программист, когда писал программу.

Некоторые методы разбора, такие как LL и LR, выявляют ошибки, как только это становится возможным. Точнее говоря, они обладают свойством проверки корректности префиксов, т.е. выявляют ошибку, как только выясняется, что префикс входной информации не является префиксом ни одной корректной строки языка.

## Пример 4.1

Чтобы получить представление о видах ошибок, встречающихся на практике, рассмотрим ошибки, найденные Рипли и Друсейкисом [372] в студенческих программах на Pascal.

Они обнаружили, что ошибки не так уж часты — 60% программ были синтаксически и семантически корректны. Даже когда в программах встречались ошибки, они были достаточно разбросаны: 80% ошибочных инструкций имели только по одной ошибке и 13% — по две ошибки. И наконец, большинство ошибок были тривиальны — 90% из них представляли собой ошибки в одном токене.

Многие из ошибок могли быть легко классифицированы: 60% составляли ошибки пунктуации, 20% — ошибки в операторах и операндах, 15% — ошибки в ключевых словах, а оставшиеся 5% — ошибки других типов. Основная часть ошибок пунктуации состояла в некорректном использовании точек с запятой.

В качестве конкретного примера рассмотрим следующую программу на языке программирования Pascal.

```
(1) program prmax(input, output);
(2) var
(3) x, y: integer;
(4) function max(i:integer; j: integer) : integer;
(5) { Возвращает максимальное из чисел i и j }
(6) begin
(7) if i > j then max := i
(8) else max := j
(9) end;
(10) begin
(11) readln (x,y);
(12) writeln (max(x,y))
(13) end.
```

Обычная ошибка пунктуации состоит в использовании запятой вместо точки с запятой в списке аргументов в объявлении функции (т.е. применение запятой вместо первой точки с запятой в строке (4)); другой ошибкой является пропуск обязательной точки с запятой в конце строки (например, в конце строки (4)). Еще одной часто встречающейся ошибкой является запись лишней точки с запятой в конце строки перед `else` (например, в конце строки (7)).

Возможно, столь частое появление “ошибок точки с запятой” обусловлено тем, что использование этого знака сильно различается в разных языках. В Pascal точка с запятой служит разделителем между инструкциями; в PL/I и С этот символ завершает инструкции. Некоторые исследования показывают, что последнее использование точки с запятой порождает меньше ошибок [155].

Типичный пример ошибки в операторе — пропуск двоеточия в операторе присвоения `:=`<sup>1</sup>. Ошибки в ключевых словах встречаются значительно реже. Достаточно репрезентативным примером может служить пропуск символа `i` в ключевом слове `writeln`.

Многие компиляторы Pascal не имеют проблем при обработке обычных ошибок вставки, удаления или замены символов. В действительности ряд компиляторов Pascal корректно скомпилирует приведенную выше программу с обычными ошибками пунк-

<sup>1</sup> В С соответствующая типичная ошибка — пропуск одного знака равенства в операторе эквивалентности `==`. — Прим. ред.

туации или операторов; при этом будут выданы только предупреждающие сообщения, указывающие на ошибочные конструкции.

Однако еще один тип ошибок намного более сложен для корректного исправления. Это — отсутствие `begin` или `end` (например, отсутствие строки (9)). Большинство компиляторов не будут пытаться исправить ошибки такого типа. □

Как именно обработчик ошибок должен сообщить о наличии ошибки? Как минимум — сообщить о месте в исходной программе, где была выявлена ошибка, так как на самом деле ошибка находится, как правило, в пределах нескольких предыдущих токенов. Общая стратегия, используемая многими компиляторами, заключается в выводе ошибочной строки с указанием позиции, в которой обнаружена ошибка. Если имеется обоснованное предположение о природе ошибки, вывод включает диагностическое пояснение типа “отсутствие точки с запятой в данной позиции”.

После того как ошибка выявлена, в чем же должно состоять восстановление синтаксического анализатора? Как вы увидите, имеется ряд общепринятых стратегий, но ни один из методов восстановления не является доминирующим. В большинстве случаев окончание анализа после выявления первой ошибки не является хорошим решением, так как восстановление и продолжение анализа могло бы выявить ряд последующих ошибок. Обычно восстановление после ошибки заключается в том, что синтаксический анализатор пытается восстановить некоторое свое состояние, в котором продолжается обработка входного потока так, как если бы он был корректным.

Неадекватное восстановление после ошибки может привести к настоящей лавине ложных ошибок, которые не были допущены программистом, а появились вследствие изменений состояния синтаксического анализатора в процессе восстановления после ошибки. Аналогично восстановление после синтаксической ошибки может привести к ложным семантическим ошибкам, которые выявятся позже, на стадии семантического анализа или генерации кода. Например, при восстановлении после ошибки синтаксический анализатор может пропустить объявление некоторой переменной, скажем `зар`. Когда позже `зар` появится в выражении, синтаксической ошибки не будет, но поскольку в таблице символов не будет записи для этой переменной, компилятор выдаст сообщение об ошибке “`зар` не определено”.

Консервативная стратегия компилятора состоит в запрете сообщений об ошибках, которые возникают во входном потоке слишком близко. После обнаружения одной синтаксической ошибки компилятор должен успешно проанализировать несколько входных токенов и только после этого разрешить вывод других сообщений об ошибках. В некоторых случаях ошибок может оказаться слишком много, чтобы имело смысл продолжать работу (например, как должен реагировать компилятор Pascal при попытке скомпилировать программу на языке Fortran?). Похоже, стратегия восстановления после ошибок должна представлять собой разумный компромисс, принимающий во внимание количество и типы ошибок, которые можно обнаружить и обработать.

Как мы упоминали, некоторые компиляторы пытаются исправить ошибки, т.е. догадаться, что программист собирался написать. Примером такого компилятора может служить PL/C (Конвей и Вилкокс [95]). За исключением, пожалуй, коротких программ, написанных начинающими студентами, исправление ошибок, вероятно, не слишком эффективно. В действительности, с возрастанием роли интерактивности и эффективного программного окружения имеется тенденция к использованию простых механизмов восстановления после ошибок.

## Стратегии восстановления после ошибок

Существуют различные стратегии, которые синтаксический анализатор может использовать для восстановления после синтаксической ошибки. Хотя ни одна из стратегий не является универсальной, несколько методов используются весьма широко. Здесь мы вкратце познакомимся со следующими стратегиями:

- режим паники;
- уровень фразы;
- продукции ошибок;
- глобальная коррекция.

*Режим паники.* Эта стратегия проще всего реализуется и может использоваться большинством методов синтаксического анализа. При обнаружении ошибки синтаксический анализатор пропускает входные символы по одному, пока не будет найден один из специально определенного множества синхронизирующих токенов. Обычно такими токенами являются разделители, например `end` или точка с запятой, роль которых в исходной программе совершенно очевидна. Разработчик компилятора, естественно, должен выбирать синхронизирующие токены исходя из особенностей исходного языка. Хотя такая коррекция часто пропускает значительное количество символов без проверки на наличие ошибок, она достаточно проста и, в отличие от некоторых методов, рассматриваемых позже, гарантирует отсутствие зацикливания. В случаях, когда несколько ошибок в одной инструкции встречаются очень редко, этот метод работает вполне адекватно.

*Уровень фразы.* При обнаружении ошибки синтаксический анализатор может выполнить локальную коррекцию оставшегося входного потока. Таким образом, он может заменить префикс остальной части потока некоторой строкой, которая позволит синтаксическому анализатору продолжить работу. Типичная локальная коррекция может состоять в замене запятой точкой с запятой, удалении лишней точки с запятой или вставке недостающей. Выбор способа локальной коррекции — прерогатива разработчика компилятора. Естественно, мы должны быть предельно осторожны при выборе способа коррекции, чтобы он не привел, например, к бесконечному циклу (что возможно при вставке некоторого символа перед текущим сканируемым).

Этот тип замещения может скорректировать любую входную строку и используется в ряде компиляторов, исправляющих ошибки. Этот метод первоначально применялся совместно с нисходящим разбором. Основной его недостаток заключается в сложности обработки ситуации, когда реально ошибка располагается до точки обнаружения.

*Продукции ошибок.* Если мы знаем наиболее распространенные ошибки, можно расширить грамматику языка продукциями, порождающими ошибочные конструкции. Затем мы строим синтаксический анализатор на основе такой расширенной грамматики языка. Если синтаксический анализатор использует одну из “ошибочных” производств, мы можем генерировать корректное диагностическое сообщение для распознанной синтаксическим анализатором ошибки.

*Глобальная коррекция.* В идеале при обработке некорректной входной строки компилятор должен вносить минимально возможное количество изменений. Существует ряд алгоритмов, которые позволяют выбрать минимальную последовательность вносимых изменений для проведения коррекции. По заданной строке  $x$  и грамматике  $G$  такие алгоритмы строят дерево разбора для строки  $y$ , такой, что количество вставок, удалений и изменений токенов, требуемых для преобразования  $x$  в  $y$ , — минимально возможное. К

сожалению, в целом эти методы слишком невыгодны для применения в компиляторах в силу высоких требований к памяти и времени работы, а потому представляют в основном теоретический интерес.

Мы должны отметить, что ближайшая к исходной исправленная программа может оказаться вовсе не тем, что хотел получить программист. Тем не менее понятие минимальной коррекции дает нам критерий оценки качества различных технологий восстановления после ошибок и используется для поиска оптимальной замены строки при восстановлении на уровне фразы.

## 4.2. Контекстно-свободные грамматики

Многие языки программирования по своей природе имеют рекурсивную структуру, которая может определяться контекстно-свободными грамматиками. Например, у нас может быть условная инструкция, определяемая следующим правилом.

Если  $S_1$  и  $S_2$  являются инструкциями, а  $E$  — выражением, то

$$\text{if } E \text{ then } S_1 \text{ else } S_2 \quad (4.1)$$

является инструкцией

Этот тип условных инструкций не может быть определен с использованием регулярных выражений. В главе 3, “Лексический анализ”, мы видели, что регулярные выражения могут определять лексическую структуру токенов. Однако, используя для обозначения класса инструкций синтаксическую переменную *stmt*, а для класса выражений — *expr*, можем легко выразить (4.1) с помощью продукции грамматики

$$\text{stmt} \rightarrow \text{if } expr \text{ then } stmt \text{ else } stmt \quad (4.2)$$

В этом разделе рассмотрим определение контекстно-свободной грамматики и познакомимся с терминологией, используемой при описании процесса синтаксического разбора. Из раздела 2.2 мы знаем, что контекстно-свободная грамматика (или, для краткости, просто грамматика) состоит из терминалов, нетерминалов, стартового символа и продукции.

1. Терминалы представляют собой базовые символы, из которых формируются строки. Слово “токен” является синонимом слова “терминал”, когда мы говорим о грамматиках языков программирования. В (4.2) каждое из ключевых слов *if*, *then*, *else* является терминалом.
2. Нетерминалы представляют собой синтаксические переменные, которые обозначают множества строк. В (4.2) *stmt* и *expr* являются нетерминалами. Нетерминалы определяют множества строк, которые помогают в определении языка, порождаемого грамматикой. Кроме того, они налагают на язык иерархическую структуру, облегчающую синтаксический анализ и трансляцию.
3. Один из нетерминалов грамматики считается стартовым символом, и множество строк, которые он обозначает, является языком, определяемым грамматикой.
4. Продукции грамматики определяют способ, которым терминалы и нетерминалы могут объединяться для создания строк. Каждая продукция состоит из нетерминала, за которым следует стрелка (или символ  $::=$ ), и строки нетерминалов и терминалов.

## Пример 4.2

Грамматика со следующими продукциями определяет простые арифметические выражения.

$$\begin{array}{l} \textit{expr} \rightarrow \textit{expr op expr} \\ \textit{expr} \rightarrow ( \textit{expr} ) \\ \textit{expr} \rightarrow - \textit{expr} \\ \textit{expr} \rightarrow \textit{id} \\ \textit{op} \rightarrow + \\ \textit{op} \rightarrow - \\ \textit{op} \rightarrow * \\ \textit{op} \rightarrow / \\ \textit{op} \rightarrow \uparrow \end{array}$$

В этой грамматике терминальными символами являются

$\textit{id} + - * / \uparrow ( )$

Нетерминальные символы грамматики —  $\textit{expr}$  и  $\textit{op}$ ; стартовым символом грамматики является  $\textit{expr}$ .  $\square$

## Соглашения по обозначениям

Для того чтобы избежать постоянного упоминания “это — терминал”, “это — нетерминал” и т.п., будем использовать следующие соглашения по обозначениям при записи грамматик во всей книге.

1. Эти символы являются терминалами:
  - i) строчные буквы из начала алфавита, такие как  $a, b, c$ ;
  - ii) символы операторов, такие как  $+, -, *, /$  и т.п.;
  - iii) символы пунктуации, такие как запятые, скобки и т.п.;
  - iv) цифры  $0, 1, \dots, 9$ ;
  - v) строки, выделенные полужирным шрифтом, такие как  $\textit{id}$  или  $\textit{if}$ .
2. Эти символы являются нетерминалами:
  - i) прописные буквы из начала алфавита, такие как  $A, B, C$ ;
  - ii) буква  $S$ , которая обычно означает стартовый символ;
  - iii) имена из строчных букв, выделенные курсивом, такие как  $\textit{stmt}$  или  $\textit{expr}$ .
3. Прописные буквы из конца алфавита, такие как  $X, Y, Z$ , представляют *грамматические символы*, т.е. либо терминалы, либо нетерминалы.
4. Строчные буквы из конца алфавита, такие как  $u, v, \dots, z$ , обозначают строки терминалов.
5. Строчные греческие буквы, такие как  $\alpha, \beta, \gamma$ , представляют строки грамматических символов. Таким образом, в общем виде продукция может быть записана как  $A \rightarrow \alpha$ , в которой одиночный нетерминал  $A$  располагается слева от стрелки (в *левой части* продукции), а строка грамматических символов  $\alpha$  — справа от стрелки (в *правой части* продукции).

6. Если  $A \rightarrow \alpha_1, A \rightarrow \alpha_2, \dots, A \rightarrow \alpha_k$  представляют собой продукцию с  $A$  в левой части (мы называем их *A-продукциями*), то можем записать  $A \rightarrow \alpha_1|\alpha_2|...|\alpha_k$ . Мы называем  $\alpha_1, \alpha_2, \dots, \alpha_k$  *альтернативами A*.
7. Если иное не указано явно, левая часть первой продукции является стартовым символом.

### Пример 4.3

Используя приведенные соглашения, можно записать грамматику из примера 4.2 как

$$\begin{aligned} E &\rightarrow EA\,E \mid (E) \mid -E \mid \text{id} \\ A &\rightarrow + \mid - \mid * \mid / \mid \uparrow \end{aligned}$$

Из наших соглашений следует, что  $E$  и  $A$  — нетерминалы, причем  $E$  является стартовым символом. Остальные символы представляют собой терминалы.  $\square$

### Порождение

Существует несколько способов рассматривать процесс определения языка грамматикой. В разделе 2.2 мы видели его как процесс построения деревьев разбора. Однако имеется и другой полезный способ, который дает точное описание нисходящего построения дерева разбора. Основная идея состоит в том, что продукция рассматривается как переписывающее правило, в котором нетерминал из левой части замещается строкой из правой части продукции.

Рассмотрим, например, следующую грамматику арифметических выражений с нетерминалом  $E$ , представляющим выражение.

$$E \rightarrow E + E \mid E * E \mid (E) \mid -E \mid \text{id} \quad (4.3)$$

Продукция  $E \rightarrow -E$  означает, что выражение, которому предшествует знак минуса, также является выражением. Эта продукция может использоваться для порождения<sup>2</sup> более сложных выражений из простых, позволяя нам заменять любой экземпляр  $E$  на  $-E$ . В простейшем случае мы можем заместить одно  $E$  на  $-E$  и описать это как  $E \Rightarrow -E$ , что можно прочесть как “ $E$  порождает  $-E$ ”, или “из  $E$  выводится  $-E$ ”. Продукция  $E \rightarrow (E)$  говорит, что мы можем также заменить один экземпляр  $E$  в любой строке грамматики на  $(E)$ , например  $E^*E \Rightarrow (E)^*E$  или  $E^*E \Rightarrow E^*(E)$ .

Мы можем взять нетерминал  $E$  и неоднократно применять продукцию в произвольном порядке для получения последовательности замещений, например,  $E \Rightarrow -E \Rightarrow -(E) \Rightarrow -(id)$ . Такую последовательность замен назовем *выводом* (derivation), или *порождением*  $-(id)$  из  $E$ . Такой вывод является доказательством того, что одним из примеров выражения является строка  $-(id)$ .

Более абстрактно, мы говорим, что  $\alpha A \beta \Rightarrow \alpha \gamma \beta$ , если  $A \rightarrow \gamma$  является продукцией, а  $\alpha$  и  $\beta$  представляют собой произвольные строки грамматических символов. Если  $\alpha_1 \Rightarrow \alpha_2 \Rightarrow \dots \Rightarrow \alpha_n$ , говорим, что  $\alpha_1$  *порождает*  $\alpha_n$ . Символ  $\Rightarrow$  означает “порождает за

---

<sup>2</sup> В оригинале — derivations (выводы). В связи с неоднозначностью общепринятого в русскоязычной литературе термина “вывод” далее используется термин “порождение”, отражающий порождение форм из нетерминалов — *Прим. перев.*

один шаг". Часто мы говорим "порождает за нуль или более шагов" и обозначаем это символом  $\Rightarrow^*$ . Таким образом,

- 1)  $\alpha \Rightarrow^* \alpha$  для любой строки  $\alpha$ ,
- 2) если  $\alpha \Rightarrow^* \beta$  и  $\beta \Rightarrow^* \gamma$ , то  $\alpha \Rightarrow^* \gamma$ .

Аналогично символ  $\Rightarrow^+$  используется для обозначения "порождает за один или более шагов".

Имея грамматику  $G$  со стартовым символом  $S$ , можно использовать отношение  $\Rightarrow^+$  для определения  $L(G)$ , языка, порожденного грамматикой  $G$ . Строки в  $L(G)$  могут содержать только терминальные символы из  $G$ , т.е. строка терминалов  $w$  принадлежит  $L(G)$  тогда и только тогда, когда  $S \Rightarrow^+ w$ . Стока  $w$  при этом называется *предложением G*, а язык, который может быть порожден грамматикой, — *контекстно-свободным*. Если две грамматики порождают один и тот же язык, то эти грамматики называются *эквивалентными*.

Если  $S \Rightarrow^* \alpha$ , где  $\alpha$  может содержать нетерминалы, то мы говорим о том, что  $\alpha$  является *сентенциальной формой*<sup>3</sup>  $G$ . Предложение представляет собой сентенциальную форму без нетерминалов.

#### Пример 4.4

Стока  $-(\text{id}+\text{id})$  является предложением грамматики (4.3) ввиду наличия порождения  $E \Rightarrow -E \Rightarrow -(E) \Rightarrow -(E+E) \Rightarrow -(\text{id}+E) \Rightarrow -(\text{id}+\text{id})$  (4.4)

Строки  $E$ ,  $-E$ ,  $-(E)$ , ...,  $-(\text{id}+\text{id})$ , появляющиеся в процессе порождения, представляют собой сентенциальные формы данной грамматики. Для указания того, что  $-(\text{id}+\text{id})$  может быть выведено из  $E$ , мы записываем  $E \Rightarrow^* -(\text{id}+\text{id})$ .

С помощью индукции по длине порождения можно показать, что каждое предложение в языке грамматики (4.3) представляет собой арифметическое выражение, включающее бинарные операторы  $*$  и  $+$ , унарный оператор  $-$ , скобки и operand  $\text{id}$ . Аналогично с помощью индукции по длине арифметического выражения можно показать, что все такие выражения могут быть порождены данной грамматикой. Следовательно, грамматика (4.3) генерирует в точности множество всех арифметических выражений, которые включают бинарные операторы  $*$  и  $+$ , унарный оператор  $-$ , скобки и operand  $\text{id}$ .  $\square$

На каждом шаге порождения осуществляется два выбора — мы должны выбрать заменяемый нетерминал, а затем его альтернативу. Например, порождение (4.4) из примера 4.4 может продолжиться после  $-(E+E)$  следующим образом:

$-(E+E) \Rightarrow -(\text{id}+\text{id}) \Rightarrow -(\text{id}+\text{id})$  (4.5)

Каждый нетерминал в (4.5) замещается той же правой частью, что и в (4.4), но в другом порядке.

Чтобы понять, как работают некоторые синтаксические анализаторы, необходимо рассмотреть порождения, в каждой сентенциальной форме которых замещается самый левый нетерминал (такие порождения называются *левыми* (*leftmost*)<sup>4</sup>). Если  $\alpha \Rightarrow \beta$  пред-

<sup>3</sup> В русскоязычной литературе также используется термин "цепочка": цепочка терминалов и нетерминалов. — Прим. ред.

<sup>4</sup> Термину "левое порождение" соответствует используемый иногда термин "левосторонний вывод". — Прим. ред.

ставляет собой порождение, в котором замещается крайний слева нетерминал в  $\alpha$ , мы записываем  $\alpha \xrightarrow{lm} \beta$ . Поскольку порождение (4.4) является левым, можем записать его как

$$\begin{array}{c} E \Rightarrow - \\ lm \end{array} E \Rightarrow - \begin{array}{c} (E) \Rightarrow - \\ lm \end{array} (E + E) \Rightarrow - \begin{array}{c} (\text{id} + E) \Rightarrow - \\ lm \end{array} (\text{id} + \text{id})$$

Используя введенные нами обозначения, каждый шаг левого порождения можно записать как  $wA\gamma \xrightarrow{lm} w\delta\gamma$ , где  $w$  состоит только из терминалов,  $A \rightarrow \delta$  представляет собой использованную в шаге продукцию, а  $\gamma$  — строку символов грамматики. Для подчеркивания того факта, что  $\alpha$  порождает  $\beta$  левым порождением, мы записываем  $\alpha \xrightarrow{lm} \beta$ . Если  $S \xrightarrow{*}_{lm} \alpha$ , то говорим, что  $\alpha$  является левосентенциальной формой рассматриваемой грамматики.

Аналогичные определения существуют для *правых* (rightmost) порождений, в которых на каждом шаге замещается крайний справа нетерминал. Иногда правые приведения называются *каноническими*.

## Деревья разбора и приведения

Дерево разбора может рассматриваться как графическое представление порождения, из которого удалена информация о порядке замещения. Вспомним из раздела 2.2, что каждый внутренний узел дерева разбора помечается некоторым нетерминалом  $A$ , а дочерние узлы слева направо — символами из правой части продукции, использованной в порождении для замены  $A$ . Листья дерева разбора помечены нетерминалами или терминалами и, будучи прочитаны слева направо, образуют сентенциальную форму, называемую кроной, или границей дерева. Например, дерево разбора для  $-(\text{id}+\text{id})$ , полученное порождением (4.4), показано на рис. 4.2.

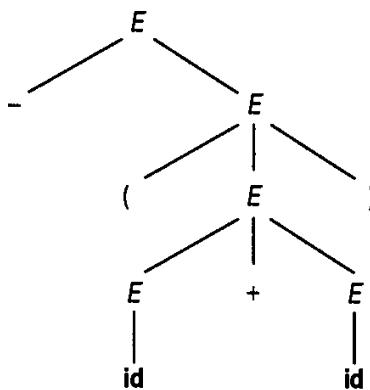


Рис. 4.2. Дерево разбора для  $-(\text{id}+\text{id})$

Для того чтобы увидеть взаимосвязь между порождением и деревьями разбора, рассмотрим любое приведение  $\alpha_1 \Rightarrow \alpha_2 \Rightarrow \dots \Rightarrow \alpha_n$ , где  $\alpha_i$  — отдельный нетерминал  $A$ . Для каждой сентенциальной формы  $\alpha_i$  в приведении строим дерево разбора, результатом которого является  $\alpha_i$ . Этот процесс представляет собой индукцию по  $i$ . Базисом служит дерево для  $\alpha_1 = A$ , которое представляет собой единственный узел, помеченный как  $A$ . Для выполнения индукции предположим, что мы уже построили дерево разбора, имеющее крону  $\alpha_{i-1} = X_1 X_2 \dots X_k$  (вспомним, что  $X_j$  может означать терминал или нетерминал). Предположим, что  $\alpha_{i-1}$  порождает  $\alpha_i$  заменой нетерминала  $X_j$  на  $\beta = Y_1 Y_2 \dots Y_r$ . Таким об-

разом, на  $i$ -м шаге порождения к  $\alpha_{i-1}$  применяется продукция  $X_j \rightarrow \beta$ , порождая  $\alpha_i = X_1 X_2 \dots X_{j-1} \beta X_{j+1} \dots X_k$ .

Для моделирования этого шага находим  $j$ -й слева лист в текущем дереве разбора, который помечен  $X_j$ . Мы даем этому листу  $r$  дочерних узлов, помеченных  $Y_1, Y_2, \dots, Y_r$  слева направо. В специальном случае  $r = 0$ , т.е.  $\beta = \epsilon$ , у  $j$ -го листа появляется один дочерний узел, помеченный как  $\epsilon$ .

#### Пример 4.5

Рассмотрим порождение (4.4). Последовательность деревьев разбора, построенная на его основе, показана на рис. 4.3. Первый шаг этого порождения —  $E \Rightarrow -E$ . Для моделирования этого шага мы добавляем к корню начального дерева два дочерних узла, помеченных как “ $-$ ” и “ $E$ ”.

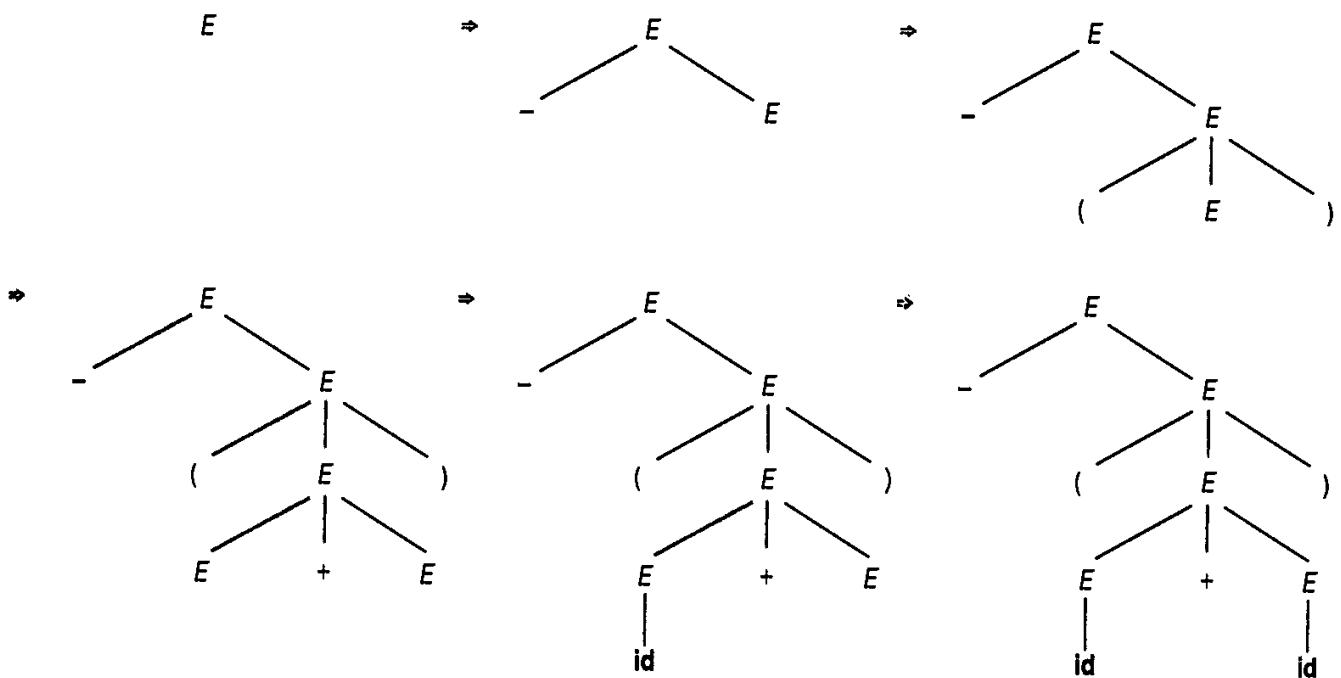


Рис. 4.3. Построение дерева разбора из порождения (4.4)

Второй шаг представляет собой  $-E \Rightarrow -(E)$ . Соответственно, мы добавляем три дочерних узла — “(”, “ $E$ ” и “)” — к листу  $E$  во втором дереве для получения третьего дерева, дающего  $-(E)$ . Продолжая построения, мы получим шестое дерево в качестве полного дерева разбора.  $\square$

Как упоминалось, дерево разбора игнорирует порядок, в котором производилось замещение символов в сентенциальной форме. Например, если порождение (4.4) изменить в соответствии с (4.5), окончательное дерево разбора будет таким же, как на рис. 4.3. Вариации порядка использования продукции можно избежать, рассматривая только левые (или правые) порождения. Нетрудно увидеть, что каждому дереву разбора соответствуют единственное левое и единственное правое порождения — однако следует отдавать себе отчет, что предложение может иметь не одно дерево разбора и даже не одно левое или правое порождение.

### Пример 4.6

Обратимся вновь к грамматике арифметических выражений (4.3). Предложение  $id + id * id$  имеет два разных левых порождения

$$\begin{aligned} E &\Rightarrow E + E \\ &\Rightarrow id + E \\ &\Rightarrow id + E * E \\ &\Rightarrow id + id * E \\ &\Rightarrow id + id * id \end{aligned}$$

$$\begin{aligned} E &\Rightarrow E * E \\ &\Rightarrow E + E * E \\ &\Rightarrow id + E * E \\ &\Rightarrow id + id * E \\ &\Rightarrow id + id * id \end{aligned}$$

с двумя соответствующими им деревьями разбора, показанными на рис. 4.4. □

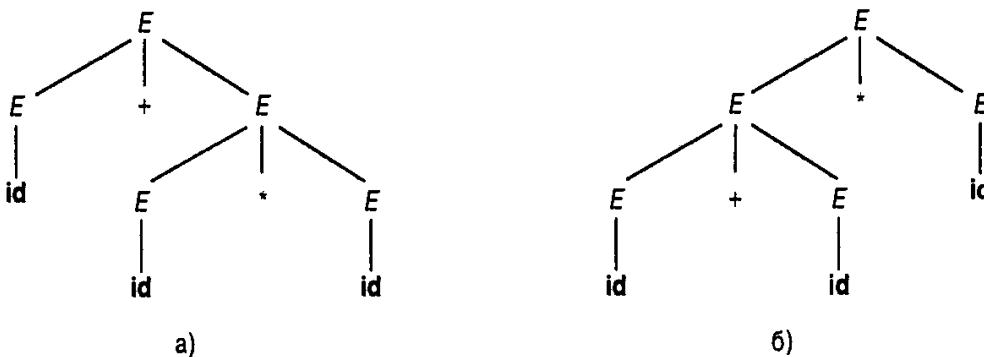


Рис. 4.4. Два дерева разбора для  $id + id * id$

Обратите внимание, что дерево разбора на рис. 4.4a отражает обычные приоритеты операций  $+$  и  $*$ , в отличие от дерева на рис. 4.4b. Обычно приоритет умножения выше, и выражение типа  $a+b*c$  трактуется как  $a+(b*c)$ , а не как  $(a+b)*c$ .

### Неоднозначность

Грамматика, которая дает более одного дерева разбора для некоторого предложения, называется *неоднозначной* (*ambiguous*). Иначе говоря, неоднозначная грамматика — это та, которая для одного и того же предложения дает не менее двух левых или правых порождений. Для большинства типов синтаксических анализаторов грамматика должна быть однозначной, поскольку, если это не так, мы не в состоянии определить дерево разбора для предложения единственным образом. Для некоторых приложений мы рассмотрим методы, допускающие использование ряда неоднозначных грамматик совместно с *правилами устранения неоднозначностей* (*disambiguating rules*), которые “отбрасывают” нежелательные деревья разбора, оставляя для каждого предложения по одному дереву.

## 4.3. Разработка грамматики

Грамматикой можно описать большую часть синтаксиса (но не весь) языков программирования. Некоторая часть синтаксического анализа выполняется лексическим анализатором в процессе получения последовательности токенов из входного потока символов. Некоторые ограничения, налагаемые на входной поток (например, требование объявления идентификаторов до их использования), не могут быть описаны в рамках контекстно-свободной грамматики. Следовательно, последовательности токенов, допускаемые синтаксическим анализатором, образуют надмножество языка программирования; последующие фазы компиляции должны анализировать выход синтаксического ана-

лизатора на его соответствие правилам, которые не проверяются этим анализатором (см. главу 6, “Проверка типов”).

Мы начнем этот раздел с рассмотрения разделения работы между лексическим и синтаксическим анализаторами. Поскольку каждый метод разбора в состоянии работать с грамматикой определенного типа, изначальная грамматика, возможно, должна быть переписана, чтобы удовлетворять требованиям выбранного метода разбора. Подходящая грамматика для выражений зачастую может быть построена с использованием информации об ассоциативности и приоритете операций, о чем говорилось в разделе 2.2. В этом разделе рассмотрим преобразования, помогающие записать грамматику в виде, пригодном для нисходящего разбора. Завершится этот раздел рассмотрением некоторых конструкций языков программирования, которые не могут быть описаны ни одной грамматикой.

## Регулярные выражения и контекстно-свободные грамматики

Каждая конструкция, которая может быть описана регулярным выражением, может быть описана и грамматикой. Например, регулярное выражение  $(a|b)^*abb$  и грамматика

$$\begin{array}{lcl} A_0 & \rightarrow & aA_0 \mid bA_0 \mid aA_1 \\ A_1 & \rightarrow & bA_2 \\ A_2 & \rightarrow & bA_3 \\ A_3 & \rightarrow & \epsilon \end{array}$$

описывают один и тот же язык, множество строк из  $a$  и  $b$ , заканчивающихся на  $abb$ .

Мы можем механически преобразовать недетерминированный конечный автомат (НКА) в грамматику, порождающую тот же язык, что и распознаваемый НКА. Приведенная выше грамматика построена на основе НКА, показанного на рис. 3.23, следующим образом. Для каждого состояния  $i$  в НКА создается нетерминальный символ  $A_i$ . Если состояние  $i$  имеет переход в состояние  $j$  по символу  $a$ , вводим в грамматику соответствующую продукцию  $A_i \rightarrow aA_j$  (если состояние  $i$  переходит в состояние  $j$  для входа  $\epsilon$ , мы добавляем продукцию  $A_i \rightarrow A_j$ ). Для заключительного состояния  $i$  вводим продукцию  $A_i \rightarrow \epsilon$ ; если же  $i$  — начальное состояние, то  $A_i$  становится стартовым символом грамматики.

Поскольку каждое регулярное множество является контекстно-свободным языком, вы можете вполне резонно спросить: “Зачем же использовать регулярные выражения для определения лексической структуры языка?”. Тому есть несколько причин.

1. Лексические правила языка зачастую достаточно просты, и для их описания не нужен такой мощный инструмент, как грамматика.
2. Регулярные выражения обычно представляют собой более простую и выразительную запись токенов, чем грамматика.
3. На основе регулярных выражений можно автоматически построить более эффективные лексические анализаторы, чем на основе грамматики.
4. Разделение синтаксической структуры языка на лексическую и нелексическую части обеспечивает разделение начальной стадии компилятора на модули, что повышает управляемость проектом.

Не существует четких правил относительно того, что следует размещать в лексической части, а что — в синтаксической. Регулярные выражения наиболее пригодны для

описания структуры лексических конструкций, таких как идентификаторы, константы, ключевые слова и т.п. Грамматики же более применимы для описания вложенных структур типа сбалансированных скобок, `begin-end`, условных операторов `if-then-else` и т.п. Как уже упоминалось, эти структуры не могут быть описаны регулярными выражениями.

## Проверка языка, порожденного грамматикой

Хотя разработчики компиляторов редко делают это для полной грамматики языка программирования, очень важно иметь возможность убедиться, что данное множество продукции порождает определенный язык. Вызывающие головную боль конструкции можно изучить, написав сокращенный вариант грамматики и изучив язык, порождаемый ею. Ниже мы построим такую грамматику для условных операторов.

Для доказательства того, что грамматика  $G$  порождает язык  $L$ , мы должны показать, что любая строка, генерируемая  $G$ , принадлежит  $L$ , а каждая строка из  $L$  может быть порождена грамматикой  $G$ .

### Пример 4.7

Рассмотрим грамматику

$$S \rightarrow (S) S \mid \epsilon \quad (4.6)$$

Хотя на первый взгляд этого не видно, данная простая грамматика порождает все строки из сбалансированных скобок, и только такие строки. Вначале покажем, что каждое предложение, выводимое из  $S$ , сбалансировано, а затем — что каждая сбалансированная строка может быть выведена из  $S$ . Для того чтобы показать, что каждое предложение, выводимое из  $S$ , сбалансировано, воспользуемся индукцией по количеству шагов в порождении. Для доказательства базы индукции заметим, что за один шаг из  $S$  выводится только пустая строка (которая, само собой, является сбалансированной).

Теперь предположим, что все порождения из менее чем  $n$  шагов приводят к сбалансированным предложениям, и рассмотрим левое порождение, состоящее ровно из  $n$  шагов. Такое порождение должно иметь вид  $S \Rightarrow (S)S \xrightarrow{*}(x)S \xrightarrow{*}(x)y$ . Порождения  $x$  и  $y$  занимают менее  $n$  шагов, и, согласно индуктивной гипотезе,  $x$  и  $y$  сбалансированы. Следовательно, строка  $(x)y$  должна быть сбалансированной.

Таким образом, любая строка, выводимая из  $S$ , сбалансирована. Теперь нужно показать, что любая сбалансированная строка может быть получена из  $S$ . Для этого воспользуемся индукцией по длине строки. База индукции очевидна, поскольку пустая строка выводима из  $S$ .

Теперь предположим, что любая сбалансированная строка длиной менее  $2n$  выводима из  $S$ , и рассмотрим сбалансированную строку  $w$  длиной  $2n$ , где  $n \geq 1$ . Несомненно, что  $w$  начинается с левой скобки. Пусть  $(x)$  — кратчайший префикс  $w$ , имеющий одинаковое количество левых и правых скобок. Тогда  $w$  можно записать как  $(x)y$ , где  $x$  и  $y$  сбалансированы. Поскольку  $x$  и  $y$  имеют длину менее  $2n$ , они выводимы из  $S$  согласно гипотезе индукции. Таким образом, мы можем найти порождение вида  $S \Rightarrow (S)S \xrightarrow{*}(x)S \xrightarrow{*}(x)y$ , доказывающее, что  $w = (x)y$  также выводится из  $S$ .  $\square$

## Устранение неоднозначности

Зачастую для устранения неоднозначности грамматика может быть переписана. В качестве примера устраним неоднозначность из следующей грамматики:

$$\begin{array}{lcl}
 \text{stmt} & \rightarrow & \text{if } \text{expr} \text{ then } \text{stmt} \\
 & | & \text{if } \text{expr} \text{ then } \text{stmt} \text{ else } \text{stmt} \\
 & | & \text{other}
 \end{array} \tag{4.7}$$

Здесь “other” означает все прочие инструкции. Согласно этой грамматике составная условная инструкция

$\text{if } E_1 \text{ then } S_1 \text{ else if } E_2 \text{ then } S_2 \text{ else } S_3$

имеет дерево разбора, показанное на рис. 4.5. Грамматика (4.7) неоднозначна, поскольку строка

$\text{if } E_1 \text{ then if } E_2 \text{ then } S_1 \text{ else } S_2$

(4.8)

имеет два дерева разбора, показанных на рис. 4.6.

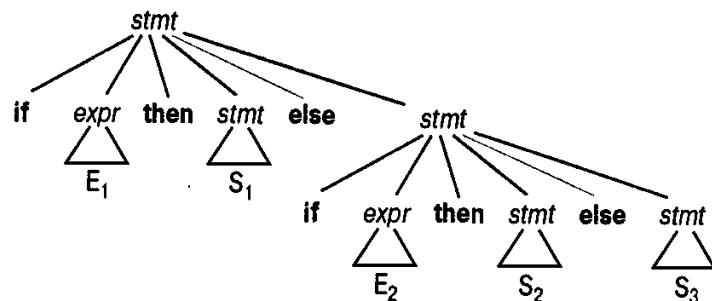


Рис. 4.5. Дерево разбора для условной инструкции

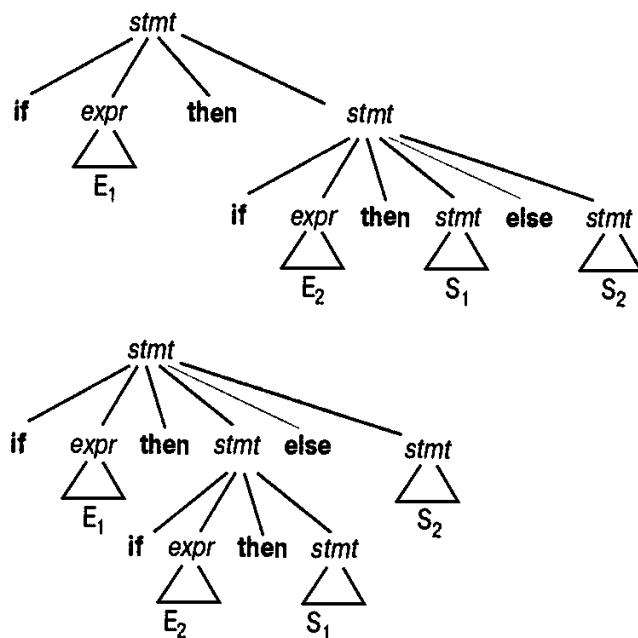


Рис. 4.6. Два дерева разбора для неоднозначного предложения

Во всех языках программирования с условными инструкциями такого вида предпочтительно первое дерево разбора. Общее правило гласит “сопоставить каждое `else` ближайшему незанятыму `then`”. Это правило устранения неоднозначности может быть встроено непосредственно в грамматику. Например, мы можем переписать грамматику (4.7) как однозначную. Основная идея заключается в том, что инструкция, появляющаяся между `then` и `else`, должна быть “сбалансированная” (matched), т.е. не должна оканчиваться на `then`, не соответствующее некоторому `else`. Такая “сбалансированная” инструкция

ция может представлять собой либо полную инструкцию **if-then-else**, содержащую только “соответствующие” инструкции, либо любые инструкции, отличающиеся от условной, т.е. мы получаем грамматику

$$\begin{array}{lcl}
 \textit{stmt} & \rightarrow & \textit{matched_stmt} \\
 & | & \textit{unmatched_stmt} \\
 \textit{matched_stmt} & \rightarrow & \text{if } \textit{expr} \text{ then } \textit{matched_stmt} \text{ else } \textit{matched_stmt} \\
 & | & \text{other} \\
 \textit{unmatched_stmt} & \rightarrow & \text{if } \textit{expr} \text{ then } \textit{stmt} \\
 & | & \text{if } \textit{expr} \text{ then } \textit{matched_stmt} \text{ else } \textit{unmatched_stmt}
 \end{array} \tag{4.9}$$

Эта грамматика генерирует то же множество строк, что и грамматика (4.7), но для строки (4.8) дает только одно дерево разбора, а именно то, которое связывает каждое **else** с ближайшим незанятым **then**.

## Устранение левой рекурсии

Грамматика является **леворекурсивной**, если в ней имеется нетерминал  $A$ , такой, что существует порождение  $A \xrightarrow{+} A\alpha$  для некоторой строки  $\alpha$ . Методы нисходящего разбора не в состоянии работать с леворекурсивными грамматиками, поэтому требуется преобразование грамматики, которое устранило бы из нее левую рекурсию. В разделе 2.4 мы рассматривали простейшую левую рекурсию, где имелась только одна продукция вида  $A \rightarrow A\alpha$ . Сейчас же рассмотрим общий случай. В разделе 2.4 показано, как леворекурсивная пара продуктов  $A \rightarrow A\alpha|\beta$  может быть заменена нелеворекурсивными продукциями

$$\begin{array}{lcl}
 A & \rightarrow & \beta A' \\
 A' & \rightarrow & \alpha A' |\epsilon
 \end{array}$$

без изменения множества строк, порождаемых из  $A$ . Этого правила достаточно для многих грамматик.

### Пример 4.8

Рассмотрим следующую грамматику для арифметических выражений.

$$\begin{array}{lcl}
 E & \rightarrow & E+T \mid T \\
 T & \rightarrow & T^*F \mid F \\
 F & \rightarrow & (E) \mid \text{id}
 \end{array} \tag{4.10}$$

Устранив **непосредственную левую рекурсию** (продукции вида  $A \rightarrow A\alpha$ ) из продуктов для  $E$  и  $T$ , мы получим

$$\begin{array}{lcl}
 E & \rightarrow & TE' \\
 E' & \rightarrow & +TE' \mid \epsilon \\
 T & \rightarrow & FT' \\
 T' & \rightarrow & *FT' \mid \epsilon \\
 F & \rightarrow & (E) \mid \text{id}
 \end{array} \tag{4.11}$$

Не важно, каково общее количество  $A$ -продукций. Мы можем устраниТЬ непосредственную левую рекурсию из них с помощью следующей технологии. Вначале группируем  $A$ -продукции

$$A \rightarrow A\alpha_1 | A\alpha_2 | \dots | A\alpha_m | \beta_1 | \beta_2 | \dots | \beta_n$$

где  $\beta_i$  не начинаются с  $A$ . Затем заменим эти  $A$ -продукции на

$$\begin{aligned} A &\rightarrow \beta_1 A' | \beta_2 A' | \dots | \beta_n A' \\ A' &\rightarrow \alpha_1 A' | \alpha_2 A' | \dots | \alpha_m A' | \epsilon \end{aligned}$$

Нетерминал  $A$  порождает те же строки, что и ранее, но без левой рекурсии. Эта процедура устраняет все непосредственные левые рекурсии из продукции для  $A$  и  $A'$  (при условии, что ни одна строка  $\alpha_i$  не является  $\epsilon$ ), но не устраивает левую рекурсию, вызванную двумя или более шагами порождения. Рассмотрим, например, грамматику

$$\begin{aligned} S &\rightarrow Aa | b \\ A &\rightarrow Ac | Sd | \epsilon \end{aligned} \tag{4.12}$$

Нетерминал  $S$  леворекурсивен, поскольку  $S \Rightarrow Aa \Rightarrow Sda$ , но эта рекурсия не является непосредственной.

Алгоритм 4.1, приведенный ниже, удаляет из грамматики левую рекурсию. Он гарантированно работает с грамматиками, не имеющими циклов (порождений типа  $A \overset{+}{\Rightarrow} A$ ) и  $\epsilon$ -продукций (продукций типа  $A \rightarrow \epsilon$ ). Из грамматики могут быть также удалены и циклы, и  $\epsilon$ -продукции (см. упр. 4.20 и 4.22).

#### Алгоритм 4.1. Устранение левой рекурсии

*Вход.* Грамматика  $G$  без циклов и  $\epsilon$ -продукций.

*Выход.* Эквивалентная грамматика без левой рекурсии.

*Метод.* Применить алгоритм, приведенный на рис. 4.7. Обратите внимание, что результирующая грамматика без левых рекурсий может иметь  $\epsilon$ -продукции.  $\square$

1. Расположить нетерминалы в некотором порядке  $A_1, A_2, \dots, A_n$
2. **for**  $i:=1$  to  $n$  **do begin**
  - for**  $j:=1$  to  $i-1$  **do begin**
    - Заменить каждую продукцию вида  $A_i \rightarrow A_j \gamma$
    - продукциями  $A_i \rightarrow \delta_1 \gamma | \delta_2 \gamma | \dots | \delta_k \gamma$
    - где  $A_j \rightarrow \delta_1 | \delta_2 | \dots | \delta_k$  – все текущие  $A_j$ -продукции
  - end**
  - УстраниТЬ непосредственную левую рекурсию
  - среди  $A_i$ -продукций**end**

Рис. 4.7. Алгоритм устранения левой рекурсии из грамматики

Обоснование работоспособности приведенного на рис. 4.7 алгоритма основано на том, что после  $(i-1)$ -й итерации внешнего цикла **for** шага (2) в любой продукции вида  $A_k \rightarrow A_l \alpha$ ,  $k < i$ , должно быть  $l > k$ . В результате при следующей итерации внутреннего цикла (по  $j$ ) растет нижний предел  $m$  всех продукции вида  $A_i \rightarrow A_m \alpha$  до тех пор, пока не будет достигнуто  $m \geq i$ . Затем из  $A_i$ -продукций устраняется непосредственная левая рекурсия, вследствие чего достигается  $m > i$ .

#### Пример 4.9

Применим описанную процедуру к грамматике (4.12). Технически из-за наличия  $\epsilon$ -продукции алгоритм может не работать, но в данном случае продукция  $A \rightarrow \epsilon$  не мешает работе.

Мы располагаем нетерминалы в порядке  $S, A$ . Непосредственной левой рекурсии среди  $S$ -продукций нет, так что в процессе работы шага (2) при  $i = 1$  ничего не происходит. Для  $i = 2$  мы подставляем  $S$ -продукцию в  $A \rightarrow Sd$  для получения следующей  $A$ -продукции:  $A \rightarrow Ac | Aad | bd | \epsilon$ .

Устранение непосредственной левой рекурсии дает грамматику

$$\begin{array}{lcl} S & \rightarrow & Aa | b \\ A & \rightarrow & bdA' | A' \\ A' & \rightarrow & cA' | adA' | \epsilon \end{array}$$

□

## Левая факторизация

*Левая факторизация* (left factoring) представляет собой преобразование грамматики в пригодную для предиктивного анализа. Основная идея левой факторизации заключается в том, что когда не ясно, какая из двух альтернативных продукции должна использоваться для нетерминала  $A$ ,  $A$ -продукции можно переписать так, чтобы отложить принятие решения до тех пор, пока из входного потока не будет прочитано достаточно символов для правильного выбора.

Например, если есть две продукции

$$\begin{array}{lcl} stmt & \rightarrow & \text{if } expr \text{ then } stmt \text{ else } stmt \\ & | & \text{if } expr \text{ then } stmt \end{array}$$

то, обнаружив во входном потоке `if`, мы не в состоянии тут же выбрать ни одну из них. В общем случае, если  $A \rightarrow \alpha\beta_1 | \alpha\beta_2$  представляют собой две  $A$ -продукции и входной поток начинается с непустой строки, порождаемой  $\alpha$ , то мы не можем сказать, будет ли использоваться первая или вторая продукция. Однако можно отложить решение, расширив  $A$  до  $\alpha A'$ . В этом случае, после того как рассмотрен входной поток, выводимый из  $\alpha$ , мы работаем с  $A'$ , расширяя его до  $\beta_1$  или  $\beta_2$ . Таким образом, будучи левофакторизованными, исходные продукции становятся

$$\begin{array}{lcl} A & \rightarrow & \alpha A' \\ A' & \rightarrow & \beta_1 | \beta_2 \end{array}$$

### Алгоритм 4.2. Левая факторизация грамматики

*Вход.* Грамматика  $G$ .

*Выход.* Эквивалентная левофакторизованная грамматика.

*Метод.* Для каждого нетерминала  $A$  находим наилоннейший префикс  $\alpha$ , общий для двух или большего числа альтернатив. Если  $\alpha \neq \epsilon$ , т.е. имеется нетривиальный общий префикс, заменим все продукции  $A \rightarrow \alpha\beta_1 | \alpha\beta_2 | \dots | \alpha\beta_n | \gamma$ , где  $\gamma$  представляет все альтернативы, не начинающиеся с  $\alpha$ , продукциями

$$\begin{array}{lcl} A & \rightarrow & \alpha A' | \gamma \\ A' & \rightarrow & \beta_1 | \beta_2 | \dots | \beta_n \end{array}$$

Здесь  $A'$  — новый нетерминал. Выполняем это преобразование до тех пор, пока никакие две альтернативы не будут иметь общий префикс. □

### Пример 4.10

Следующая грамматика абстрагирует проблему “кочующего else”:

$$\begin{array}{lcl} S & \rightarrow & iEtS \mid iEtSeS \mid a \\ E & \rightarrow & b \end{array} \quad (4.13)$$

Здесь  $i$ ,  $t$  и  $e$  означают **if**, **then** и **else**,  $E$  и  $S$  соответствуют *expr* и *stmt*. Будучи левофакторизованной, эта грамматика принимает следующий вид:

$$\begin{array}{lcl} S & \rightarrow & iEtSS' \mid a \\ S' & \rightarrow & eS \mid \epsilon \\ E & \rightarrow & b \end{array} \quad (4.14)$$

Таким образом, при получении на входе  $i$  мы используем продукцию  $iEtSS'$  и ждем, пока  $iEtS$  не будет просмотрена, чтобы затем решить, расширять ли  $S'$  до  $eS$  или  $\epsilon$ . Конечно, и грамматика (4.13), и грамматика (4.14) неоднозначны, и при входном символе  $e$  неясно, какая из альтернатив для  $S'$  должна быть выбрана. В упражнении 4.19 обсуждается путь решения этой дилеммы.  $\square$

### Построение не-контекстно-свободных грамматик

То, что некоторые языки не могут быть сгенерированы ни одной грамматикой, не должно удивлять. Фактически во многих языках программирования имеются синтаксические конструкции, которые не могут быть определены только грамматикой. В этом разделе мы представим ряд таких конструкций, используя для иллюстрации абстрактные языки.

### Пример 4.11

Рассмотрим абстрактный язык  $L_1 = \{ wsw \mid w \in (a \mid b)^*\}$ .  $L_1$  состоит из всех слов, образованных повторяющимися строками из  $a$  и  $b$ , разделенными символом  $s$ , например  $aabcaab$ . Можно доказать, что этот язык не является контекстно-свободным. Он абстрагирует проблему проверки объявления переменных до их использования — первое  $w$  в  $wsw$  представляет объявление переменной, а второе  $w$  — ее использование. Из того, что  $L_1$  не является контекстно-свободным языком, следует, что языки типа Algol или Pascal не есть контекстно-свободными, так как в них требуется объявление переменной до ее использования, а идентификаторы могут иметь произвольную длину. Доказательство этого лежит за пределами данной книги.

По этой причине грамматики, описывающие синтаксис Algol или Pascal, не определяют символы в идентификаторе. Все идентификаторы представлены в грамматике токеном типа **id**. В компиляторах этих языков проверка того, что идентификаторы объявлены до их использования, производится на фазе семантического анализа.  $\square$

### Пример 4.12

Язык  $L_2 = \{ a^n b^m c^n d^m \mid n \geq 1, m \geq 1\}$  не является контекстно-свободным. Он состоит из строк, порождаемых регулярным выражением  $a^*b^*c^*d^*$ , причем количество  $a$  в строке равно количеству  $c$ , а количество  $b$  — количеству  $d$  (вспомните, что запись  $a^n$  обозначает  $a$ , записанное  $n$  раз). Язык  $L_2$  абстрагирует проблему проверки соответствия количества фактических параметров при вызове процедуры количеству формальных параметров в ее объявлении (т.е.  $a^n$  и  $b^m$  могут представлять списки формальных параметров в двух про-

цидурах, объявленных как имеющие, соответственно,  $n$  и  $m$  аргументов, а  $c^n$  и  $d^m$  — списки фактических параметров в вызовах этих двух процедур).

Отметим, что типичный синтаксис как определения процедуры, так и ее использования не определяет сам по себе количество ее параметров. Например, инструкция CALL в языках типа Fortran может быть описана как

$$\begin{array}{lcl} \text{stmt} & \rightarrow & \text{call id} (\text{expr\_list}) \\ \text{expr\_list} & \rightarrow & \text{expr\_list}, \text{expr} \\ & | & \text{expr} \end{array}$$

с соответствующими продукциями для  $\text{expr}$ . Проверка того, что число действительных параметров в вызове корректно, выполняется обычно во время семантического анализа.  $\square$

### Пример 4.13

Язык  $L_3 = \{a^n b^n c^n \mid n \geq 0\}$ , т.е. строки  $L(a^* b^* c^*)$  с одинаковым количеством  $a$ ,  $b$  и  $c$ , также не является контекстно-свободным. Пример проблемы, которую абстрагирует язык  $L_3$ , заключается в следующем. В типографском тексте, где для простейшего алфавитно-цифрового принтера используется подчеркивание, применяется курсив. При конвертировании файла с текстом, предназначенный для печати на простейшем принтере, в типографский текст, следует распознать все подчеркивания и преобразовать их в курсив. Подчеркнутое слово в данном файле представляет собой слово, за которым следует столько символов “забоя” (backspace), сколько и букв в слове, а затем — то же количество символов подчеркивания. Если рассматривать  $a$  как произвольную букву,  $b$  — как символ забоя и  $c$  — как символ подчеркивания, то язык  $L_3$  представляет подчеркнутые слова. Таким образом, мы не можем использовать грамматику для задания подчеркнутых описанным способом слов. Если же представить подчеркнутые слова как последовательность подчеркнутых букв, т.е. троек буква-забой-подчеркивание, то в этом случае подчеркнутые слова описываются регулярным выражением  $(abc)^*$ .  $\square$

Интересно отметить, что языки, очень похожие на  $L_1$ ,  $L_2$ ,  $L_3$ , являются контекстно-свободными. Например, язык  $L'_1 = \{wcw^R \mid w \in (a \mid b)^*\}$ , где  $w^R$  представляет “перевернутую” строку  $w$ , является контекстно-свободным и генерируется грамматикой

$$S \rightarrow aSa \mid bSb \mid c$$

Язык  $L'_2 = \{a^n b^m c^n d^m \mid n \geq 1, m \geq 1\}$  контекстно-свободный, с грамматикой

$$\begin{array}{lcl} S & \rightarrow & aSd \mid aAd \\ A & \rightarrow & bAc \mid bc \end{array}$$

Язык  $L'_3 = \{a^n b^m c^n d^m \mid n \geq 1, m \geq 1\}$  также контекстно-свободный, с грамматикой

$$\begin{array}{lcl} S & \rightarrow & AB \\ A & \rightarrow & aAb \mid ab \\ B & \rightarrow & cBd \mid cd \end{array}$$

И наконец, язык  $L'_3 = \{a^n b^n \mid n \geq 1\}$  также контекстно-свободный, с грамматикой

$$S \rightarrow aSb \mid ab$$

Стоит отметить, что  $L'_3$  представляет собой прототип примера языка, не определяемого никаким регулярным выражением. Для того чтобы убедиться в этом, предположим, что  $L'_3$  — язык, определяемый некоторым регулярным выражением. Это эквивалентно предположению, что мы можем построить ДКА  $D$ , допускающий  $L'_3$ .  $D$  должен иметь некоторое конечное число состояний, скажем  $k$ . Рассмотрим последовательность состоя-

ний  $s_0, s_1, s_2, \dots, s_k$ , в которые  $D$  входит при считывании последовательностей  $\epsilon, a, aa, \dots, a^k$ , т.е.  $D$  попадает в состояние  $s_i$ , прочитав  $i$  символов  $a$ .

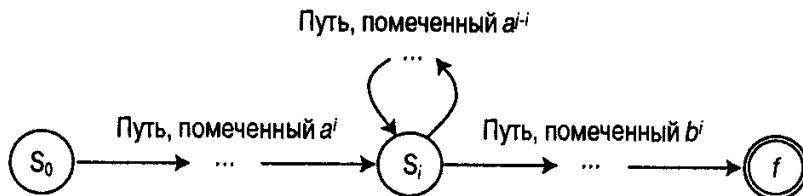


Рис. 4.8. ДКА  $D$ , допускающий  $a^i b^i$  и  $a^j b^i$

Поскольку  $D$  имеет только  $k$  различных состояний, как минимум, два состояния в последовательности  $s_0, s_1, s_2, \dots, s_k$  должны совпадать, скажем  $s_i$  и  $s_j$ . Из состояния  $s_i$  последовательность из  $i$  символов  $b$  приводит  $D$  в заключительное состояние  $f$ , поскольку  $a^i b^i \in L'_3$ . Но тогда имеется также путь из начального состояния  $s_0$  в  $s_i$  и  $f$ , помеченный как  $a^j b^i$ , как показано на рис. 4.8. Таким образом,  $D$  допускает также строку  $a^j b^i \notin L'_3$ . Это противоречит предположению о том, что  $L'_3$  допускается ДКА  $D$ .

Неформально говоря, “конечный автомат не помнит количества”, т.е. он не может допускать язык типа  $L'_3$ , для которого требуется запомнить количество просмотренных символов  $a$ , перед тем как приступить к просмотру символов  $b$ . Аналогично мы можем сказать, что “грамматика может запомнить количество двух элементов, но не трех”, поскольку с помощью грамматики можно определить  $L'_3$ , но не  $L_3$ .

## 4.4. Нисходящий анализ

В этом разделе мы рассмотрим основные идеи нисходящего разбора и покажем, как построить эффективный нисходящий синтаксический анализатор без отката, называемый предиктивным синтаксическим анализатором. Мы определим класс грамматик LL(1), на основе которых эти анализаторы могут строиться автоматически. Кроме формализации предиктивных синтаксических анализаторов из раздела 2.4, рассмотрим нерекурсивные предиктивные синтаксические анализаторы. Завершится этот раздел обсуждением проблемы восстановления после ошибок. Восходящие синтаксические анализаторы будут рассмотрены в разделах 4.5–4.7.

### Анализ методом рекурсивного спуска

Нисходящий разбор можно рассматривать как попытку найти левое порождение входной строки. Точно так же его можно рассматривать и как попытку построить дерево разбора для входной строки, начиная с корня и создавая вершины в прямом порядке. В разделе 2.4 мы познакомились со специальным случаем анализа методом рекурсивного спуска, именуемым предиктивным разбором, при котором не требуется откат. Сейчас мы рассмотрим нисходящий анализ в общем виде, а именно — анализ методом рекурсивного спуска, который может использовать откаты, т.е. производить повторное сканирование входного потока. Однако такие синтаксические анализаторы с откатом встречаются не очень часто. Одна из причин этого состоит в том, что для анализа конструкций реальных языков программирования откат требуется крайне редко. В ситуациях наподобие анализа естественных языков технология отката не очень эффективна, и предпочтительными являются таб-

личные методы, вроде алгоритма динамического программирования из упр. 4.63 или метод из работы [118] (см. также описание общих методов анализа в [18]).

Ниже приведен пример, в котором откат необходим. Здесь же предложен способ отслеживания входного потока при откате.

### Пример 4.14

Рассмотрим грамматику

$$\begin{aligned} S &\rightarrow cAd \\ A &\rightarrow ab \mid a \end{aligned}$$

и входную строку  $w = cad$ . При нисходящем построении дерева разбора для этой строки мы вначале создаем дерево, состоящее из одного узла, помеченного как  $S$ . Указатель входа указывает на  $c$ , первый символ строки  $w$ . Теперь воспользуемся первой продукцией для  $S$ , чтобы получить дерево, показанное на рис. 4.9a.

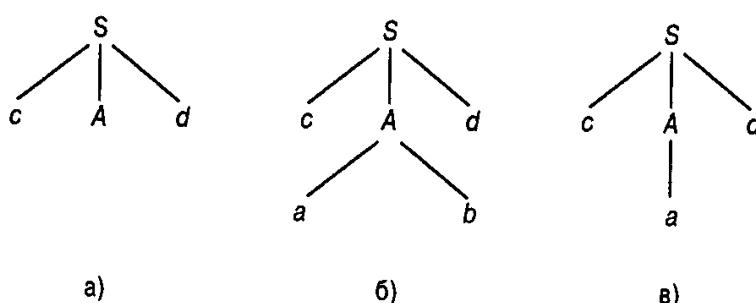


Рис. 4.9. Шаги нисходящего разбора

Крайний слева лист,  $c$ , соответствует первому символу  $w$ , так что переместим указатель входа к  $a$ , второму символу строки  $w$ , и рассмотрим следующий лист дерева, помеченный  $A$ . Теперь можно воспользоваться для  $A$  первой альтернативой и получить дерево, изображенное на рис. 4.9б. Нами обнаружено соответствие считанного символа  $a$  листу дерева, и мы переходим к следующему символу —  $d$ . Однако  $d$  не соответствует листу дерева  $b$ , а значит, мы должны вернуться к  $A$  с тем, чтобы выбрать новую альтернативу для работы.

Возвращаясь к  $A$ , мы должны вернуть указатель входа в позицию 2, в которой он был, когда мы впервые пришли к разложению  $A$ . Это означает, что процедура для  $A$  (аналогичная процедуре для нетерминалов на рис. 2.17) должна хранить указатель входа в локальной переменной. При рассмотрении второй альтернативы для  $A$  мы получаем дерево, показанное на рис. 4.9в. Лист  $a$  соответствует второму символу  $w$ , а лист  $d$  — третьему. Поскольку в этот момент нами построено дерево разбора для  $w$ , мы прекращаем работу и сообщаем об успешном завершении разбора. □

Леворекурсивная грамматика может вызвать зацикливание синтаксического анализатора, работающего методом рекурсивного спуска, даже с откатами (при разборе  $A$  может потребоваться вновь проанализировать  $A$ , находясь в той же входной позиции, что автоматически приводит к бесконечной рекурсии).

## Предиктивные анализаторы

Во многих случаях аккуратная разработка грамматики, устранение из нее левой рекурсии и ее левая факторизация позволяют получить грамматику, которая может быть проанализирована синтаксическим анализатором, работающим методом рекурсивного

спуска и не требующим отката (т.е. предиктивным анализатором, обсуждавшимся в разделе 2.4). Для построения предиктивного синтаксического анализатора мы должны знать, какая из альтернатив продукции  $A \rightarrow \alpha_1|\alpha_2|...|\alpha_n$  данного анализируемого нетерминала  $A$  порождает строку, начинающуюся с полученного входного символа  $a$ . Иначе говоря, правильная альтернатива должна точно определяться по первому порождаемому ею символу. Обычно в языках программирования управляющие конструкции отвечают этому правилу. Например, если есть продукция

$$\begin{array}{lcl} stmt & \rightarrow & \text{if } expr \text{ then } stmt \text{ else } stmt \\ & | & \text{while } expr \text{ do } stmt \\ & | & \text{begin } stmt\_list \text{ end} \end{array}$$

то ключевые слова `if`, `while` и `begin` однозначно определяют возможную альтернативу для рассматриваемой инструкции `stmt`.

## Диаграммы переходов предиктивных синтаксических анализаторов

В разделе 2.4 мы рассмотрели реализацию предиктивных синтаксических анализаторов с помощью рекурсивных процедур, как было показано на рис. 2.17. Так же, как в разделе 3.4 диаграмма переходов представляла собой план лексического анализатора, при построении предиктивного синтаксического анализатора мы можем создать его план в виде диаграммы переходов.

При этом мы обнаружим ряд отличий между диаграммами переходов лексического и предиктивного синтаксического анализаторов. В случае синтаксического анализатора для каждого нетерминала имеется своя диаграмма переходов. Метки дуг представляют собой токены и нетерминалы. Переход, помеченный токеном (терминалом), означает, что мы должны осуществить его, если данный токен является очередным символом во входном потоке; переход, помеченный нетерминалом, означает вызов процедуры для этого нетерминала.

Для построения диаграммы переходов предиктивного синтаксического анализатора на основе грамматики вначале следует устраниТЬ из нее левые рекурсии, а затем провести левую факторизацию. После этого для каждого нетерминала  $A$  выполняется следующее.

1. Создаем начальное и заключительное состояния.
2. Для каждой продукции  $A \rightarrow X_1X_2...X_n$  создаем путь от начального к заключительному состоянию с дугами, помеченными как  $X_1, X_2, \dots, X_n$ .

Предиктивный синтаксический анализатор работает с диаграммой переходов следующим образом. Изначально он находится в начальном состоянии стартового символа. Если после некоторых действий синтаксический анализатор находится в состоянии  $s$  с дугой, помеченной терминалом  $a$  и ведущей в состояние  $t$ , и если бчередной входной символ представляет собой  $a$ , то синтаксический анализатор перемещает входной курсор на одну позицию вправо и переходит в состояние  $t$ . Если же дуга помечена нетерминалом  $A$ , то синтаксический анализатор переходит в начальное состояние для этого нетерминала, не перемещая при этом входной курсор. По достижении заключительного состояния для  $A$  синтаксический анализатор переходит в состояние  $t$ , фактически “прочитав” из входного потока нетерминал  $A$  при переходе из состояния  $s$  в состояние  $t$ . И, наконец, если из состояния  $s$  в состояние  $t$  ведет дуга, помеченная как  $\epsilon$ , то из состояния  $s$  синтаксический анализатор непосредственно переходит в состояние  $t$ , не перемещаясь по входному потоку.

Программа предиктивного синтаксического анализа, созданная на основе диаграммы переходов, пытается найти соответствие каждому терминальному символу из входного потока и осуществляет потенциально рекурсивные вызовы процедур для каждой дуги, помеченной нетерминалом. Нерекурсивная реализация может быть получена путем помещения состояния  $s$  в стек при исходящем переходе из  $s$  по нетерминальному символу и снятия его со стека, когда достигнуто заключительное состояние для этого нетерминала.

Описанный выше подход работает корректно, если в диаграмме переходов нет недетерминизма, т.е. отсутствуют узлы с более чем одним переходом для одного входного символа. Как показано в следующем примере, неоднозначности могут быть разрешены специальным способом. Если же устранить недетерминизм невозможно, то и построить предиктивный синтаксический анализатор для этой грамматики невозможно (но можно построить синтаксический анализатор, который работает по методу рекурсивного спуска с откатом и исследует все имеющиеся возможности в поисках корректной).  $\square$

#### Пример 4.15

На рис. 4.10 приведен набор диаграмм переходов для грамматики (4.11). Неоднозначности в этих диаграммах связаны только с  $\epsilon$ -дугами. Рассмотрим дуги, исходящие из состояния  $E'$ . Будем считать, что при получении из входного потока символа  $+$  мы должны осуществить переход по дуге, помеченной этим символом, а в противном случае —  $\epsilon$ -переход (аналогичное правило следует использовать и в состоянии  $T'$ ). Таким образом мы благополучно устраним неоднозначность и можем написать программу предиктивного синтаксического анализатора для грамматики (4.11).  $\square$

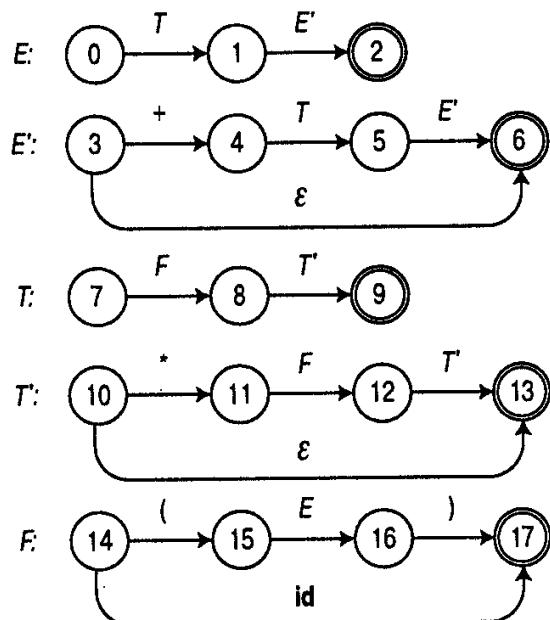


Рис. 4.10. Диаграмма переходов для грамматики (4.11)

Диаграммы переходов могут быть упрощены подстановкой одних диаграмм в другие; такая подстановка похожа на преобразования грамматики, используемые в разделе 2.5. Например, на рис. 4.11 $a$  вызов  $E'$  из себя же можно заменить переходом в начало диаграммы переходов для  $E'$ .

На рис. 4.11 $b$  показана полученная эквивалентная диаграмма для  $E'$ , которую мы можем подставить вместо перехода по  $E'$  в диаграмму для  $E$  на рис. 4.10, получив диаграмму, изображенную на рис. 4.11 $c$ . И, наконец, мы видим, что в этой диаграмме первый и

третий узлы эквивалентны, поэтому можно объединить их, получив в результате диаграмму, показанную на рис. 4.11г. Та же технология применима и к диаграммам для  $T$  и  $T'$ . Полностью набор упрощенных диаграмм переходов приведен на рис. 4.12. Программа предиктивного разбора, реализованная на основе этих диаграмм, работает на 20–25% быстрее, чем программа на основе набора диаграмм, показанного на рис. 4.10.

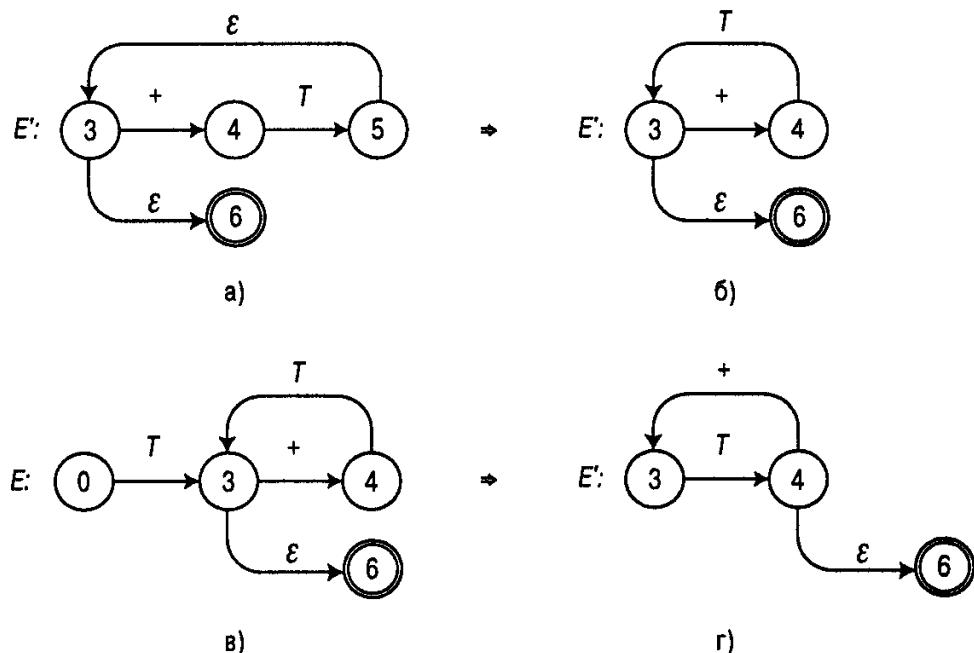


Рис. 4.11. Упрощенная диаграмма переходов

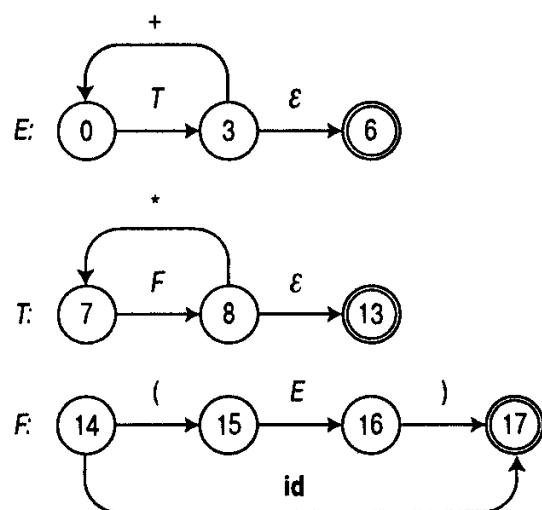


Рис. 4.12. Упрощенная диаграмма переходов для арифметических выражений

## Нерекурсивный предиктивный анализ

Нерекурсивный предиктивный синтаксический анализатор можно построить с помощью явного использования стека (вместо неявного при рекурсивных вызовах). Ключевая проблема предиктивного анализа заключается в определении продукции, которую следует применить к нетерминалу. Нерекурсивный синтаксический анализатор, представленный на рис. 4.13, ищет необходимую для анализа продукцию в таблице разбора (далее мы увидим, каким образом можно построить эту таблицу на основе грамматики).

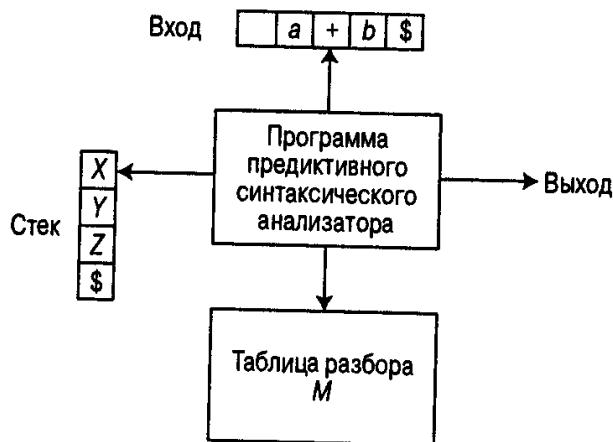


Рис. 4.13. Модель нерекурсивного предиктивного синтаксического анализатора

Предиктивный синтаксический анализатор, управляемый таблицей, имеет входной буфер, стек, таблицу разбора и выходной поток. Входной буфер содержит анализируемую строку с маркером ее правого конца — специальным символом. Стек содержит последовательность символов грамматики с \$ на дне. Изначально стек содержит стартовый символ грамматики непосредственно над символом \$. Таблица разбора представляет собой двухмерный массив  $M[A, a]$ , где  $A$  — нетерминал, а  $a$  — терминал или символ \$.

Синтаксический анализатор управляет программой, которая работает следующим образом. Программа рассматривает  $X$  — символ на вершине стека, и  $a$ , текущий входной символ. Эти два символа определяют действия синтаксического анализатора. Имеется три варианта.

1. Если  $X = a = \$$ , синтаксический анализатор прекращает работу и сообщает об успешном завершении разбора.
2. Если  $X = a \neq \$$ , синтаксический анализатор снимает со стека  $X$  и перемещает указатель входного потока к следующему символу.
3. Если  $X$  представляет собой нетерминал, программа рассматривает запись  $M[X, a]$  из таблицы разбора  $M$ . Эта запись представляет собой либо  $X$ -продукцию грамматики, либо запись об ошибке. Если, например,  $M[X, a] = \{X \rightarrow UVW\}$ , синтаксический анализатор замещает  $X$  на вершине стека на  $UVW$  (с  $U$  на вершине стека). Мы полагаем, что в качестве выхода синтаксический анализатор просто выводит использованную продукцию, но, конечно же, здесь может выполняться любой необходимый код. Если  $M[X, a] = \text{error}$ , синтаксический анализатор вызывает программу восстановления после ошибки.

Поведение синтаксического анализатора может описываться его *конфигурациями*, которые дают содержимое стека и оставшийся входной поток.

#### Алгоритм 4.3. Нерекурсивный предиктивный анализ

*Вход.* Стока  $w$  и таблица разбора  $M$  грамматики  $G$ .

*Выход.* В случае  $w \in L(G)$  — левое порождение  $w$ ; в противном случае — сообщение об ошибке.

*Метод.* Изначально синтаксический анализатор находится в конфигурации с  $\$S$  (на его вершине — стартовый символ  $S$  грамматики  $G$ ) и строкой  $w\$$  во входном буфере. Программа, использующая для анализа входного потока таблицу предиктивного разбора  $M$ , показана на рис. 4.14. □

Установить указатель *ip* на первый символ *w\$*;

**repeat**

Обозначим через *X* символ на вершине стека,  
а через *a* — символ, на который указывает *ip*.

**if** *X* — терминал или *\$ then*

**if** *X* = *a then*

Снять со стека *X* и переместить *ip*

**else** *error()*

**else** /\* *X* — нетерминал \*/

**if** *M[X, a] = X → Y<sub>1</sub>Y<sub>2</sub>...Y<sub>k</sub> then begin*

Снять *X* со стека;

Поместить в стек *Y<sub>k</sub>, Y<sub>k-1</sub>, ..., Y<sub>1</sub>*, с *Y<sub>1</sub>*

на вершине стека; вывести продукцию *X → Y<sub>1</sub>Y<sub>2</sub>...Y<sub>k</sub>*

**end**

**else** *error()*

**until** *X = \$ /\* Стек пуст \*/*

Рис. 4.14. Программа предиктивного синтаксического анализа

### Пример 4.16

Рассмотрим грамматику (4.11) из примера 4.8. Таблица предиктивного анализа этой грамматики показана на рис. 4.15. Пустые ячейки таблицы означают ошибки; непустые указывают продукции, с помощью которых заменяются нетерминалы на вершине стека. Заметьте, пока мы не указываем, каким образом выбирать записи из таблицы — об этом поговорим чуть позже.

| НЕТЕРМИНАЛ | ВХОДНОЙ СИМВОЛ |                  |                  |                |               |               |
|------------|----------------|------------------|------------------|----------------|---------------|---------------|
|            | id             | +                | *                | (              | )             | \$            |
| <i>E</i>   | <i>E → TE'</i> |                  |                  | <i>E → TE'</i> |               |               |
| <i>E'</i>  |                | <i>E' → +TE'</i> |                  |                | <i>E' → ε</i> | <i>E' → ε</i> |
| <i>T</i>   | <i>T → FT'</i> |                  |                  | <i>T → FT'</i> |               |               |
| <i>T'</i>  |                | <i>T' → ε</i>    | <i>T' → *FT'</i> |                | <i>T' → ε</i> | <i>T' → ε</i> |
| <i>F</i>   | <i>F → id</i>  |                  |                  | <i>F → (E)</i> |               |               |

Рис. 4.15. Таблица разбора для грамматики (4.11)

| СТЕК                             | ВХОД       | ВЫХОД            |
|----------------------------------|------------|------------------|
| \$ <i>E</i>                      | id+id*id\$ |                  |
| \$ <i>E'</i> <i>T</i>            | id+id*id\$ | <i>E → TE'</i>   |
| \$ <i>E'</i> <i>T'</i> <i>F</i>  | id+id*id\$ | <i>T → FT'</i>   |
| \$ <i>E'</i> <i>T'</i> <i>id</i> | id+id*id\$ | <i>F → id</i>    |
| \$ <i>E'</i> <i>T'</i>           | +id*id\$   |                  |
| \$ <i>E'</i>                     | +id*id\$   | <i>T' → ε</i>    |
| \$ <i>E'</i> <i>T</i> +          | +id*id\$   | <i>E' → +TE'</i> |
| \$ <i>E'</i> <i>T</i>            | id*id\$    |                  |

Рис. 4.16. Перемещения предиктивного синтаксического анализатора при входной строке *id+id\*id*

| Стек       | Вход    | Выход                     |
|------------|---------|---------------------------|
| \$E' T' F  | id*id\$ | $T \rightarrow FT'$       |
| \$E' T' id | id*id\$ | $F \rightarrow id$        |
| \$E' T'    | *id\$   |                           |
| \$E' T' F* | *id\$   | $T' \rightarrow *FT'$     |
| \$E' T' F  | id\$    |                           |
| \$E' T' id | id\$    | $F \rightarrow id$        |
| \$E' T'    | \$      |                           |
| \$E'       | \$      | $T' \rightarrow \epsilon$ |
| \$         | \$      | $E' \rightarrow \epsilon$ |

Рис. 4.16. Перемещения предиктивного синтаксического анализатора при входной строке  $id+id^*id$  (окончание)

При входном потоке  $id+id^*id$  предиктивный синтаксический анализатор осуществляет последовательность перемещений, показанную на рис. 4.16. Входной указатель при перемещении указывает на крайний слева символ в столбце “Вход”. Если мы внимательно рассмотрим действия этого синтаксического анализатора, то обнаружим, что его выход совпадает с последовательностью продукции, применяемых в левом порождении. Если же к прочитанным входным символам присписать символы в стеке (от вершины до дна), то получится левосентенциальная форма в порождении.

## FIRST и FOLLOW

При построении предиктивного синтаксического анализатора нам помогут две функции, связанные с грамматикой  $G$ , — FIRST и FOLLOW, которые обеспечивают заполнение таблицы предиктивного анализа грамматики  $G$ . Множества токенов, порождаемые функцией FOLLOW, могут также использоваться как синхронизирующие токены в процессе восстановления после ошибки в “режиме паники”.

Если  $\alpha$  — произвольная строка символов грамматики, то определим  $\text{FIRST}(\alpha)$  как множество терминалов, с которых начинаются строки, выводимые из  $\alpha$ . Если  $\alpha \Rightarrow^* \epsilon$ , то  $\epsilon \in \text{FIRST}(\alpha)$ .

Определим  $\text{FOLLOW}(A)$  для нетерминала  $A$  как множество терминалов  $a$ , которые могут располагаться непосредственно справа от  $A$  в некоторой сентенциальной форме, т.е. множество терминалов  $a$ , таких, что существует порождения вида  $S \Rightarrow^* \alpha A a \beta$  для некоторых  $\alpha$  и  $\beta$ . Заметим, что в процессе приведения между  $A$  и  $a$  могут появиться символы, но они порождают  $\epsilon$  и в конечном счете исчезают. Если  $A$  может оказаться крайним справа символом некоторой сентенциальной формы, то  $\$ \in \text{FOLLOW}(A)$ .

Чтобы вычислить  $\text{FIRST}(X)$  для всех символов грамматики  $X$ , будем применять следующие правила до тех пор, пока ни к одному из множеств FIRST не смогут быть добавлены ни терминалы, ни  $\epsilon$ .

1. Если  $X$  — терминал, то  $\text{FIRST}(X) = \{X\}$ .
2. Если имеется продукция  $X \rightarrow \epsilon$ , добавим  $\epsilon$  к  $\text{FIRST}(X)$ .
3. Если  $X$  — нетерминал и имеется продукция  $X \rightarrow Y_1 Y_2 \dots Y_k$ , то поместим  $a \in \text{FIRST}(Y_i)$  в  $\text{FIRST}(X)$ , если для некоторого  $i$   $a \in \text{FIRST}(Y_i)$  и  $\epsilon$  входит во все множества  $\text{FIRST}(Y_1), \dots, \text{FIRST}(Y_{i-1})$ , т.е.  $Y_1 \dots Y_{i-1} \Rightarrow^* \epsilon$ . Если  $\epsilon$  имеется во всех  $\text{FIRST}(Y_i)$ ,  $i = \overline{1, k}$ , то добавляем

$\epsilon$  к  $\text{FIRST}(X)$ . Например, все, что находится в множестве  $\text{FIRST}(Y_1)$ , есть и в множестве  $\text{FIRST}(X)$ . Если  $Y_1$  не порождает  $\epsilon$ , то больше мы ничего не добавляем к  $\text{FIRST}(X)$ , но если  $Y_1 \Rightarrow^* \epsilon$ , то к  $\text{FIRST}(X)$  добавляем  $\text{FIRST}(Y_2)$  и т.д.

Теперь можно вычислить  $\text{FIRST}$  для любой строки  $X_1 X_2 \dots X_n$  следующим образом. Добавим к  $\text{FIRST}(X_1 X_2 \dots X_n)$  все не- $\epsilon$  символы из  $\text{FIRST}(X_1)$ . Добавим также все не- $\epsilon$  символы из  $\text{FIRST}(X_2)$ , если  $\epsilon \in \text{FIRST}(X_1)$ , все не- $\epsilon$  символы из  $\text{FIRST}(X_3)$ , если  $\epsilon$  имеется как в  $\text{FIRST}(X_1)$ , так и в  $\text{FIRST}(X_2)$  и т.д. И наконец, добавим  $\epsilon$  к  $\text{FIRST}(X_1 X_2 \dots X_n)$ , если для всех  $i$   $\text{FIRST}(X_i)$  содержит  $\epsilon$ .

Чтобы вычислить  $\text{FOLLOW}(A)$  для всех нетерминалов  $A$ , будем применять следующие правила до тех пор, пока ни к одному множеству  $\text{FOLLOW}$  нельзя будет добавить ни одного символа.

1. Поместим  $\$$  в  $\text{FOLLOW}(S)$ , где  $S$  — стартовый символ, а  $\$$  — правый ограничитель входного потока.
2. Если имеется продукция  $A \rightarrow \alpha B \beta$ , то все элементы множества  $\text{FIRST}(\beta)$ , кроме  $\epsilon$ , помещаются в множество  $\text{FOLLOW}(B)$ .
3. Если имеется продукция  $A \rightarrow \alpha B$  или  $A \rightarrow \alpha B \beta$ , где  $\text{FIRST}(\beta)$  содержит  $\epsilon$  (т.е.  $\beta \Rightarrow^* \epsilon$ ), то все элементы из множества  $\text{FOLLOW}(A)$  помещаются в множество  $\text{FOLLOW}(B)$ .

#### Пример 4.17

Обратимся вновь к грамматике (4.11):

$$\begin{aligned} E &\rightarrow TE' \\ E' &\rightarrow +TE' \mid \epsilon \\ T &\rightarrow FT' \\ T' &\rightarrow *FT' \mid \epsilon \\ F &\rightarrow (E) \mid \text{id} \end{aligned}$$

Тогда

$$\begin{aligned} \text{FIRST}(E) = \text{FIRST}(T) = \text{FIRST}(F) &= \{(), \text{id}\} \\ \text{FIRST}(E') &= \{+, \epsilon\} \\ \text{FIRST}(T') &= \{*, \epsilon\} \\ \text{FOLLOW}(E) = \text{FOLLOW}(E') &= \{(), \$\} \\ \text{FOLLOW}(T) = \text{FOLLOW}(T') &= \{+, (), \$\} \\ \text{FOLLOW}(F) &= \{+, *, (), \$\} \end{aligned}$$

Например,  $\text{id}$  и левая скобка добавляются к  $\text{FIRST}(F)$  согласно правилу (3) в определении  $\text{FIRST}$ , поскольку  $\text{FIRST}(\text{id}) = \{\text{id}\}$ , и к  $\text{FIRST}(()) = \{()\}$  согласно правилу (1). По правилу (3) с  $i=1$  из продукции  $T \rightarrow FT'$  следует, что  $\text{id}$  и левая скобка входят и в  $\text{FIRST}(T)$ . В соответствии с правилом (2)  $\epsilon$  входит в  $\text{FIRST}(E')$ .

Для вычисления множеств  $\text{FOLLOW}$  помещаем  $\$$  в  $\text{FOLLOW}(E)$  в соответствии с правилом (1) для вычисления  $\text{FOLLOW}$ . По правилу (2), примененному к продукции  $F \rightarrow (E)$ , правая скобка также входит в множество  $\text{FOLLOW}(E)$ . В соответствии с правилом (3), примененным к продукции  $E \rightarrow TE'$ ,  $\$$  и правая скобка входят в  $\text{FOLLOW}(E')$ . Поскольку  $E' \Rightarrow^* \epsilon$ , эти же символы входят и в  $\text{FOLLOW}(T)$ . Из продукции  $E \rightarrow TE'$  согласно правилу (2), вытекает, что все, что имеется в множестве  $\text{FIRST}(E')$  (за исключени-

ем  $\epsilon$ ), должно входить в множество  $\text{FOLLOW}(T)$ . Как мы уже видели, в это множество входит также  $\$$ .  $\square$

## Построение таблиц предиктивного анализа

Для построения таблицы предиктивного анализа данной грамматики  $G$  может использоваться приведенный далее алгоритм. Идея, лежащая в основе алгоритма, состоит в следующем. Предположим, что  $A \rightarrow \alpha$  представляет собой продукцию, у которой  $a \in \text{FIRST}(\alpha)$ . Тогда синтаксический анализатор заменит  $A$  строкой  $\alpha$  при текущем входном символе  $a$ . Единственная сложность возникает при  $\alpha = \epsilon$  или  $\alpha \xrightarrow{*} \epsilon$ . В этом случае мы снова должны заменить  $A$  на  $\alpha$ , если текущий входной символ имеется в  $\text{FOLLOW}(A)$  или из входного потока получен  $\$$ , который входит в  $\text{FOLLOW}(A)$ .

### Алгоритм 4.4. Построение таблицы предиктивного анализа

*Вход.* Грамматика  $G$ .

*Выход.* Таблица анализа  $M$ .

*Метод.*

1. Для каждой продукции грамматики  $A \rightarrow \alpha$  выполняем шаги 2 и 3.
2. Для каждого терминала  $a$  из  $\text{FIRST}(\alpha)$  добавляем  $A \rightarrow \alpha$  в ячейку  $M[A, a]$ .
3. Если в  $\text{FIRST}(\alpha)$  входит  $\epsilon$ , для каждого терминала  $b$  из  $\text{FOLLOW}(A)$  добавим  $A \rightarrow \alpha$  в ячейку  $M[A, b]$ . Если  $\epsilon$  входит в  $\text{FIRST}(\alpha)$ , а  $\$$  — в  $\text{FOLLOW}(A)$ , добавим  $A \rightarrow \alpha$  в ячейку  $M[A, \$]$ .
4. Сделаем каждую неопределенную ячейку таблицы  $M$  указывающей на ошибку.  $\square$

### Пример 4.18

Применим алгоритм 4.4 к грамматике (4.11). Поскольку  $\text{FIRST}(TE') = \text{FIRST}(T) = \{ , \text{id}\}$ , продукция  $E \rightarrow TE'$  приводит к размещению в ячейках  $M[E, ()]$  и  $M[E, \text{id}]$  записи  $E \rightarrow TE'$ .

Продукция  $E' \rightarrow +TE'$  позволяет внести ее в ячейку  $M[E', +]$ . Продукция  $E' \rightarrow \epsilon$  приводит, с учетом  $\text{FOLLOW}(E') = \{ , \$\}$ , к внесению  $E' \rightarrow \epsilon$  в ячейки  $M[E', ()]$  и  $M[E', \$]$ .

Полностью таблица предиктивного анализа, построенная по алгоритму 4.4 для грамматики (4.11), приведена на рис. 4.15.  $\square$

## LL(1)-грамматики

Алгоритм 4.4 может быть применен к любой грамматике  $G$  для получения таблицы разбора  $M$ . Однако для некоторых грамматик  $M$  может иметь несколько записей в одной ячейке таблицы. Например, если  $G$  — леворекурсивная или неоднозначная грамматика, то  $M$  будет иметь как минимум одну ячейку с несколькими записями.

### Пример 4.19

Обратимся вновь к грамматике (4.13) из примера 4.10.

$$\begin{aligned} S &\rightarrow iEtSS' \mid a \\ S' &\rightarrow es \mid \epsilon \\ E &\rightarrow b \end{aligned}$$

Таблица анализа для этой грамматики показана на рис. 4.17.

| НЕТЕРМИНАЛ | ВХОДНОЙ СИМВОЛ    |                   |                                                  |                        |          |                           |
|------------|-------------------|-------------------|--------------------------------------------------|------------------------|----------|---------------------------|
|            | <i>a</i>          | <i>b</i>          | <i>e</i>                                         | <i>i</i>               | <i>t</i> | \$                        |
| <i>S</i>   | $S \rightarrow a$ |                   |                                                  | $S \rightarrow iEtSS'$ |          |                           |
| <i>S'</i>  |                   |                   | $S' \rightarrow \epsilon$<br>$S' \rightarrow eS$ |                        |          | $S' \rightarrow \epsilon$ |
| <i>E</i>   |                   | $E \rightarrow b$ |                                                  |                        |          |                           |

Рис. 4.17. Таблица анализа для грамматики (4.13)

Ячейка таблицы  $M[S', e]$  содержит две записи —  $S' \rightarrow eS$  и  $S' \rightarrow \epsilon$ , поскольку  $\text{FOLLOW}(S') = \{e, \$\}$ . Неоднозначность грамматики проявляется в выборе продукции, используемой при получении из входного потока символа *e* (*else*). Мы можем разрешить неоднозначность выбором  $S' \rightarrow eS$ , что означает соответствие *else* ближайшему предыдущему *then*. Заметим, что выбор  $S' \rightarrow \epsilon$  будет препятствовать внесению *e* в стек и удалению из входного потока, что, естественно, неверно.  $\square$

Грамматика, таблица анализа которой не имеет множественных записей, называется *LL(1)*. Первое “L” означает просмотр входного потока слева направо, второе “L” — левое порождение, а “1” — просмотр одного символа из входного потока на каждом шаге для принятия решения о дальнейших действиях. Можно показать, что алгоритм 4.4 для каждой *LL(1)*-грамматики *G* строит таблицу анализа, которая разбирает все предложения *G*, и только их.

*LL(1)*-грамматики имеют ряд отличительных свойств. Такая грамматика не может быть неоднозначной или леворекурсивной. Можно также показать, что грамматика *G* является *LL(1)*-грамматикой тогда и только тогда, когда для любых двух различных ее продукции  $A \rightarrow \alpha \mid \beta$  выполняются следующие условия.

1. Не существует такого терминала *a*, для которого  $\alpha$  и  $\beta$  порождают строку, начинающуюся с *a*.
2. Пустую строку может порождать только одна из продукции  $\alpha$  или  $\beta$ .
3. Если  $\beta \Rightarrow^* \epsilon$ , то  $\alpha$  не порождает ни одну строку, начинающуюся с терминала из  $\text{FOLLOW}(A)$ .

Ясно, что грамматика (4.11) для арифметических выражений является *LL(1)*-грамматикой. Грамматика же (4.13), моделирующая инструкции *if-then-else*, таковой не является.

Остается вопрос, что делать, если таблица анализа имеет ячейки с несколькими записями. Один выход состоит в преобразовании грамматики, устраниющем левую рекурсию, и левой факторизации, в надежде получить грамматику, в таблице анализа которой отсутствуют ячейки с несколькими записями. К сожалению, имеются грамматики, никакие изменения которых не приведут к *LL(1)*-грамматике. Примером может служить грамматика (4.13). Как мы видели, грамматику (4.13) можно анализировать с помощью предиктивного синтаксического анализатора путем выбора  $M[S', e] = \{S' \rightarrow eS\}$ . Вообще-то, не существует универсальных правил, с помощью которых ячейки с несколькими записями можно превратить в однозначно определенные без воздействия на язык, распознаваемый синтаксическим анализатором.

Основная сложность в использовании предиктивного анализа состоит в написании для исходного языка такой грамматики, которая позволяет построить предиктивный син-

таксический анализатор. Хотя устранение левой рекурсии и левая факторизация просты в реализации, они делают грамматику неудобочитаемой и трудной для трансляции. Обычная организация синтаксического анализатора в компиляторе состоит в использовании предиктивного синтаксического анализатора для управляющих конструкций и приоритета операторов (обсуждаемого в разделе 4.6) для выражений. Однако если имеется генератор LR-анализаторов, описанный в разделе 4.9, то можно получить все преимущества предиктивного анализа и приоритета операторов автоматически.

## Восстановление после ошибок в предиктивном анализе

Стек нерекурсивных предиктивных синтаксических анализаторов делает явными терминалы и нетерминалы, соответствие которым синтаксический анализатор намеревается найти в оставшейся части входного потока. Поэтому в дальнейшем мы будем ссылаться на символы, находящиеся в стеке синтаксического анализатора. Синтаксический анализатор находит ошибку в тот момент, когда терминал на вершине стека не соответствует очередному входному символу или на вершине стека находится нетерминал  $A$ , очередной входной символ —  $a$ , а ячейка таблицы синтаксического анализа  $M[A, a]$  пуста.

Восстановление после ошибки “в режиме паники” основано на пропуске символов из входного потока до тех пор, пока не будет обнаружен токен из выбранного множества синхронизирующих токенов. Эффективность этого метода зависит от выбора синхронизирующего множества. Множества должны выбираться так, чтобы синтаксический анализатор быстро восстанавливался после часто встречающихся ошибок. Вот некоторые эвристические правила.

1. В качестве первого приближения можем поместить в синхронизирующее множество для нетерминала  $A$  все символы из множества  $\text{FOLLOW}(A)$ . Если пропустим все токены до элемента из  $\text{FOLLOW}(A)$  и снимем  $A$  со стека, вероятно, мы сможем продолжить синтаксический анализ.
2. В качестве синхронизирующего множества для  $A$  недостаточно использовать  $\text{FOLLOW}(A)$ . Например, если инструкция завершается точкой с запятой, как в C, то ключевое слово, с которого начинается инструкция, может не оказаться во множестве  $\text{FOLLOW}$  нетерминалов, порождающих выражения. Отсутствующая точка с запятой после присвоения может, таким образом, привести к пропуску ключевого слова, с которого начинается новая инструкция. Зачастую в языке имеется иерархическая структура конструкций — например, выражения появляются в инструкциях, которые находятся в блоках, и т.д. В таком случае к синхронизирующему множеству конструкции нижнего уровня можно добавить символы, с которых начинаются конструкции более высокого уровня. Например, можно добавить ключевые слова, с которых начинаются инструкции, в синхронизирующие множества нетерминалов, порождающих выражения.
3. Если мы добавим символы из  $\text{FIRST}(A)$  в синхронизирующее множество для нетерминала  $A$ , станет возможным продолжение анализа в соответствии с  $A$ , когда во входном потоке появится символ из множества  $\text{FIRST}(A)$ .
4. Если нетерминал может порождать пустую строку, то в качестве продукции по умолчанию может использоваться продукция, порождающая  $\epsilon$ . Это может отсрочить обнаружение ошибки, зато не может вызвать ее пропуск, и сокращает число нетерминалов, которые должны быть рассмотрены в процессе восстановления после ошибки.

5. Если терминал на вершине стека не может быть сопоставлен с входным символом, то простейший метод состоит в снятии терминала со стека, генерации сообщения о вставке терминала в программу и продолжении синтаксического анализа. По сути, при этом подходе синхронизирующее множество токена состоит из всех остальных токенов.

### Пример 4.20

Использование символов из множеств FOLLOW и FIRST в качестве синхронизирующих достаточно неплохо работает при разборе грамматики (4.11). Таблица синтаксического анализа для этой грамматики, приведенная на рис. 4.15, повторена на рис. 4.18. “Synch” указывает синхронизирующие токены, полученные из множества FOLLOW соответствующего терминала. Множества FOLLOW для нетерминалов получены из примера 4.17.

| НЕТЕРМИНАЛ | ВХОДНОЙ СИМВОЛ      |                           |                       |                     |                           |                           |
|------------|---------------------|---------------------------|-----------------------|---------------------|---------------------------|---------------------------|
|            | id                  | +                         | *                     | (                   | )                         | \$                        |
| $E$        | $E \rightarrow TE'$ |                           |                       | $E \rightarrow TE'$ | <i>Synch</i>              | <i>Synch</i>              |
| $E'$       |                     | $E' \rightarrow +TE'$     |                       |                     | $E' \rightarrow \epsilon$ | $E' \rightarrow \epsilon$ |
| $T$        | $T \rightarrow FT'$ | <i>Synch</i>              |                       | $T \rightarrow FT'$ | <i>Synch</i>              | <i>Synch</i>              |
| $T'$       |                     | $T' \rightarrow \epsilon$ | $T' \rightarrow *FT'$ |                     | $T' \rightarrow \epsilon$ | $T' \rightarrow \epsilon$ |
| $F$        | $F \rightarrow id$  | <i>Synch</i>              | <i>Synch</i>          | $F \rightarrow (E)$ | <i>Synch</i>              | <i>Synch</i>              |

Рис. 4.18. Добавление к таблице на рис. 4.15 синхронизирующих токенов

Таблица на рис. 4.18 используется следующим образом. Если синтаксический анализатор просматривает ячейку  $M[A, a]$  и обнаруживает, что она пуста, то входной символ  $a$  пропускается. Если в этой ячейке находится запись *Synch*, то нетерминал снимается с вершины стека в попытке продолжить синтаксический анализ. Если токен на вершине стека не соответствует входному символу, то, как упоминалось выше, мы снимаем его со стека.

В случае ошибочного ввода  $)id^*+id$  синтаксический анализатор и механизм восстановления после ошибок с рис. 4.18 ведут себя так, как показано на рис. 4.19.  $\square$

| СТЕК       | ВХОД         | ПРИМЕЧАНИЕ             |
|------------|--------------|------------------------|
| $$E$       | $)id^*+id\$$ | Ошибка, пропускаем “)” |
| $$E$       | $id^*+id\$$  | $id \in FIRST(E)$      |
| $$E'T$     | $id^*+id\$$  |                        |
| $$E'T'F$   | $id^*+id\$$  |                        |
| $$E'T'id$  | $id^*+id\$$  |                        |
| $$E'T'$    | $*+id\$$     |                        |
| $$E'T'F^*$ | $*+id\$$     |                        |
| $$E'T'F$   | $+id\$$      | Ошибка, $M[F,+]=Synch$ |
| $$E'T'$    | $+id\$$      | $F$ снимается со стека |
| $$E'$      | $+id\$$      |                        |
| $$E'T^+$   | $+id\$$      |                        |
| $$E'T$     | $id\$$       |                        |

Рис. 4.19. Перемещения предиктивного синтаксического анализатора при синтаксическом анализе и восстановлении после ошибки

| СТЕК         | ВХОД | ПРИМЕЧАНИЕ |
|--------------|------|------------|
| $\$E' T' F$  | id\$ |            |
| $\$E' T' id$ | id\$ |            |
| $\$E' T'$    | \$   |            |
| $\$E'$       | \$   |            |
| \$           | \$   |            |

Рис. 4.19. Перемещения предиктивного синтаксического анализатора при синтаксическом анализе и восстановлении после ошибки (окончание)

Приведенное обсуждение метода восстановления после ошибки “в режиме паники” не касалось важного вопроса сообщений об ошибках. Вообще говоря, разработчик компилятора должен обеспечить вывод информативных сообщений об обнаруженных ошибках.

**Восстановление на уровне фразы.** Восстановление на уровне фразы реализуется заполнением пустых ячеек в таблице предиктивного синтаксического анализа указателями на подпрограммы обработки ошибок. Эти подпрограммы могут изменять, вставлять или удалять символы входного потока и выводить соответствующие сообщения об ошибках. Кроме того, они могут снимать элементы со стека. При таком способе восстановления возникает вопрос, должны ли мы разрешить изменение символов в стеке или только размещение в стеке новых символов, поскольку после этого шаги, выполняемые синтаксическим анализатором, могут не соответствовать порождению ни одного слова языка вообще. В любом случае нужно гарантировать, что программа не зацикливается. Надежная защита от зацикливания — проверка того, что каждое действие по восстановлению после ошибки в конечном счете приводит к обработке очередного входного символа (или к снятию элементов со стека, если достигнут конец входного потока).

## 4.5. Восходящий синтаксический анализ

В этом разделе будет рассмотрен основной метод восходящего синтаксического анализа, известный как синтаксический анализ типа “перенос/свертка” (shift-reduce) и называемый далее сокращенно ПС-анализом. Простой для реализации вид ПС-анализа, именуемый синтаксическим анализом приоритета операторов, представлен в разделе 4.6. Более общий метод ПС-анализа, который называется LR-анализом, обсуждается в разделе 4.7 и используется во многих автоматических генераторах синтаксических анализаторов.

ПС-анализ пытается строить дерево разбора для входной строки, начиная с листьев (снизу) и работая по направлению к корню дерева (вверх). Этот процесс можно рассматривать как свертку строки *w* к стартовому символу грамматики. На каждом шаге *свертки* (reduction step) некоторая подстрока, соответствующая правой части продукции, заменяется символом из левой части этой продукции, и если на каждом шаге подстроки выбираются корректно, то мы получаем обращенное правое порождение.

### Пример 4.21

Рассмотрим грамматику

$$\begin{array}{lcl} S & \rightarrow & aABe \\ A & \rightarrow & Abc \mid b \\ B & \rightarrow & d \end{array}$$

Предложение  $abbcde$  сводится к  $S$  с помощью следующих шагов:

$abbcde$   
 $aAbcde$   
 $aAde$   
 $aABe$   
 $S$

Мы сканируем строку  $abbcde$  в поисках подстроки, соответствующей правой части какой-либо продукции. Такими подстроками являются  $b$  и  $d$ . Выберем крайнее слева  $b$  и заменим его нетерминалом  $A$ , который представляет собой левую часть продукции  $A \rightarrow b$ ; таким образом, получим строку  $aAbcde$ . Теперь правым частям продукции соответствуют подстроки  $Abc$ ,  $b$  и  $d$ . Хотя  $b$  и является крайней слева подстрокой, соответствующей правой части одной из продукции, выберем для замены подстроку  $Abc$  и заменим ее нетерминалом  $A$  в соответствии с продукцией  $A \rightarrow Abc$ . В результате получим строку  $aAde$ . Заменяя  $d$  на  $B$ , левую часть продукции  $B \rightarrow d$ , получаем  $aABe$ , которая в соответствии с первой продукцией заменяется стартовым символом  $S$ . Итак, последовательность из четырех сверток позволяет привести строку  $abbcde$  к стартовому символу  $S$ . Эти сокращения представляют собой обращенное (т.е. записанное в обратном порядке) правое приведение  $S \xrightarrow{m} aABe \xrightarrow{m} aAde \xrightarrow{m} aAbcde \xrightarrow{m} abbcde$ .  $\square$

## Основы

Неформально говоря, основа, или дескриптор (handle) строки — это подстрока, которая совпадает с правой частью продукции и свертка которой в левую часть продукции представляет собой один шаг обращенного правого порождения. Во многих случаях крайняя слева подстрока  $\beta$ , соответствующая правой части некоторой продукции  $A \rightarrow \beta$ , не является основой, поскольку свертка в соответствии с продукцией  $A \rightarrow \beta$  приводит к строке, которая не может быть свернута к стартовому символу. Если в примере 4.21 мы заменим во второй строке  $aAbcde$  символ  $b$  нетерминалом  $A$ , то получим строку  $aAAcde$ , которая не может быть свернута в  $S$ . По этой причине нам следует дать более точное определение основы.

Говоря формально, *основа* правосентенциальной формы  $\gamma$  является продукцией  $A \rightarrow \beta$  и позицией строки  $\beta$  в  $\gamma$ , такими, что  $\beta$  может быть заменена нетерминалом  $A$  для получения предыдущей правосентенциальной формы в правом порождении  $\gamma$ . Таким образом, если  $S \xrightarrow{m} \alpha Aw \xrightarrow{m} \alpha \beta w$ , то  $A \rightarrow \beta$  в позиции после  $\alpha$  представляет собой основу строки  $\alpha \beta w$ . Стока  $w$  справа от основы содержит только терминальные символы. Заметим, что грамматика может быть неоднозначной, с несколькими правыми порождениями  $\alpha \beta w$ . Если грамматика однозначна, то каждая правосентенциальная форма грамматики имеет ровно одну основу.

В приведенном выше примере  $abbcde$  представляет собой правосентенциальную форму, основой которой является  $A \rightarrow \beta$  в позиции 2. Аналогично  $aAbcde$  представляет собой правосентенциальную форму, дескриптор которой —  $A \rightarrow Abc$  в позиции 2. Иногда мы будем говорить “подстрока  $\beta$  представляет собой основу  $\alpha \beta w$ ”, если позиция  $\beta$  и продукция  $A \rightarrow \beta$  определяются однозначно.

На рис. 4.20 изображена основа  $A \rightarrow \beta$  в дереве разбора правосентенциальной формы  $\alpha \beta w$ . Основа представляет крайнее слева завершенное поддерево, состоящее из узла и

всех его потомков. На рис. 4.20 узел  $A$  — нижний крайний слева внутренний узел, все потомки которого находятся в дереве. Свертку  $\beta$  к  $A$  в  $\alpha\beta w$  можно представить как “обрезку основы”, т.е. удаление из дерева разбора всех потомков  $A$ .

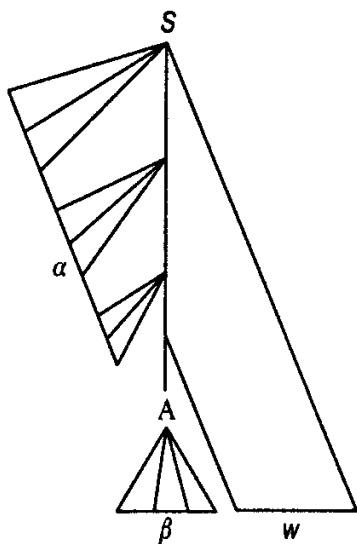


Рис. 4.20. Основа  $A \rightarrow \beta$  в дереве разбора  $\alpha\beta w$

### Пример 4.22

Рассмотрим следующую грамматику

- (1)  $E \rightarrow E+E$
  - (2)  $E \rightarrow E^*E$
  - (3)  $E \rightarrow (E)$
  - (4)  $E \rightarrow \text{id}$
- (4.16)

и правое порождение

$$\begin{aligned}
 E &\xrightarrow{rm} \underline{E+E} \\
 &\xrightarrow{rm} \underline{E+E^*E} \\
 &\xrightarrow{rm} \underline{E+E^*\underline{\text{id}_3}} \\
 &\xrightarrow{rm} \underline{E+\underline{\text{id}_2}^*\underline{\text{id}_3}} \\
 &\xrightarrow{rm} \underline{\text{id}_1} + \underline{\text{id}_2}^*\underline{\text{id}_3}
 \end{aligned}$$

Для удобства мы пометили подстрочными индексами  $\text{id}$  и подчеркнули основу каждой правосентенциальной формы. Например,  $\text{id}_1$  представляет собой основу правосентенциальной формы  $\text{id}_1 + \text{id}_2^* \text{id}_3$ , поскольку  $\text{id}$  является правой частью продукции  $E \rightarrow \text{id}$ , и замена  $\text{id}_1$  на  $E$  приведет к предыдущей правосентенциальной форме  $E + \text{id}_2^* \text{id}_3$ . Обратите внимание на то, что строка справа от основы состоит только из терминальных символов.

Поскольку грамматика (4.16) неоднозначна, имеется еще одно правое порождение той же строки:

$$\begin{aligned}
 E &\xrightarrow{rm} \underline{E+E} \\
 &\xrightarrow{rm} E^* \underline{\text{id}_3}
 \end{aligned}$$

$$\begin{aligned}
 &\stackrel{m}{\Rightarrow} E + E * \underline{id}_3 \\
 &\stackrel{m}{\Rightarrow} E + \underline{id}_2 * id_3 \\
 &\stackrel{m}{\Rightarrow} \underline{id}_1 + id_2 * id_3
 \end{aligned}$$

Рассмотрим правосентенциальную форму  $E + E * id_3$ . В этом порождении  $E + E$  — основа  $E + E * id_3$ , в то время как в ранее представленном порождении ее основой является  $id_3$ .

В этом примере два правых порождения — аналог двух левых порождений из примера 4.6. Первое порождение дает оператору  $*$  больший приоритет, чем оператору  $+$ , в то время как во втором порождении выше приоритет оператора  $+$ .  $\square$

## Обрезка основ

Обращенное правое порождение может быть получено посредством “обрезки основ”. Мы начинаем процесс со строки терминалов  $w$ , которую хотим проанализировать. Если  $w$  — предложение рассматриваемой грамматики, то  $w = \gamma_n$ , где  $\gamma_n$  —  $n$ -я правосентенциальная форма некоторого, еще неизвестного правого порождения  $S = \gamma_0 \stackrel{m}{\Rightarrow} \gamma_1 \stackrel{m}{\Rightarrow} \gamma_2 \stackrel{m}{\Rightarrow} \dots \stackrel{m}{\Rightarrow} \gamma_{n-1} \stackrel{m}{\Rightarrow} \gamma_n = w$ . Для воссоздания этого порождения в обратном порядке мы находим основу  $\beta_n$  в  $\gamma_n$  и заменяем ее левой частью продукции  $A_n \rightarrow \beta_n$  для получения  $(n-1)$ -й сентенциальной формы  $\gamma_{n-1}$ . Заметим, что пока мы не знаем, каким образом искать основы, но вскоре познакомимся с соответствующими методами.

Затем мы повторяем описанный процесс, т.е. находим в  $\gamma_{n-1}$  основу  $\beta_{n-1}$  и свертываем ее для получения правосентенциальной формы  $\gamma_{n-2}$ . Если после очередного шага правосентенциальная форма содержит только стартовый символ  $S$ , мы прекращаем процесс и сообщаем об успешном завершении анализа. Обращенная последовательность продукции, использованных в свертках, представляет собой правое порождение входной строки.

### Пример 4.23

Рассмотрим грамматику (4.16) из примера 4.22 и входную строку  $id_1 + id_2 * id_3$ . Последовательность сверток, приводящая входную строку к стартовому символу  $E$ , показана на рис. 4.21. Читатель должен заметить, что последовательность правосентенциальных форм в этом примере представляет собой обращение первой последовательности правых порождений из примера 4.22.  $\square$

| ПРАВОСЕНТЕНЦИАЛЬНАЯ ФОРМА | Основа  | СВОРАЧИВАЮЩАЯ ПРОДУКЦИЯ |
|---------------------------|---------|-------------------------|
| $id_1 + id_2 * id_3$      | $id_1$  | $E \rightarrow id$      |
| $E + id_2 * id_3$         | $id_2$  | $E \rightarrow id$      |
| $E + E * id_3$            | $id_3$  | $E \rightarrow id$      |
| $E + E * E$               | $E * E$ | $E \rightarrow E * E$   |
| $E + E$                   | $E + E$ | $E \rightarrow E + E$   |
| $E$                       |         |                         |

Рис. 4.21. Свертки, выполняемые ПС-анализатором

## Стековая реализация ПС-анализа

Существует две проблемы при синтаксическом анализе методом обрезки основ. Первая заключается в обнаружении подстроки для свертки в правосентенциальной форме, а вторая — в определении, какая именно продукция должна быть выбрана, если имеется несколько продуктов с соответствующей подстрокой в правой части. Перед тем как ответить на эти вопросы, сначала рассмотрим структуры данных, используемые в ПС-анализаторе.

Достаточно удобный путь реализации ПС-анализатора состоит в использовании стека для хранения символов грамматики и входного буфера для хранения анализируемой строки. В качестве маркера дна стека мы используем \$, и этот же символ является маркером правого конца входной строки. Изначально стек пуст, а входной буфер содержит строку *w*:

Стек              Вход

\$              *w*\$

Синтаксический анализатор работает путем переноса нуля или нескольких символов в стек до тех пор, пока на вершине стека не окажется основа  $\beta$ . Затем он свертывает  $\beta$  к левой части соответствующей продукции. Синтаксический анализатор повторяет этот цикл, пока не будет обнаружена ошибка или он не придет в конфигурацию, когда в стеке находится только стартовый символ, а входной буфер пуст:

Стек              Вход

\$*S*              \$

Попав в эту конфигурацию, синтаксический анализатор прекращает работу и сообщает об успешном разборе входной строки.

### Пример 4.24

Пройдем пошагово все действия, выполняемые синтаксическим анализатором при разборе входной строки  $id_1+id_2*id_3$  грамматики (4.16), используя первое порождение из примера 4.22. Последовательность действий показана на рис. 4.22. Заметим, что поскольку грамматика (4.16) имеет два правых порождения для данной входной строки, существует еще одна последовательность переносов и сверток, которые может выполнить анализатор. □

| Стек                                                | Вход              | Действие                       |
|-----------------------------------------------------|-------------------|--------------------------------|
| (1) \$                                              | $id_1+id_2*id_3$$ | Перенос                        |
| (2) \$ <i>id</i> <sub>1</sub>                       | $+id_2*id_3$$     | Свертка по $E \rightarrow id$  |
| (3) \$ <i>E</i>                                     | $+id_2*id_3$$     | Перенос                        |
| (4) \$ <i>E</i> +                                   | $id_2*id_3$$      | Перенос                        |
| (5) \$ <i>E</i> + <i>id</i> <sub>2</sub>            | $*id_3$$          | Свертка по $E \rightarrow id$  |
| (6) \$ <i>E</i> + <i>E</i>                          | $*id_3$$          | Перенос                        |
| (7) \$ <i>E</i> + <i>E</i> *                        | $id_3$$           | Перенос                        |
| (8) \$ <i>E</i> + <i>E</i> * <i>id</i> <sub>3</sub> | \$                | Свертка по $E \rightarrow id$  |
| (9) \$ <i>E</i> + <i>E</i> * <i>E</i>               | \$                | Свертка по $E \rightarrow E*E$ |
| (10) \$ <i>E</i> + <i>E</i>                         | \$                | Свертка по $E \rightarrow E+E$ |
| (11) \$ <i>E</i>                                    | \$                | Допуск                         |

Рис. 4.22. Конфигурации ПС-анализатора для входной строки  $id_1+id_2*id_3$

Основными операциями синтаксического анализатора являются перенос и свертка, но на самом деле ПС-анализатор может выполнять четыре действия: (1) перенос, (2) свертка, (3) допуск, (4) ошибка.

1. При *переносе* очередной входной символ переносится на вершину стека.
2. При *свертке* синтаксический анализатор распознает правый конец основы на вершине стека, после чего он должен найти левый конец основы и принять решение о том, каким нетерминалом заменить основу.
3. При *допуске* синтаксический анализатор сообщает об успешном разборе входной строки.
4. При *ошибке* синтаксический анализатор обнаруживает ошибку во входном потоке и вызывает программу восстановления после ошибок.

Рассмотрим очень важный факт, который поясняет использование стека в ПС-анализаторе: основа всегда находится на вершине стека и никогда — внутри него. Это становится очевидным при рассмотрении возможных видов двух последовательных шагов в любом правом порождении. Эти два шага могут быть следующими:

- (1)  $S \xrightarrow[\text{rm}]{*} \alpha A z \xrightarrow[\text{rm}]{*} \alpha \beta B y z \xrightarrow[\text{rm}]{*} \alpha \beta \gamma y z$
- (2)  $S \xrightarrow[\text{rm}]{*} \alpha B x A z \xrightarrow[\text{rm}]{*} \alpha B x y z \xrightarrow[\text{rm}]{*} \alpha y x u z$

В случае (1)  $A$  заменяется на  $\beta B y$ , а затем крайний справа нетерминал  $B$  в правой части — на  $\gamma$ . В случае (2)  $A$  также заменяется первым, но в этот раз правая часть представляет собой строку  $y$ , состоящую только из терминалов. Следующий крайний справа нетерминал  $B$  находится слева от  $y$ .

Рассмотрим случай (1) в обратном порядке, начиная с момента, когда ПС-анализатор достиг конфигурации

| СТЕК  | ВХОД |
|-------|------|
| \$αβγ | yz\$ |

Теперь синтаксический анализатор сворачивает основу  $\gamma$  в  $B$  и переходит в конфигурацию

| СТЕК  | ВХОД |
|-------|------|
| \$αβB | yz\$ |

Поскольку  $B$  является крайним справа нетерминалом в  $\alpha\beta B y z$ , правый конец основы строки  $\alpha\beta B y z$  не может находиться внутри стека. Таким образом, синтаксический анализатор может перенести строку  $y$  в стек для получения конфигурации

| СТЕК   | ВХОД |
|--------|------|
| \$αβBy | z\$  |

в которой  $\beta B y$  является основой и сворачивается в  $A$ .

В случае (2) в конфигурации

| СТЕК | ВХОД  |
|------|-------|
| \$αy | xuz\$ |

основа  $y$  находится на вершине стека. После свертки  $y$  в  $B$  синтаксический анализатор может перенести строку  $xu$  для получения следующей основы  $u$  на вершине стека:

| СТЕК     | ВХОД  |
|----------|-------|
| $\$aBxy$ | $z\$$ |

Теперь синтаксический анализатор свертывает у в  $A$ .

В обоих случаях после свертки синтаксический анализатор для получения очередной основы переносит нуль или несколько символов в стек. Синтаксический анализатор никогда не заглядывает внутрь стека в поисках правого края основы. Все это делает стек особенно удобным для использования в реализации ПС-анализатора. Впрочем, мы все еще не выяснили, каким образом осуществлять выбор очередного действия для корректной работы синтаксического анализатора. Приоритет операторов и LR-анализаторы представляют собой интересующие нас технологии, с которыми мы вскоре детально ознакомимся.

## Активные префиксы

Префиксы правосентенциальных форм, которые встречаются в стеке ПС-анализатора, называются *активными* (viable prefixes). Эквивалентное определение активного префикса заключается в том, что это — префикс правосентенциальной формы, который не выходит за правый конец крайней справа основы этой сентенциальной формы. Согласно этому определению, к концу активного префикса всегда можно добавить терминальные символы для получения правосентенциальной формы. Следовательно, просканированная часть входного потока не содержит ошибок только в том случае, когда она может быть свернута в активный префикс.

## Конфликты в процессе ПС-анализа

Существуют контекстно-свободные грамматики, для которых ПС-анализ не применим. Любой ПС-анализатор для такой грамматики может достичь конфигурации, в которой синтаксический анализатор по информации о содержимом стека и об очередном входном символе не в состоянии решить, должен ли использоваться перенос или свертка (конфликт *перенос/свертка*, shift/reduce conflict), либо какая из нескольких возможных сверток должна применяться (конфликт *свертка/свертка*, reduce/reduce conflict). Сейчас мы рассмотрим несколько примеров синтаксических конструкций, приводящих к построению таких грамматик. Технически эти грамматики не входят в класс LR( $k$ )-грамматик, определенный в разделе 4.7; мы говорим о них как о не-LR-грамматиках.  $k$  в LR( $k$ ) означает количество символов входного потока, следующих за текущим, которые синтаксический анализатор может при необходимости просмотреть, не перенося в стек. Практически используемые грамматики в основном принадлежат классу LR(1).

### Пример 4.25

Неоднозначная грамматика не может быть LR-грамматикой. Рассмотрим классический случай грамматики “кочующего else” (4.7) из раздела 4.3:

```
stmt → if expr then stmt
 | if expr then stmt else stmt
 | other
```

Если ПС-анализатор находится в конфигурации

|                                     |             |
|-------------------------------------|-------------|
| СТЕК                                | ВХОД        |
| ... if <i>expr</i> then <i>stmt</i> | else ... \$ |

то мы не можем сказать, является ли подстрока *if expr then stmt* основой, независимо от того, что находится в стеке под нею. Здесь возникает конфликт перенос/свертка — в зависимости от того, что следует за *else* во входном потоке, верным решением может оказаться свертка *if expr then stmt* в *stmt*, а может — перенос *else* и поиск еще одного *stmt* для завершения альтернативы *if expr then stmt else stmt*. Таким образом, мы не можем сказать, что следует использовать в данном случае — перенос или свертку, а значит, грамматика не является LR(1)-грамматикой. Более того, никакая неоднозначная грамматика не может быть LR(*k*)-грамматикой ни при каком *k*.

Следует отметить, однако, что ПС-анализ может быть легко адаптирован к разбору некоторых неоднозначных грамматик вроде приведенной выше. При построении такого синтаксического анализатора для грамматики, содержащей две приведенные выше продукции, мы получим конфликт переноса/свертки — переноса *else* или свертки *stmt*  $\rightarrow$  *if expr then stmt*. Если мы разрешим конфликт в пользу переноса, синтаксический анализатор будет работать нормально. Синтаксические анализаторы для таких неоднозначных грамматик мы рассмотрим в разделе 4.8.  $\square$

Еще одна причина того, что грамматика не является LR, возникает, когда есть основа, но содержимого стека и очередного входного символа недостаточно для определения продукции, которая должна использоваться в свертке. Следующий пример иллюстрирует эту ситуацию.

#### Пример 4.26

Предположим, что есть лексический анализатор, который возвращает токен *id* для всех идентификаторов, независимо от их использования. Предположим также, что наш язык вызывает процедуры по именам, с параметрами, взятыми в скобки; тот же синтаксис используется и для работы с массивами. Поскольку трансляции индексов массива и параметров процедуры существенно отличаются друг от друга, мы должны использовать различные продукции для порождения списка фактических параметров и индексов. Следовательно, наша грамматика может иметь (среди прочих) продукции типа

- (1)  $stmt \rightarrow id (parameter\_list)$
- (2)  $stmt \rightarrow expr := expr$
- (3)  $parameter\_list \rightarrow parameter\_list , parameter$
- (4)  $parameter\_list \rightarrow parameter$
- (5)  $parameter \rightarrow id$
- (6)  $expr \rightarrow id (expr\_list)$
- (7)  $expr \rightarrow id$
- (8)  $expr\_list \rightarrow expr\_list , expr$
- (9)  $expr\_list \rightarrow expr$

Инструкция, начинающаяся с *A(I,J)*, будет передана синтаксическому анализатору как поток токенов *id(id,id)*. После переноса первых трех токенов в стек ПС-анализатор окажется в конфигурации

|             |            |
|-------------|------------|
| СТЕК        | ВХОД       |
| ... id ( id | , id ) ... |

Очевидно, что токен **id** на вершине стека должен быть свернут, но какой продукцией? Правильный выбор — продукция (5), если **A** — процедура, и (7), если **A** — массив. Содержимое стека не может подсказать, чем является **A**; для принятия решения мы должны использовать информацию из таблицы символов, которая была занесена туда при объявлении **A**.

Одно из решений состоит в замене токена **id** в продукции (1) на **procid** и использовании более интеллектуального лексического анализатора, который возвращает **procid** при распознавании идентификатора, представляющего собой имя процедуры. Такой способ требует от лексического анализатора обращения к таблице символов перед тем, как вернуть токен.

Если мы внесем эти изменения, то при обработке  $A(I, J)$  синтаксический анализатор может оказаться либо в конфигурации, приведенной ранее, либо в следующей:

| СТЕК                                            | ВХОД |
|-------------------------------------------------|------|
| ... <b>procid</b> ( <b>id</b> , <b>id</b> ) ... |      |

В первом случае выбираем свертку по продукции (7); в последнем — по продукции (5). Обратите внимание, что выбор определяется третьим от вершины символом в стеке, который даже не участвует в свертке. Для управления разбором ПС-анализ может использовать информацию “из глубин” стека.  $\square$

## 4.6. Синтаксический анализ приоритета операторов

Самый широкий класс грамматик, для которых успешно строятся ПС-анализаторы, — LR-грамматики — будет обсуждаться в разделе 4.7. Однако для небольшого, но важного класса грамматик мы легко можем построить эффективные ПС-анализаторы вручную. У этих грамматик нет продукции, правые части которых представляют собой  $\epsilon$  или имеют два соседних нетерминала. Грамматика при наличии последнего свойства называется *операторной* (operator grammar).

### Пример 4.27

Следующая грамматика для выражений

$$\begin{aligned} E &\rightarrow EAE \mid (E) \mid -E \mid id \\ A &\rightarrow + \mid - \mid * \mid / \mid \uparrow \end{aligned}$$

не является операторной, поскольку правая часть  $EAE$  имеет два (на самом деле — даже три) последовательных нетерминала. Однако если мы заменим  $A$  каждой из его альтернатив, то получим операторную грамматику

$$E \rightarrow E+E \mid E-E \mid E^*E \mid E/E \mid E\uparrow E \mid (E) \mid -E \mid id \tag{4.17}$$

Теперь опишем простую в реализации технологию синтаксического анализа, называемую анализом приоритета операторов. Впервые эта технология была описана как работа с токенами, без какого-либо упоминания о грамматике. В самом деле, по завершении построения синтаксического анализатора приоритета операторов мы можем практически игнорировать грамматику, используя нетерминалы в стеке просто как место хранения атрибутов, связанных с нетерминалами.

В качестве технологии синтаксического анализа общего назначения синтаксический анализ приоритета операторов имеет массу недостатков. Например, с его помощью

сложно обрабатывать токены вроде знака “минус”, который имеет два разных приоритета — в зависимости от того, является ли он унарным или бинарным. Более того, поскольку связь между грамматикой анализируемого языка и синтаксическим анализатором приоритета операторов весьма слабая, нельзя быть уверенными, что синтаксический анализатор допускает именно тот язык, который нас интересует. Наконец, класс грамматик, с которыми может работать такой синтаксический анализатор, весьма невелик.

Тем не менее из-за своей простоты эта технология используется для разбора выражений многими компиляторами. Зачастую эти синтаксические анализаторы для конструкций более высокого уровня применяют описанный в разделе 4.4 рекурсивный спуск. Имеются даже синтаксические анализаторы приоритета операторов, построенные для целых языков.

При синтаксическом анализе приоритета операторов мы определим три непересекающихся *отношения приоритетов* (precedence relations)<sup>5</sup> —  $<$ ,  $\doteq$  и  $>$  — между терминалами. Эти отношения приоритетов управляют выбором основ и имеют следующее содержание.

| ОТНОШЕНИЕ    | ЗНАЧЕНИЕ                              |
|--------------|---------------------------------------|
| $a < b$      | $a$ “уступает приоритет” $b$          |
| $a \doteq b$ | $a$ имеет тот же приоритет, что и $b$ |
| $a > b$      | $a$ “забирает приоритет у” $b$        |

Мы должны предостеречь читателя, что, хотя эти отношения и выглядят похожими на арифметические отношения “меньше”, “равно” и “больше”, они имеют совершенно иные свойства. Например, в одном и том же языке может быть так, что и  $a < b$ , и  $a > b$ , или для некоторых терминалов  $a$  и  $b$  не выполняется ни одно из отношений  $a < b$ ,  $a > b$  и  $a \doteq b$ .

Имеется два обычных способа определения отношений приоритетов, которые должны выполняться между терминалами. Первый метод — интуитивный. Он основан на традиционных понятиях ассоциативности и приоритета операторов. Например, если  $*$  имеет приоритет выше, чем  $+$ , то  $* > +$  и  $+ < *$ . Такой подход, как мы увидим, разрешает неоднозначности грамматики (4.17) и позволяет написать для нее синтаксический анализатор приоритета операторов (хотя унарный минус и вызывает определенные проблемы).

По второму методу выбора отношений приоритетов операторов вначале строится однозначная грамматика языка, которая отражает правильную ассоциативность и приоритеты операторов в своих деревьях разбора. Выполнить эту работу для выражений не слишком сложно; синтаксис выражений из раздела 2.2 отражает эту парадигму. Для другого источника неоднозначности, кочующего *else*, полезной моделью является грамматика (4.9). Получив однозначную грамматику, можно воспользоваться механическим методом построения на ее основе отношений приоритетов операторов. Эти отношения могут пересекаться и задавать язык, отличающийся от порождаемого исходной грамматикой, но со стандартными арифметическими выражениями проблем практически не бывает. Мы не будем рассматривать здесь указанное построение; оно описано в книге [18].

<sup>5</sup> В русскоязычной литературе иногда использовался термин “отношения предшествования”. — Прим. ред.

## Использование отношений приоритетов операторов

Цель отношений приоритетов состоит в определении границ основы правосентенциальной формы:  $<$  отмечает ее левый конец,  $>$  — правый, а  $\doteq$  находится внутри основы. Чтобы быть более точными, предположим, что у нас есть правосентенциальная форма операторной грамматики. Из того, что у продукции не может быть двух смежных нетерминалов в правой части, следует, что и правосентенциальная форма не может иметь двух смежных нетерминалов. Таким образом, мы можем записать правосентенциальную форму в виде  $\beta_0 a_1 \beta_1 \dots a_n \beta_n$ , где каждое  $\beta_i$  является либо  $\epsilon$  (пустой строкой), либо одиночным нетерминалом, а каждое  $a_i$  представляет собой одиночный терминал.

Предположим, что между  $a_i$  и  $a_{i+1}$  выполняется ровно одно отношение —  $>$ ,  $<$  или  $\doteq$ . Используем для маркировки концов строки символ  $\$$  и определим, что  $\$ < b$  и  $b > \$$  для всех терминалов  $b$ . Теперь предположим, что мы удалили из строки нетерминалы и поместили одно из отношений  $>$ ,  $<$  или  $\doteq$  между каждой парой терминалов и между крайними терминалами и маркерами  $\$$ . Пусть, например, вначале у нас есть правосентенциальная форма  $id + id * id$ , а отношения приоритетов показаны в таблице на рис. 4.23 (эти отношения выбраны при рассмотрении грамматики (4.17)).

|           | <b>id</b> | <b>+</b> | <b>*</b> | <b>\$</b> |
|-----------|-----------|----------|----------|-----------|
| <b>id</b> |           | $>$      | $>$      | $>$       |
| <b>+</b>  | $<$       | $>$      | $<$      | $>$       |
| <b>*</b>  | $<$       | $>$      | $>$      | $>$       |
| <b>\$</b> | $<$       | $<$      | $<$      |           |

Рис. 4.23. Отношения приоритетов операторов

Тогда строка с отношениями приоритетов принимает вид

$\$ < id > + < id > * < id > \$$  (4.18)

Например,  $<$  вставлен между крайним слева  $\$$  и  $id$ , поскольку в ячейке в строке  $\$$  и столбце  $id$  находится отношение  $<$ . Основа может быть найдена следующим образом.

1. Сканируем строку слева направо, пока не встретим первый символ  $>$ . В (4.18) этот символ располагается между первым  $id$  и  $+$ .
2. Затем сканируем строку в обратном направлении (влево), пропуская все  $\doteq$ , пока не встретим  $<$ . В (4.18) сканирование идет до символа  $\$$ .
3. Основа содержит все, что находится слева от первого  $>$  и справа от  $<$ , найденного на шаге (2), включая все промежуточные и окружающие нетерминалы (включение окружающих нетерминалов необходимо для того, чтобы в правосентенциальной форме не появлялись два смежных нетерминала). В (4.18) основой является первый  $id$ .

Работая с грамматикой (4.17), мы свертываем  $id$  в  $E$ , в результате чего получаем правосентенциальную форму  $E + id * id$ . После свертки двух оставшихся  $id$  в  $E$  мы получим правосентенциальную форму  $E + E * E$ . Рассмотрим теперь строку  $\$ + * \$$ , полученную удалением нетерминалов. Вставка отношений приоритетов дает строку  $\$ < + < * > \$$ , из которой ясно, что левый конец основы находится между  $+$  и  $*$ , а правый — между  $*$  и  $\$$ . Эти отношения приоритетов указывают, что в правосентенциальной форме  $E + E * E$  основой является  $E * E$ . Обратите внимание, что окружающие  $*$  нетерминалы  $E$  также являются частью основы.

Поскольку нетерминалы не влияют на процесс анализа, мы можем не беспокоиться об их распознавании. В стеке синтаксического анализатора приоритета операторов достаточно одного маркера “нетерминал”, чтобы обозначать места для хранения значений атрибутов.

Из описания метода может показаться, что на каждом шаге для поиска основы требуется сканирование всей правосентенциальной формы, однако это не так, если мы применяем стек для хранения уже просмотренных символов, и для управления синтаксическим анализатором типа “перенос/свертка” используются отношения приоритетов. Если между терминалом на вершине стека и очередным входным символом выполняется отношение  $<$  или  $\doteq$ , то синтаксический анализатор выполняет перенос (это означает, что правый конец основы еще не найден). Если отношение —  $>$ , то выполняется свертка. В этот момент синтаксический анализатор обнаружил правый конец основы, и для поиска ее левого конца в стеке можно воспользоваться отношениями приоритета.

Если между парами терминалов не выполняется ни одно из отношений приоритетов (обратите внимание на пустые ячейки в таблице на рис. 4.23), то обнаружена синтаксическая ошибка, и должна быть вызвана программа восстановления после ошибки (что обсуждается в этом разделе немного позже). Все описанные идеи можно формализовать следующим алгоритмом.

#### Алгоритм 4.5. Алгоритм синтаксического анализа приоритета операторов

*Вход.* Входная строка  $w$  и таблица отношений приоритетов.

*Выход.* В случае корректной строки  $w$  — скелетное дерево разбора с маркером — нетерминалом  $E$ , помечающим внутренние узлы; в противном случае — сообщение об ошибке.

*Метод.* Вначале стек содержит \$, а входной буфер —  $w\$$ . Для анализа используем программу, приведенную на рис. 4.24.  $\square$

```
(1) Устанавливаем указатель ip на первый символ $w\$$
(2) repeat forever
(3) if на вершине стека находится $,
 и ip указывает на $, then
 return
 else begin
(5) Пусть a — терминал на вершине стека,
 а b — символ, на который указывает ip
(6) if $a < b$ или $a \doteq b$ then begin
(7) Поместить b в стек
(8) Переместить ip к следующему входному символу
 end
(9) else if $a > b$ then /* Свертка */
(10) repeat
(11) Снять со стека
(12) until Терминал на вершине стека связан
 отношением $<$ с последним снятым
 терминалом
(13) else error()
end
```

Рис. 4.24. Алгоритм синтаксического анализа приоритета операторов

## Нахождение отношений приоритетов операторов

Мы вольны в создании отношений приоритета операторов любым подходящим способом и можем надеяться, что алгоритм синтаксического анализа приоритета операторов будет корректно работать с ними. Для языка арифметических выражений, наподобие генерируемого грамматикой (4.17), мы можем использовать следующую эвристику для создания корректного множества отношений приоритетов. Заметим, что грамматика (4.17) неоднозначна, и правосентенциальные формы могут иметь несколько основ. Наши правила должны выбрать “правильные” основы для отражения заданного множества правил ассоциативности и приоритета для бинарных операторов.

1. Если оператор  $\theta_1$  имеет более высокий приоритет, чем оператор  $\theta_2$ , определим, что  $\theta_1 > \theta_2$  и  $\theta_2 < \theta_1$ . Например, если оператор \* имеет более высокий приоритет, чем оператор +, то  $* > +$  и  $+ < *$ . Эти отношения гарантируют, что при получении выражения вида  $E+E^*E+E$  основой является  $E^*E$ , и именно  $E^*E$  будет свернуто первым.
2. Если  $\theta_1$  и  $\theta_2$  представляют собой операторы равного приоритета (это может быть один и тот же оператор), то устанавливаем, что  $\theta_1 > \theta_2$  и  $\theta_2 > \theta_1$ , если операторы левоассоциативны, и  $\theta_1 < \theta_2$  и  $\theta_2 < \theta_1$  в случае правоассоциативности. Например, для левоассоциативных + и - устанавливаем, что  $+ > -, - > -$  и  $- > +$ . Для правоассоциативного оператора  $\uparrow$  следует принять, что  $\uparrow < \uparrow$ . Такие отношения гарантируют, что в выражении  $E-E+E$  основой является  $E-E$ , а в  $E\uparrow E\uparrow E$  — последнее подвыражение  $E\uparrow E$ .
3. Для всех операторов  $\theta$  определяем отношения  $\theta < \text{id}$ ,  $\text{id} > \theta$ ,  $\theta < ($ ,  $( < \theta , ) > \theta$ ,  $\theta > )$ ,  $\theta > \$$  и  $\$ < \theta$ . Кроме того, считаем, что

$$\begin{array}{lll} (\doteq) & \$ < ( & \$ < \text{id} \\ (< ( & \text{id} > \$ & ) > \$ \\ (< \text{id} & \text{id} >) & ) > ) \end{array}$$

Эти правила гарантируют, что и **id**, и  $(E)$  будут приведены к  $E$ . Кроме того,  $\$$  служит как левым, так и правым маркером конца строки, что заставляет основы находиться между ними.

### Пример 4.28

На рис. 4.25 приведены отношения приоритета операторов для грамматики (4.17), в предположении, что:

1. оператор  $\uparrow$  имеет высший приоритет и правоассоциативен;
2. операторы \* и / имеют приоритет следующего уровня и левоассоциативны;
3. операторы + и - имеют низший приоритет и левоассоциативны.

Пустые ячейки означают ошибки. Читатель может попытаться проверить корректность этой таблицы, временно игнорируя проблему унарного минуса. Попробуйте рассмотреть, например, входную строку  $\text{id}^*(\text{id}\uparrow\text{id})-\text{id}/\text{id}$ . □

|            | + | - | * | / | $\uparrow$ | id | ( | ) | \$ |
|------------|---|---|---|---|------------|----|---|---|----|
| +          | > | > | < | < | <          | <  | < | > | >  |
| -          | > | > | < | < | <          | <  | < | > | >  |
| *          | > | > | > | > | <          | <  | < | > | >  |
| /          | > | > | > | > | <          | <  | < | > | >  |
| $\uparrow$ | > | > | > | > | <          | <  | < | > | >  |
| id         | > | > | > | > | >          |    |   |   |    |
| (          | < | < | < | < | <          | <  | = |   |    |
| )          | > | > | > | > | >          |    | > |   |    |
| \$         | < | < | < | < | <          | <  | < |   |    |

Рис. 4.25. Отношения приоритета операторов

## Обработка унарных операторов

Если у нас есть унарный оператор типа  $\neg$  (логическое отрицание), который не имеет бинарного аналога, мы можем ввести его в описанную выше схему для создания отношений приоритета операторов. Предполагая, что  $\neg$  — унарный префиксный оператор, мы определяем, что  $\theta < \neg$  для любого оператора  $\theta$ , независимо от того, унарный он или бинарный. Кроме того, мы устанавливаем  $\neg > \theta$ , если  $\neg$  имеет более высокий приоритет, чем  $\theta$ , и  $\neg < \theta$  в противном случае. Например, если  $\neg$  имеет более высокий приоритет, чем  $\&$ , и  $\&$  — левоассоциативный оператор, то в соответствии с приведенными правилами выражение  $E \& \neg E \& E$  должно быть сгруппировано как  $(E \& (\neg E)) \& E$ . Правила для постфиксного унарного оператора аналогичны рассмотренным.

Ситуация изменяется при рассмотрении операторов типа знака “минус”, который может использоваться и как унарный префиксный оператор, и как бинарный инфиксный. Даже если дать им одинаковый приоритет, таблица на рис. 4.25 приведет к ошибке при разборе строки типа  $id^* \cdot id$ . Наилучшим решением этой проблемы было бы распознавание типа оператора лексическим анализатором и использование им разных токенов для бинарного и унарного минусов. К сожалению, лексическому анализатору не достаточно сканируемого символа для принятия решения — ему требуется предыдущий токен. Например, в Fortran знак “минус” унарен, если предшествующим токеном был оператор, левая скобка, запятая или знак присвоения.

## Функции приоритета

Компиляторы, использующие синтаксические анализаторы приоритета операторов, не нуждаются в хранении таблицы отношений приоритетов. В большинстве случаев таблица может быть закодирована двумя *функциями приоритетов* (precedence functions)  $f$  и  $g$ , отображающими терминальные символы в целые числа. Мы пытаемся выбрать  $f$  и  $g$  такими, что для символов  $a$  и  $b$

1.  $f(a) < g(b)$  при  $a < b$
2.  $f(a) = g(b)$  при  $a = b$
3.  $f(a) > g(b)$  при  $a > b$

Таким образом, отношение приоритетов между  $a$  и  $b$  можно определить сравнением числовых значений  $f(a)$  и  $g(b)$ . Заметим, однако, что при этом теряется смысл

пустых ячеек таблицы, информирующих об ошибке, поскольку всегда выполняется одно из трех приведенных условий, независимо от того, чему равны  $f(a)$  и  $g(b)$ . Потеря возможности обнаружения ошибок, вообще говоря, не считается достаточно серьезной, чтобы отказаться от использования функций приоритета там, где это возможно; ошибки могут быть обнаружены при вызове процедуры свертки для необнаруженной основы.

Не всякая таблица отношений приоритетов имеет функции приоритета для ее кодирования, но обычно на практике эти функции существуют.

### Пример 4.29

Таблица приоритетов на рис. 4.25 имеет следующую пару функций приоритетов.

|     | + | - | * | / | $\uparrow$ | ( | ) | $\text{id}$ | \$ |
|-----|---|---|---|---|------------|---|---|-------------|----|
| $f$ | 2 | 2 | 4 | 4 | 4          | 0 | 6 | 6           | 0  |
| $g$ | 1 | 1 | 3 | 3 | 5          | 5 | 0 | 5           | 0  |

Например,  $* < \text{id}$  и  $f(*) < g(\text{id})$ . Заметим, что  $f(\text{id}) > g(\text{id})$  предполагает, что  $\text{id} > \text{id}$ . Однако в действительности между  $\text{id}$  и  $\text{id}$  нет никаких отношений приоритета. Аналогично заменены некоторыми отношениями и остальные пустые ячейки в таблице на рис. 4.25.  $\square$

Далее приведен достаточно простой метод поиска функций приоритетов для таблицы (если таковые существуют).

### Алгоритм 4.6. Построение функций приоритетов

*Вход.* Матрица приоритета операторов.

*Выход.* Функции приоритетов, представляющие входную матрицу, либо сообщение, что такие функции не существуют.

*Метод.*

1. Создаем символы  $f_a$  и  $g_a$  для каждого  $a$ , являющегося термином или \$.
2. Разделим созданные символы на как можно большее число групп таким образом, что если  $a \doteq b$ , то  $f_a$  и  $g_b$  принадлежат одной группе. Возможно, нам придется поместить в одну группу символы, даже не связанные отношением  $\doteq$ . Например, если  $a \doteq b$  и  $c \doteq b$ , то  $f_a$  и  $f_c$  должны находиться в одной и той же группе, поскольку они оба находятся в той же группе, что и  $g_b$ . Если, кроме того,  $c \doteq d$ , то  $f_a$  и  $g_d$  находятся в одной группе, даже если не выполняется условие  $a \doteq d$ .
3. Создадим направленный граф, узлы которого представляют собой группы, найденные в (2). Для всех  $a$  и  $b$ , если  $a < b$ , проведем дугу из группы, в которой находится  $g_b$ , в группу  $f_a$ . Если  $a > b$ , дуга проводится из группы  $f_a$  в  $g_b$ . Отметим, что дуга (или путь) от  $f_a$  к  $g_b$  означает, что  $f(a)$  превосходит  $g(b)$ ; путь от  $g_b$  к  $f_a$  означает, что  $g(b)$  должно превосходить  $f(a)$ .
4. Если построенный в (3) граф имеет циклы, это говорит о том, что функции приоритетов для данной таблицы не существуют. Если циклов нет, то значением  $f(a)$  служит длина самого длинного пути, начинающегося в группе  $f_a$ ; соответственно, значение  $g(a)$  равно длине самого длинного пути из группы  $g_a$ .  $\square$

### Пример 4.30

Рассмотрим матрицу на рис. 4.23. В ней нет отношений  $\doteq$ , и поэтому каждый символ сам по себе представляет группу. На рис. 4.26 показан граф, построенный с использованием алгоритма 4.6.

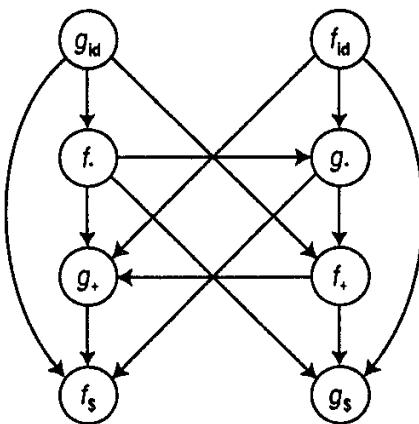


Рис. 4.26. Граф, представляющий функции приоритетов

Этот граф не имеет циклов; следовательно, функции приоритетов существуют. Поскольку  $f\$$  и  $g\$$  не имеют исходящих дуг,  $f(\$)=g(\$)=0$ . Наиболее длинный путь из  $g_+$  имеет длину 1, так что  $g(+)=1$ . Имеется путь  $g_{id} \rightarrow f_* \rightarrow g_* \rightarrow f_* \rightarrow g_+ \rightarrow f_*$ , так что  $g(id)=5$ . Результат построения функций приоритета выглядит следующим образом:

|     | + | * | id | \$ |
|-----|---|---|----|----|
| $f$ | 2 | 4 | 4  | 0  |
| $g$ | 1 | 3 | 5  | 0  |

□

### Восстановление после ошибок при синтаксическом анализе приоритета операторов

Имеются два условия, когда процесс синтаксического анализа может обнаружить синтаксическую ошибку.

- Если между терминалом на вершине стека и текущим входным символом не определено ни одно из отношений приоритета<sup>6</sup>.
- Если найдена основа, но не имеется продукции с этой основой в правой части.

Вспомним, что в алгоритме синтаксического анализа приоритета операторов (алгоритм 4.5) свертываются основы, составленные только из терминалов. Однако хотя нетерминалы и рассматриваются как неразличимые, в стеке синтаксического анализа для них имеются соответствующие места. Таким образом, когда в (2) мы говорим о соответствии основы и правой части продукции, то подразумеваем, что в них должны быть одни и те же терминалы и позиции, занятые нетерминалами.

Мы должны также заметить, что, помимо (1) и (2), других условий обнаружения синтаксических ошибок нет. При сканировании вниз по стеку для поиска левого конца основы в шагах (10–12) алгоритма синтаксического анализа приоритета операторов

<sup>6</sup> В компиляторах, использующих функции приоритета для представления таблиц отношений приоритетов, это условие может оказаться недоступным.

(см. рис. 4.24) мы гарантированно находим отношение  $<$ , поскольку нижним в стеке находится символ  $\$$ , связанный со всеми символами, которые могут быть в стеке непосредственно над ним, отношением  $<$ . Заметим также, что мы не допускаем наличия в стеке соседних символов, не связанных отношениями  $<$  или  $\doteq$ . Следовательно, шаги (10–12) должны успешно выполнять свертку.

То, что последовательность символов  $a < b_1 \doteq b_2 \doteq \dots \doteq b_k$  найдена в стеке, еще не означает, что  $b_1 b_2 \dots b_k$  — строка терминальных символов из правой части некоторой продукции. Мы не проверяем это условие на рис. 4.24, но очевидно, что мы можем сделать это и должны делать, если намерены связать семантические правила со свертками. Таким образом, у нас есть возможность обнаружения ошибок путем внесения изменений в шаги (10–12) на рис. 4.24 для определения того, какая продукция представляет собой основу для свертки.

### Обработка ошибок в процессе свертки

Мы можем разделить программу обнаружения ошибок и восстановления на отдельные части. Одна часть обрабатывает ошибки типа (2). Например, эта программа может снимать символы со стека так же, как и в шагах (10–12) на рис. 4.24, однако при отсутствии продукции для свертки никакие семантические действия не выполняются; вместо этого выводится сообщение об ошибке. Для определения того, что должна выводить диагностика, программа обработки случая (2) должна принять решение, на какую продукцию “похожи” извлекаемые из стека символы. Предположим, например, что со стека снимается  $abc$ , но продукции с такой правой частью нет. Тогда можно попробовать удалить один из символов для получения корректной правой части (нетерминалы опускаем, как неиграющие роли). Так, например, если имеется продукция с правой частью  $aEcE$ , можно вывести сообщение об ошибке типа

“Излишнее  $b$  в строке ...”.

Мы можем также попытаться изменить или вставить терминальный символ в строку; например, при наличии продукции с правой стороной  $abEdc$  сообщение об ошибке может выглядеть как

“Пропущено  $d$  в строке ...”.

Может случиться и так, что последовательность терминалов корректна, но в ней неверно размещены нетерминальные символы. Например, если со стека снимается строка  $abc$  без нетерминалов (окружающих или содержащихся внутри) и при этом  $abc$  не является правой частью продукции, зато таковой является  $aEbC$ , то сообщение об ошибке выглядит как

“Пропущено  $E$  в строке ...”.

В последнем случае  $E$  означает соответствующую синтаксическую категорию, представляющую нетерминалом. Например, если  $a$ ,  $b$  и  $c$  — операторы, то можно говорить о пропущенном выражении, а если  $a$  — ключевое слово типа  $if$  — об условии.

В целом, сложность определения правильной диагностики при обнаружении неверной правой части зависит от того, конечно или бесконечно число строк, которые могут быть сняты со стека при выполнении строк (10–12) на рис. 4.24. Любая такая строка  $b_1 b_2 \dots b_k$  должна иметь отношения  $\doteq$  между соседними символами, т.е.  $b_1 \doteq b_2 \doteq \dots \doteq b_k$ . Если таблица приоритетов операторов говорит, что имеется только конечное количество последовательностей терминалов, связанных отношением  $\doteq$ , то можем обработать такие

строки последовательным просмотром. Для каждой такой строки  $x$  мы можем заранее определить правую часть продукции  $y$  с минимальным расстоянием от данной строки и выводить диагностическое сообщение о том, что найдено  $x$ , в то время как ожидалось  $y$ .

Легко определить все строки, которые могут быть сняты со стека на шагах (10–12). Они становятся очевидны, если построить направленный граф, узлы которого представляют терминалы, с дугами, проведенными от  $a$  к  $b$  тогда и только тогда, когда  $a \doteq b$ . Тогда возможные строки представляют собой просто метки узлов вдоль путей этого графа. Допустимы пути, состоящие из одного узла. Однако чтобы путь  $b_1b_2\dots b_k$  был “съемным со стека” при некоторой входной строке, должен существовать символ  $a$  (возможно,  $\$$ ), такой, что  $a < b_1$ . Назовем этот символ  $b_1$  начальным. Должен также существовать символ  $c$  (возможно,  $\$$ ), такой, что  $b_k > c$ . Назовем этот символ  $b_k$  заключительным. Только при этих условиях может быть вызвана процедура свертки, а  $b_1b_2\dots b_k$  может быть последовательностью символов, снимаемой со стека. Если график имеет путь, содержащий цикл, из начального узла в заключительный, то существует бесконечное число снимаемых со стека строк; в противном случае их количество конечно.

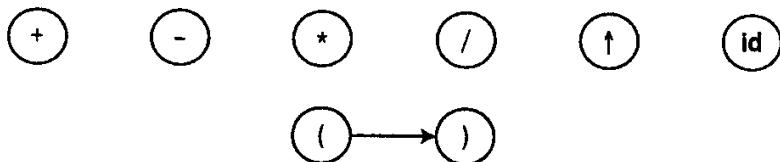


Рис. 4.27. Граф для матрицы приоритетов на рис. 4.25

### Пример 4.31

Вновь обратимся к грамматике (4.17):

$$E \rightarrow E+E \mid E-E \mid E^*E \mid E/E \mid E^\uparrow E \mid (E) \mid -E \mid \text{id}$$

Матрица приоритетов для этой грамматики показана на рис. 4.25, а ее график — на рис. 4.27. Здесь имеется только одна дуга, поскольку только одна пара символов — левая и правая скобки — связана отношением  $\doteq$ . Все символы, кроме правой скобки, являются начальными, и все, кроме левой, — заключительными. Таким образом, все возможные пути единичной длины из начального в конечный узел —  $+$ ,  $-$ ,  $*$ ,  $/$ ,  $\text{id}$  и  $\uparrow$ ; имеется также один путь длиной два — от  $($  к  $)$ . Итак, имеется конечное число путей, каждый из которых соответствует терминалам правой части некоторой продукции грамматики. Таким образом, программа обнаружения ошибки при свертке должна просто проверить корректность множества нетерминалов в сворачиваемой строке терминалов. Конкретно такая программа выполняет следующее.

- Если сворачиваются  $+$ ,  $-$ ,  $*$ ,  $/$  или  $\uparrow$ , она проверяет наличие нетерминалов с обеих сторон символа и при отсутствии какого-либо из них выдает сообщение об отсутствии операнда.
- При свертке  $\text{id}$  она проверяет отсутствие нетерминалов слева и справа. Если имеется хотя бы один из них, программа выдает сообщение об отсутствующем операторе.
- При свертке  $( )$  программа проверяет наличие нетерминала в скобках. Если его нет, она выдает сообщение об отсутствии выражения в скобках.

Кроме того, программа должна проверить, что перед скобками и после них нет нетерминалов. Если это не так, должна выдаваться та же диагностика, что и в случае (2).  $\square$

Если со стека может быть снято бесконечное число строк, табулировать сообщения об ошибках, как мы только что сделали, становится невозможным. В этом случае можно воспользоваться программой для определения того, какая из правых частей продукции ближе всего к нашей строке (например, на расстоянии 1 или 2, где расстояние определяется как количество токенов, которое следует вставить, удалить или заменить для получения искомой строки), и вывести сообщение о предполагаемой корректной строке. Если такой близкой продукции нет, то выводится сообщение, что в этой строке что-то не так...

## Обработка ошибок переноса/свертки

Теперь рассмотрим другой путь обнаружения ошибок синтаксическим анализатором приоритета операторов. При обращении к матрице приоритетов для принятия решения о переносе или свертке (строки (6) и (9) на рис. 4.24) мы можем обнаружить, что между символом на вершине стека и очередным входным символом отношения отсутствуют. Предположим, что на вершине стека находятся  $a$  и  $b$  ( $b$  на вершине стека), во входном потоке —  $c$  и  $d$  ( $c$  — первый), а между  $b$  и  $c$  нет отношений приоритетов. Для восстановления после этой ошибки нужно изменить стек или входной буфер (а может быть, и тот, и другой). Мы можем изменить символы, вставить их в стек или входной буфер или удалить оттуда. При вставке и изменении следует быть предельно осторожным, чтобы избежать бесконечного цикла, в котором мы, например, постоянно вставляем в буфер символы, перенести или свернуть которые невозможно.

Один из способов гарантировать отсутствие зацикливания — это обеспечить возможность переноса символа после восстановления (если текущий символ —  $\$$ , следует убедиться, что во входной поток не вставляются новые символы, а стек в конце концов сокращается). Например, при  $ab$  в стеке и  $cd$  во входной строке, если  $a \leq c$ <sup>7</sup>, мы можем снять со стека  $b$ . Другой способ — удаление из входного потока  $c$  при  $b \leq d$ . Третий способ заключается в поиске такого  $e$ , что  $b \leq e \leq c$ , и вставке  $e$  во входной поток перед  $c$ . В более общем виде, если единственный символ для вставки найти не удается, мы можем вставить строку символов, такую, что  $b \leq e_1 \leq e_2 \leq \dots \leq e_n \leq c$ . Выбор конкретного действия должен в конечном счете определяться интуицией разработчика компилятора.

Для каждой пустой ячейки матрицы приоритетов мы должны определить подпрограмму восстановления после ошибок; одна и та же подпрограмма может использоваться в нескольких местах. Когда синтаксический анализатор рассматривает запись для  $a$  и  $b$  на шаге (6) (рис. 4.24) и не обнаруживает отношений приоритета между  $a$  и  $b$ , он находит указатель на подпрограмму восстановления после этой ошибки.

### Пример 4.32

Вновь обратимся к матрице приоритетов, представленной на рис. 4.25. На рис. 4.28 мы показали только те строки и столбцы матрицы, в которых имеются пустые ячейки (теперь они заполнены именами подпрограмм восстановления после ошибки).

|    | id   | (    | )        | \$   |
|----|------|------|----------|------|
| id | $e3$ | $e3$ | $>$      | $>$  |
| (  | $<$  | $<$  | $\doteq$ | $e4$ |
| )  | $e3$ | $e3$ | $>$      | $>$  |
| \$ | $<$  | $<$  | $e2$     | $e1$ |

Рис. 4.28. Матрица приоритета операторов с подпрограммами обработки ошибок

<sup>7</sup> Символ  $\leq$  означает “ $<$  или  $\doteq$ ”.

Сущность этих подпрограмм состоит в следующем.

- e1: /\* Вызывается при отсутствии выражения в целом \*/  
Вставляет **id** во входной поток  
Диагностическое сообщение: *отсутствует операнд*
- e2: /\* Вызывается, когда выражение начинается с правой скобки \*/  
Удалить ")" из входного потока  
Диагностическое сообщение: *несбалансированная правая скобка*
- e3: /\* Вызывается, когда **id** или ) следует за **id** или ( \*/  
Вставляет во входной поток +  
Диагностическое сообщение: *отсутствует оператор*
- e4: /\* Вызывается при завершении выражения левой скобкой \*/  
Снимает ( со стека  
Диагностическое сообщение: *отсутствует правая скобка*

Рассмотрим работу этого механизма с неверной входной строкой **id+**). Первые действия, предпринимаемые синтаксическим анализатором — перенос **id**, свертка в **E** (мы вновь используем **E** как безымянный нетерминал в стеке) и перенос +. После этих действий получаем следующую конфигурацию.

| СТЕК | ВХОД |
|------|------|
| \$E+ | )\$  |

Поскольку + >), вызывается свертка для основы +. Подпрограмма проверки ошибок при свертке должна просмотреть наличие **E** слева и справа от оператора. Обнаружив, что один из них отсутствует, она выдает сообщение об ошибке “*отсутствует операнд*” и все равно выполняет свертку.

Теперь конфигурация принимает вид

| СТЕК | ВХОД |
|------|------|
| \$E  | )\$  |

Между \$ и ) нет отношений приоритетов, и запись на рис. 4.28 для этой пары символов — e2. Подпрограмма e2 выводит диагностическое сообщение “*несбалансированная правая скобка*” и удаляет правую скобку из входного потока. При этом мы достигаем конечной конфигурации синтаксического анализатора

| СТЕК | ВХОД |
|------|------|
| \$E  | \$   |

□

## 4.7. LR-анализаторы

В этом разделе представлена эффективная технология восходящего синтаксического анализа, которая может использоваться для анализа большого класса контекстно-свободных грамматик. Эта технология называется LR(*k*)-анализ; L означает сканирование входного потока слева направо, R — построение обращенных правых порождений, а *k* — число входных символов, которые могут быть просмотрены для принятия решения о способе проведения разбора. Если (*k*) опущено, подразумевается, что *k* равно 1. LR-анализ весьма привлекателен по множеству причин.

- LR-анализаторы могут быть созданы для распознавания, по сути, всех конструкций языков программирования, для которых может быть написана контекстно-свободная грамматика.
- Метод LR-анализа — наиболее общий известный метод ПС-анализа без отката, который, кроме того, не уступает в эффективности другим методам этого типа.
- Класс грамматик, которые могут быть разобраны с использованием LR-методов, представляет собой собственное надмножество класса грамматик, которые могут быть разобраны предиктивными синтаксическими анализаторами.
- LR-анализатор может обнаруживать синтаксические ошибки сразу же, как только это становится возможным при сканировании входного потока.

Основной недостаток этого метода состоит в том, что ручное построение LR-анализатора для грамматики типичного языка программирования требует большого объема работы. Для решения этой задачи нужен специализированный инструмент — генератор LR-анализаторов. К счастью, имеется множество таких генераторов, и мы рассмотрим один из них, Yacc, в разделе 4.9. С помощью такого генератора для разработанной контекстно-свободной грамматики можно автоматически построить ее синтаксический анализатор. Если грамматика содержит неоднозначности или другие конструкции, трудные для разбора сканированием слева направо, генератор в состоянии их локализовать и сообщить о них разработчику.

После обсуждения работы LR-анализатора познакомимся с тремя методами построения таблицы LR-анализа для грамматики. Первый метод, простой LR (simple LR, SLR), является самым легким в реализации, но наименее мощным из них. Он не может построить таблицу разбора для некоторых грамматик, которые успешно обрабатываются другими методами. Второй метод — канонический LR — наиболее мощный, но требующий наибольших ресурсов. Третий метод, метод LR с предпросмотром (lookahead LR, или LALR), по мощности и требуемым ресурсам занимает промежуточное положение. Метод LALR работает с большинством грамматик и может быть эффективно реализован. В этом разделе мы рассмотрим некоторые методы сжатия таблиц LR-анализа.

## Алгоритм LR-анализа

Схематически LR-анализатор представлен на рис. 4.29. Он состоит из входного потока, выхода, стека, управляющей программы и таблицы синтаксического анализа, состоящей из двух частей (*действие* (*action*) и *переход* (*goto*)). Управляющая программа для всех LR-анализаторов одна и та же; изменяются только таблицы синтаксического анализа. Программа синтаксического анализа считывает символы из входного буфера по одному и использует стек для хранения строк вида  $s_0 X_1 s_1 X_2 s_2 \cdots X_m s_m$  ( $s_m$  находится на вершине стека). Каждый символ  $X_i$  является символом грамматики, а каждый  $s_i$  — символом, именуемым *состоянием*. Каждый символ состояния обобщает информацию, содержащуюся в стеке ниже его. Комбинация символа состояния на вершине стека и текущего входного символа используется в качестве индекса таблицы синтаксического анализа и определяет дальнейшее действие — перенос или свертку. При реализации грамматические символы не обязаны появляться в стеке; однако мы всегда будем рассматривать их для облегчения понимания принципов работы LR-анализатора.

Таблица синтаксического анализа состоит из двух частей — функции действий синтаксического анализа *action* и функции переходов *goto*. Управляющая программа LR-

анализатора функционирует следующим образом. Она определяет  $s_m$ , текущее состояние на вершине стека, и  $a_i$ , текущий входной символ. Затем программа обращается к  $action[s_m, a_i]$ , ячейке таблицы действий синтаксического анализа, определяемой состоянием  $s_m$  и символом  $a_i$ , которая может иметь одно из четырех значений:

- 1) перенос  $s$ , где  $s$  — состояние;
- 2) свертка в соответствии с продукцией  $A \rightarrow \beta$ ;
- 3) допуск;
- 4) ошибка.

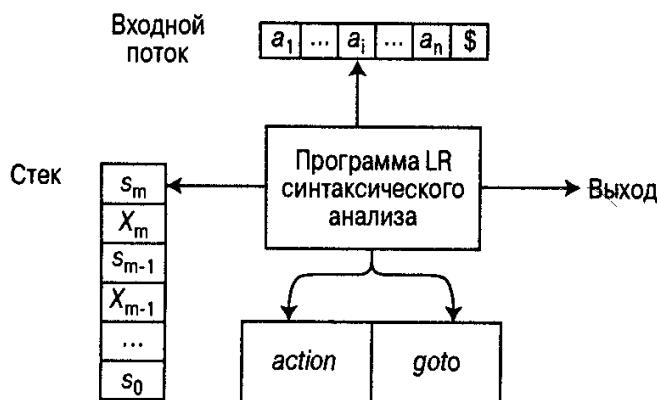


Рис. 4.29. Модель LR-анализатора

Функция  $goto$  получает в качестве аргументов состояние и символ грамматики и возвращает новое состояние. Мы увидим, что функция  $goto$  таблицы синтаксического анализа, построенная на основе грамматики  $G$  с использованием SLR-, канонического LR- или LALR-метода, представляет собой функцию переходов детерминированного конечного автомата, распознающего активные префиксы  $G$ . Начальным состоянием этого ДКА является состояние, изначально размещаемое на вершине стека LR-анализатора.

**Конфигурация** LR-анализатора представляет собой пару, первый компонент которой — содержимое стека, а второй — непросмотренная часть входного потока:  $(s_0 X_1 X_2 s_2 \cdots X_m s_m, a_i a_{i+1} \cdots a_n \$)$ . Эта конфигурация представляет правосентенциальную форму  $X_1 X_2 \cdots X_m a_i a_{i+1} \cdots a_n$ , по сути, тем же способом, что и ПС-анализатор; новым является только наличие в стеке состояний.

Следующий шаг синтаксического анализатора определяется текущим входным символом  $a_i$  и состоянием на вершине стека  $s_m$  в соответствии со значением ячейки таблицы  $action[s_m, a_i]$ . Конфигурации, получаемые после каждого из четырех типов действий, следующие.

1. Если  $action[s_m, a_i] = \text{"перенос } s\text{"}$ , синтаксический анализатор выполняет перенос, переходя в конфигурацию  $(s_0 X_1 X_2 s_2 \cdots X_m s_m a_i s, a_{i+1} \cdots a_n \$)$ . Синтаксический анализатор переносит в стек текущий входной символ  $a_i$  и очередное состояние  $s$ , определяемое значением  $action[s_m, a_i]$ ; текущим входным символом становится  $a_{i+1}$ .
2. Если  $action[s_m, a_i] = \text{"свертка } A \rightarrow \beta\text{"}$ , то синтаксический анализатор выполняет свертку, переходя в конфигурацию  $(s_0 X_1 X_2 s_2 \cdots X_{m-r} s_{m-r} As, a_i a_{i+1} \cdots a_n \$)$ , где  $s = goto[s_{m-r}, A]$ , а  $r$  — длина  $\beta$ , правой части продукции. Здесь синтаксический анализатор вначале снимает со стека  $2r$  символов ( $r$  символов состояний и  $r$  символов грамматики), выводя на вершину стека состояние  $s_{m-r}$ . Затем он вносит в стек  $A$

(левую часть продукции) и  $s$ , запись из ячейки  $goto[s_{m-r}, A]$ . Текущий входной символ при этом не изменяется. Последовательность снимаемых со стека символов грамматики  $X_{m-r+1} \dots X_m$  всегда соответствует  $\beta$ , правой части продукции свертки.

3. Если  $action[s_m, a_i] = \text{"допуск"}$ , синтаксический анализ завершается.
4. Если  $action[s_m, a_i] = \text{"ошибка"}$ , синтаксический анализатор обнаружил ошибку и вызывает подпрограмму восстановления после нее.

Полностью алгоритм LR-анализа приведен ниже. Все LR-анализаторы ведут себя одинаково; единственная разница между ними заключается в таблицах  $action$  и  $goto$ .

#### Алгоритм 4.7. Алгоритм LR-анализа

**Вход.** Входная строка  $w$  и таблица LR-анализа с функциями  $action$  и  $goto$  для грамматики  $G$ .

**Выход.** Если  $w \in L(G)$ , выдается восходящий разбор для  $w$ ; в противном случае выводится сообщение об ошибке.

**Метод.** Изначально синтаксический анализатор содержит в стеке начальное состояние  $s_0$ , а во входном буфере —  $w\$$ . Затем анализатор выполняет приведенную на рис. 4.30 программу до тех пор, пока не будет достигнуто успешное завершение анализа или не обнаружится ошибка.  $\square$

Установить указатель  $ip$  на первый символ  $w\$$ ;

**repeat forever begin**

Пусть  $s$  — состояние на вершине стека, а  $a$  — символ, на который указывает  $ip$

**if**  $action[s, a] = \text{"перенос } s\text{" then begin}$

Поместить в стек  $a$ , затем  $s'$ ; переместить  $ip$  к следующему входному символу.

**end**

**else if**  $action[s, a] = \text{"свертка } A \rightarrow \beta\text{" then begin}$

Снять со стека  $2 * |\beta|$  символов. Пусть  $s'$  — текущее состояние на вершине стека. Поместить в стек  $A$ , затем  $goto[s', A]$ ; вывести продукцию  $A \rightarrow \beta$ .

**end**

**else if**  $action[s, a] = \text{"допуск" then}$

**return**

**else error()**

**end**

Рис. 4.30. Программа LR-анализа

| СОСТОЯНИЕ | action |    |    |   |    |     | goto |   |   |
|-----------|--------|----|----|---|----|-----|------|---|---|
|           | id     | +  | *  | ( | )  | \$  | E    | T | F |
| 0         | s5     |    |    |   | s4 |     | 1    | 2 | 3 |
| 1         |        | s6 |    |   |    | acc |      |   |   |
| 2         |        | r2 | s7 |   | r2 | r2  |      |   |   |

Рис. 4.31. Таблица синтаксического анализа грамматики выражений

| Состояние | action |    |    |    |     |    | goto |   |    |
|-----------|--------|----|----|----|-----|----|------|---|----|
|           | id     | +  | *  | (  | )   | \$ | E    | T | F  |
| 3         |        | r4 | r4 |    | r4  | r4 |      |   |    |
| 4         | s5     |    |    | s4 |     |    | 8    | 2 | 3  |
| 5         |        | r6 | r6 |    | r6  | r6 |      |   |    |
| 6         | s5     |    |    | s4 |     |    |      | 9 | 3  |
| 7         | s5     |    |    | s4 |     |    |      |   | 10 |
| 8         |        | s6 |    |    | s11 |    |      |   |    |
| 9         |        | r1 | s7 |    | r1  | r1 |      |   |    |
| 10        |        | r3 | r3 |    | r3  | r3 |      |   |    |
| 11        |        | r5 | r5 |    | r5  | r5 |      |   |    |

Рис. 4.31. Таблица синтаксического анализа грамматики выражений (окончание)

### Пример 4.33

На рис. 4.31 показаны функции синтаксического анализа *action* и *goto* для следующей грамматики арифметических выражений с бинарными операторами + и \*.

- (1)  $E \rightarrow E+T$
- (2)  $E \rightarrow T$
- (3)  $T \rightarrow T^*F$
- (4)  $T \rightarrow F$
- (5)  $F \rightarrow (E)$
- (6)  $F \rightarrow \text{id}$

На рисунке использованы следующие коды действий:

1.  $s_i$  означает перенос и  $i$ -е состояние на вершине стека,
2.  $r_j$  означает свертку в соответствии с продукцией номер  $j$ ,
3.  $\text{acc}$  означает допуск входной строки,
4. пустая ячейка означает ошибку.

Заметим, что значение  $goto[s, a]$  для терминала  $a$  находится в поле *action*, связанном с переносом для входного символа  $a$  и состояния  $s$ . Поле *goto* дает значения  $goto[s, A]$  для нетерминалов  $A$ . Кроме того, следует иметь в виду, что мы пока не пояснили, каким образом выбираются записи на рис. 4.31 — об этом поговорим немного позже.

Для входной строки  $\text{id}^*\text{id}+\text{id}$  последовательность содержимого стека и входной строки показана на рис. 4.32. Например, в строке (1) LR-анализатор находится в состоянии 0 с первым входным символом  $\text{id}$ . Действие в строке 0 и столбце *id* таблицы *action* на рис. 4.31 —  $s5$ , означающее перенос и внесение в стек состояния 5. В строке (2) выполняется внесение в стек  $\text{id}$  и  $s5$  и удаление  $\text{id}$  из входного потока.

После этого текущим входным символом становится \*; действие для состояния  $s5$  и входного символа \* —  $r6$ , т.е. свертка согласно продукции  $F \rightarrow \text{id}$ . Со стека при этом снимаются два символа (символ состояния и символ грамматики), и на вершине стека появляется состояние 0. Поскольку  $goto[0, F]$  равно  $s3$ , в стек вносятся  $F$  и 3; получается конфигурация, показанная в строке (3). Остальные строки на рис. 4.32 получены аналогично.  $\square$

| СТЕК                                             | ВХОДНОЙ ПОТОК                          | ДЕЙСТВИЕ                             |
|--------------------------------------------------|----------------------------------------|--------------------------------------|
| (1) 0                                            | $\text{id} * \text{id} + \text{id} \$$ | Перенос                              |
| (2) 0 <u><math>\text{id}</math></u> 5            | * $\text{id} + \text{id} \$$           | Свертка по $F \rightarrow \text{id}$ |
| (3) 0 $F$ 3                                      | * $\text{id} + \text{id} \$$           | Свертка по $T \rightarrow F$         |
| (4) 0 $T$ 2                                      | * $\text{id} + \text{id} \$$           | Перенос                              |
| (5) 0 $T$ 2 * 7                                  | $\text{id} + \text{id} \$$             | Перенос                              |
| (6) 0 $T$ 2 * 7 <u><math>\text{id}</math></u> 5  | + $\text{id} \$$                       | Свертка по $F \rightarrow \text{id}$ |
| (7) 0 $T$ 2 * 7 $F$ 10                           | + $\text{id} \$$                       | Свертка по $T \rightarrow T^*F$      |
| (8) 0 $T$ 2                                      | + $\text{id} \$$                       | Свертка по $E \rightarrow T$         |
| (9) 0 $E$ 1                                      | + $\text{id} \$$                       | Перенос                              |
| (10) 0 $E$ 1 + 6                                 | $\text{id} \$$                         | Перенос                              |
| (11) 0 $E$ 1 + 6 <u><math>\text{id}</math></u> 5 | \$                                     | Свертка по $F \rightarrow \text{id}$ |
| (12) 0 $E$ 1 + 6 $F$ 3                           | \$                                     | Свертка по $T \rightarrow F$         |
| (13) 0 $E$ 1 + 6 $T$ 9                           | \$                                     | $E \rightarrow E+T$                  |
| (14) 0 $E$ 1                                     | \$                                     | Допуск                               |

Рис. 4.32. Действия LR-анализатора по разбору строки  $\text{id}*\text{id}+\text{id}$

## LR-грамматики

Каким образом строится таблица LR-анализа для данной грамматики? Грамматика, для которой мы можем построить таблицу синтаксического анализа, называется *LR-грамматикой*. Существуют контекстно-свободные грамматики, не являющиеся LR-грамматиками, но обычно для типичных конструкций языков программирования их можно избежать. Интуитивно, для того чтобы грамматика была LR-грамматикой, достаточно, чтобы ПС-анализатор, читающий поток слева направо, был способен распознавать основы при их появлении в стеке.

LR-анализатор не должен сканировать весь стек, чтобы распознать появление основы на вершине стека; более того, символ состояния на вершине стека содержит всю необходимую информацию. Примечательно, что если можно распознать основу, зная только символы грамматики в стеке, то существует конечный автомат, который в состоянии определить основу на вершине стека (если таковая имеется) путем считывания грамматических символов в стеке сверху вниз. По сути, таким автоматом является функция *goto* таблицы LR-анализа. Однако автомату также не нужно читать стек при каждом переходе. Символ состояния на вершине стека представляет собой состояние распознавающего основы конечного автомата, в котором бы он оказался после считывания грамматических символов в стеке снизу вверх. Следовательно, LR-анализатор может определить по состоянию на вершине стека все, что надо знать о содержимом стека.

Другой источник информации, которым может воспользоваться LR-анализатор для принятия решения о переносе/свертке, — очередные  $k$  символов входного потока. Практический интерес представляют случаи  $k = 0$  и  $k = 1$ , так что здесь будем рассматривать только LR-анализаторы с  $k \leq 1$ . Например, в таблице *action* на рис. 4.31 используется только один просматриваемый символ. Грамматика, для разбора которой LR-анализатору требуется просмотр до  $k$  символов входного потока на каждом шаге, называется *LR( $k$ )-грамматикой*.

Существует важное отличие между LL- и LR-грамматиками. Для того чтобы грамматика представляла собой LR( $k$ )-грамматику, мы должны суметь распознать появление правой части продукции, видя все, что из нее порождено, а также  $k$  входных символов. Это требование существенно менее строго, чем требование LL( $k$ )-грамматики, где нужно распознать использование продукции, видя только  $k$  первых символов того, что порождено ее правой частью. Поэтому LR-грамматики могут описывать больше языков, чем LL-грамматики.

## Построение таблиц SLR-анализа

Сейчас мы покажем, каким образом можно построить таблицу LR-анализа для данной грамматики. Мы представим три метода, разные по мощности и сложности реализации. Первый метод — “простого LR” (simple LR, SLR) — самый слабый из них по количеству грамматик, с которыми он работает, однако самый простой в реализации. Таблицу, построенную таким методом, будем называть SLR-таблицей, а синтаксический анализатор, работающий с SLR-таблицей, — SLR-анализатором (а соответствующую грамматику — SLR-грамматикой). Два других метода добавляют к SLR-методу просмотр входного потока, так что SLR-метод — хорошее начало для изучения LR-разбора.

$LR(0)$ -пункт, или элемент (далее для краткости — просто *пункт*<sup>8</sup>) грамматики  $G$  — продукция  $G$  с точкой в некоторой позиции правой части. Следовательно, продукция  $A \rightarrow XYZ$  дает четыре пункта:

$$\begin{array}{ll} A \rightarrow & \cdot XYZ \\ A \rightarrow & X \cdot YZ \\ A \rightarrow & XY \cdot Z \\ A \rightarrow & XYZ \cdot \end{array}$$

Продукция  $A \rightarrow \epsilon$  генерирует только один пункт,  $A \rightarrow \cdot$ . Пункт может быть представлен парой целых чисел, первое из которых представляет номер продукции, а второе — позицию точки. Интуитивно пункт указывает, какую часть продукции мы уже увидели в данной точке в процессе синтаксического анализа. Например, первый пункт, приведенный выше, определяет, что во входном потоке мы ожидаем встретить строку, порожденную  $XYZ$ . Второй пункт указывает, что у нас уже есть строка, порожденная  $X$ , и мы ожидаем получить из входного потока строку, порожденную  $YZ$ .

Основная идея SLR-метода состоит в том, чтобы вначале построить на базе грамматики детерминированный конечный автомат для распознавания активных префиксов. Мы группируем пункты в множества, которые приводят к состояниям SLR-анализатора. Пункты могут рассматриваться как состояния недетерминированного конечного автомата, распознающего активные префиксы, и тогда группирование представляет собой не что иное, как построение подмножества из раздела 3.6.

Система  $LR(0)$ -пунктов, которую назовем *канонической*, обеспечивает основу для построения SLR-анализаторов. Для построения канонической  $LR(0)$ -системы грамматики мы определим расширенную грамматику и две функции — *closure* и *goto*.

Если  $G$  — грамматика со стартовым символом  $S$ , то  $G'$ , *расширенная грамматика* (*augmented grammar*) грамматики  $G$ , представляет собой  $G$  с новым стартовым символом  $S'$  и продукцией  $S' \rightarrow S$ . Назначение этой новой стартовой продукции — указать

<sup>8</sup> В оригинале —  $LR(0)$ -item. В русскоязычной литературе иногда использовался термин “ситуация”. Для улучшения читаемости текста  $LR(0)$ -пункты иногда записываются в квадратных скобках. — Прим. ред.

синтаксическому анализатору, когда он должен прекратить разбор и объявить о допущении входной строки. Таким образом, допуск строки происходит тогда и только тогда, когда синтаксический анализатор выполняет свертку, соответствующую продукции  $S' \rightarrow S$ .

## Операция замыкания

Если  $I$  — множество пунктов грамматики  $G$ , то  $\text{closure}(I)$  — множество пунктов, построенное из  $I$  по следующим правилам.

1. Изначально в  $\text{closure}(I)$  входят все пункты из  $I$ .
2. Если  $A \rightarrow \alpha \cdot B\beta$  входит в  $\text{closure}(I)$  и  $B \rightarrow \gamma$  представляет собой продукцию, то добавляем в  $\text{closure}(I)$  пункт  $B \rightarrow \cdot \gamma$  (если его там еще нет). Мы применяем это правило до тех пор, пока не внесем все возможные пункты в  $\text{closure}(I)$ .

Наличие  $A \rightarrow \alpha \cdot B\beta$  в  $\text{closure}(I)$  указывает, что в некоторый момент в процессе синтаксического анализа мы полагаем, что можем встретить во входном потоке подстроку, выводимую из  $B\beta$ . Но если имеется продукция  $B \rightarrow \gamma$ , то, естественно, мы также можем встретить в этот момент строку, выводимую из  $\gamma$ , поэтому включаем  $B \rightarrow \cdot \gamma$  в  $\text{closure}(I)$ .

### Пример 4.34

Рассмотрим расширенную грамматику арифметических выражений:

$$\begin{array}{lcl} E' & \rightarrow & E \\ E & \rightarrow & E+T \mid T \\ T & \rightarrow & T^*F \mid F \\ F & \rightarrow & (E) \mid \text{id} \end{array} \quad (4.19)$$

Если  $I$  представляет собой множество из одного пункта  $\{[E' \rightarrow \cdot E]\}$ , то  $\text{closure}(I)$  содержит пункты

$$\begin{array}{lcl} E' & \rightarrow & \cdot E \\ E & \rightarrow & \cdot E+T \\ E & \rightarrow & \cdot T \\ T & \rightarrow & \cdot T^*F \\ T & \rightarrow & \cdot F \\ F & \rightarrow & \cdot (E) \\ F & \rightarrow & \cdot \text{id} \end{array}$$

Здесь  $E' \rightarrow \cdot E$  помещается в  $\text{closure}(I)$  в соответствии с правилом (1). Поскольку  $E$  расположено непосредственно за точкой, согласно правилу (2) добавляются также  $E$ -продукции с точкой слева, т.е.  $E \rightarrow \cdot E+T$  и  $E \rightarrow \cdot T$ . Теперь, поскольку среди пунктов имеется  $T$ , следующее за точкой, мы добавляем  $T \rightarrow \cdot T^*F$  и  $T \rightarrow \cdot F$  и, аналогично,  $F \rightarrow \cdot (E)$  и  $F \rightarrow \cdot \text{id}$ . Больше добавлять в  $\text{closure}(I)$  нечего.  $\square$

Функция  $\text{closure}$  может быть вычислена, как показано на рис. 4.33. При реализации вычислений удобно ввести массив булевых величин  $\text{added}$ , проиндексированный нетерминалами  $G$  так, что  $\text{added}[B]$  равно  $\text{true}$ , если мы добавляем пункты  $B \rightarrow \cdot \gamma$  для каждой  $B$ -продукции  $B \rightarrow \gamma$ .

```

function closure(I);
begin
 J := I;
 repeat
 for каждый элемент $A \rightarrow \alpha \cdot B\beta$ из J и каждая продукция
 $B \rightarrow \gamma$ грамматики G , такая, что $B \rightarrow \gamma$ не входит в J do
 добавить $B \rightarrow \gamma$ в J
 until больше добавлять в J нечего;
 return J
end

```

Рис. 4.33. Вычисление функции closure

Заметим, если одна из  $B$ -продукций добавляется в  $\text{closure}(I)$  с точкой слева, то к замыканию будут добавлены все  $B$ -продукции, так что на самом деле в некоторых случаях достаточно указать список добавленных нетерминалов, а не список всех продуктов. Это приводит к тому, что мы можем разделить любое множество интересующих нас пунктов на два класса.

1. *Базисные* (kernel) *пункты*, или *пункты ядра* (kernel items), включающие начальный пункт  $S' \rightarrow \cdot S$ , и все пункты, у которых точки расположены не у левого края.
2. *Небазисные* (nonkernel) *пункты*, у которых точки расположены слева.

Более того, каждое множество интересующих нас пунктов формируется как замыкание множества базисных пунктов; добавляемые в замыкание пункты не могут быть базисными. Таким образом, мы можем представить множества интересующих нас пунктов с использованием очень небольшого объема памяти, если отбросим все небазисные пункты, зная, что они могут быть восстановлены процессом замыкания.

## Операция goto

Второй полезной функцией является  $\text{goto}(I, X)$ , где  $I$  является множеством пунктов, а  $X$  — символом грамматики.  $\text{goto}(I, X)$  определяется как замыкание множества всех пунктов  $[A \rightarrow \alpha X \cdot \beta]$ , таких, что  $[A \rightarrow \alpha \cdot X \beta] \in I$ . Интуитивно, если  $I$  является множеством пунктов, допустимых для некоторого активного префикса  $\gamma$ , то  $\text{goto}(I, X)$  есть множество пунктов, допустимых для активного префикса  $\gamma X$ .

### Пример 4.35

Если  $I$  представляет собой множество из двух пунктов  $\{[E' \rightarrow E \cdot], [E \rightarrow E \cdot + T]\}$ , то  $\text{goto}(I, +)$  состоит из

$E \rightarrow E \cdot + T$   
 $E \rightarrow \cdot T^* F$   
 $E \rightarrow \cdot F$   
 $E \rightarrow \cdot (E)$   
 $E \rightarrow \cdot \text{id}$

Мы вычислили  $\text{goto}(I, +)$ , рассмотрев пункты из  $I$ , у которых сразу за точкой идет  $+$ . Таким пунктом является  $E \rightarrow E \cdot + T$ , в отличие от пункта  $E' \rightarrow E \cdot$ . Мы перемещаем точку за символ  $+$ , получая  $\{E \rightarrow E \cdot + T\}$ , и рассматриваем замыкание этого множества.  $\square$

## Построение множеств пунктов

Теперь мы готовы представить алгоритм для построения  $C$  — канонической системы множеств LR(0)-пунктов для расширенной грамматики  $G'$ . Этот алгоритм показан на рис. 4.34.

```

procedure items(G');
begin
 $C := \{closure(\{[S' \rightarrow \cdot S]\})\};$
 repeat
 for каждое множество пунктов I в C и каждый
 символ грамматики X , такой, что $goto(I, X)$
 не является пустым и не принадлежит C do
 добавить $goto(I, X)$ к C
 until больше нет множеств, которые можно
 добавить к C
end

```

Рис. 4.34. Построение множеств пунктов

## Пример 4.36

Каноническая система множеств LR(0)-пунктов для грамматики (4.19) из примера 4.34 показана на рис. 4.35, а функция  $goto$  в виде диаграммы переходов детерминированного конечного автомата — на рис. 4.36.  $\square$

|        |                           |                             |                         |                             |                         |                           |                                 |                             |                                 |                           |
|--------|---------------------------|-----------------------------|-------------------------|-----------------------------|-------------------------|---------------------------|---------------------------------|-----------------------------|---------------------------------|---------------------------|
| $I_0:$ | $E' \rightarrow \cdot E$  | $E \rightarrow \cdot E + T$ | $E \rightarrow \cdot T$ | $T \rightarrow \cdot T^* F$ | $T \rightarrow \cdot F$ | $F \rightarrow \cdot (E)$ | $F \rightarrow \cdot \text{id}$ | $I_5:$                      | $F \rightarrow \text{id} \cdot$ |                           |
|        |                           |                             |                         |                             |                         |                           |                                 | $I_6:$                      | $E \rightarrow E \cdot + T$     |                           |
|        |                           |                             |                         |                             |                         |                           |                                 |                             | $T \rightarrow \cdot T^* F$     |                           |
|        |                           |                             |                         |                             |                         |                           |                                 |                             | $T \rightarrow \cdot F$         |                           |
|        |                           |                             |                         |                             |                         |                           |                                 |                             | $F \rightarrow \cdot (E)$       |                           |
|        |                           |                             |                         |                             |                         |                           |                                 |                             | $F \rightarrow \cdot \text{id}$ |                           |
| $I_1:$ | $E' \rightarrow E \cdot$  | $E \rightarrow E \cdot + T$ |                         |                             |                         |                           |                                 | $I_7:$                      | $T \rightarrow T^* \cdot F$     |                           |
|        |                           |                             |                         |                             |                         |                           |                                 |                             | $F \rightarrow \cdot (E)$       |                           |
|        |                           |                             |                         |                             |                         |                           |                                 |                             | $F \rightarrow \cdot \text{id}$ |                           |
| $I_2:$ | $E \rightarrow T \cdot$   | $T \rightarrow T^* F$       |                         |                             |                         |                           |                                 | $I_8:$                      | $F \rightarrow (E) \cdot$       |                           |
|        |                           |                             |                         |                             |                         |                           |                                 |                             | $E \rightarrow E \cdot + T$     |                           |
| $I_3:$ | $T \rightarrow F \cdot$   |                             |                         |                             |                         |                           |                                 | $I_9:$                      | $E \rightarrow E + T \cdot$     |                           |
| $I_4:$ | $F \rightarrow (\cdot E)$ | $E \rightarrow \cdot E + T$ | $E \rightarrow \cdot T$ | $T \rightarrow \cdot T^* F$ | $T \rightarrow \cdot F$ | $F \rightarrow \cdot (E)$ | $F \rightarrow \cdot \text{id}$ | $T \rightarrow T^* F \cdot$ |                                 |                           |
|        |                           |                             |                         |                             |                         |                           |                                 |                             |                                 |                           |
|        |                           |                             |                         |                             |                         |                           |                                 | $I_{10}:$                   | $T \rightarrow T^* F \cdot$     |                           |
|        |                           |                             |                         |                             |                         |                           |                                 |                             | $I_{11}:$                       | $F \rightarrow (E) \cdot$ |
|        |                           |                             |                         |                             |                         |                           |                                 |                             |                                 |                           |

Рис. 4.35. Каноническая LR(0)-система для грамматики (4.19)

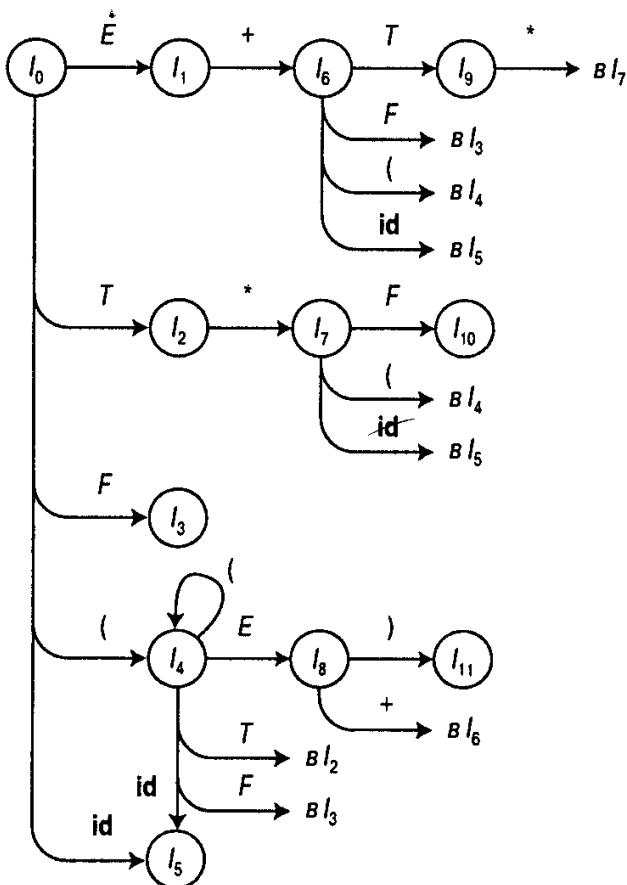


Рис. 4.36. Диаграмма переходов детерминированного конечного автомата  $D$  для активных префиксов

Если каждое состояние автомата  $D$  (рис. 4.36) является заключительным, а  $I_0$  — начальным состоянием, то  $D$  распознает в точности активные префиксы грамматики (4.19), и это не случайно. Для каждой грамматики  $G$  функция  $goto$  канонической системы множеств пунктов определяет детерминированный конечный автомат, распознающий активные префиксы  $G$ . Можно представить себе недетерминированный конечный автомат  $N$ , состояниями которого являются пункты. Имеется переход из  $A \rightarrow \alpha \cdot X \beta$  в  $A \rightarrow \alpha X \cdot \beta$ , помеченный  $X$ , и переход из  $A \rightarrow \alpha \cdot B \beta$  в  $B \rightarrow \cdot \gamma$ , помеченный  $\epsilon$ . Тогда  $closure(I)$  для множества пунктов (состояний  $N$ )  $I$  представляет собой  $\epsilon$ -closure множества состояний НКА, определенное в разделе 3.6. Следовательно, значением  $goto(I, X)$  является переход из  $I$  по символу  $X$  в ДКА, построенным из  $N$  по алгоритму построения подмножества. Отсюда следует, что процедура  $items(G')$  на рис. 4.34 представляет собой алгоритм построения подмножества, применяемый к НКА  $N$ , построенному на основе  $G'$  описанным способом.

### Допустимые пункты

Мы говорим, что пункт  $A \rightarrow \beta_1 \cdot \beta_2$  допустим (valid) для активного префикса  $\alpha\beta_1$ , если существует порождение  $S' \xrightarrow{*} \alpha A q \xrightarrow{*} \alpha \beta_1 \beta_2 w$ . Вообще говоря, пункт может быть допустимым для многих активных префиксов. Тот факт, что  $A \rightarrow \beta_1 \cdot \beta_2$  допустим для  $\alpha\beta_1$ , говорит нам о том, что именно следует выбрать — перенос или свертку — при обнаружении  $\alpha\beta_1$  в стеке. В частности, если  $\beta_2 \neq \epsilon$ , то это предполагает, что основа еще не полностью перенесена в стек и очередное действие анализатора — перенос. Если  $\beta_2 = \epsilon$ , то  $A \rightarrow \beta_1$  — основа, и мы должны выполнить свертку в соответствии с этой продукцией. Конечно, два допустимых пункта могут указать на разные действия для одного и того же активного префикса. Часть этих конфликтов может быть разре-

шена путем просмотра очередного входного символа, а другие придется разрешать специальными методами, описанными в следующем разделе. Однако не следует считать, что все конфликты действий синтаксического анализатора могут быть разрешены, если LR-метод используется для построения таблицы синтаксического анализа произвольной грамматики.

Вычислить множество допустимых пунктов для каждого активного префикса, который может появиться в стеке LR-анализатора, достаточно просто. На самом деле, основная теорема теории LR-анализа гласит, что множество допустимых пунктов для активного префикса  $\gamma$  в точности равно множеству пунктов, достижимых из начального состояния по пути, помеченному  $\gamma$ , в ДКА, построенном по канонической системе множеств пунктов, с переходами, даваемыми функцией *goto*. По сути, множество допустимых пунктов содержит в себе всю полезную информацию, которая может быть собрана из стека. Поскольку в этой книге мы не доказываем данную теорему, приведем соответствующий пример.

### Пример 4.37

Вновь рассмотрим грамматику (4.19), множество пунктов и функция *goto* которой представлены на рис. 4.35 и 4.36. Ясно, что строка  $E+T^*$  является активным префиксом (4.19). Автомат на рис. 4.36 после прочтения  $E+T^*$  будет находиться в состоянии  $I_7$ , которое содержит пункты

$$\begin{aligned} T &\rightarrow T^* \cdot F \\ F &\rightarrow \cdot(E) \\ F &\rightarrow \cdot\text{id} \end{aligned}$$

представляющие собой пункты, допустимые для  $E+T^*$ . Для того чтобы увидеть это, рассмотрим следующие три правых порождения:

$$\begin{array}{lll} E' \Rightarrow E & E' \Rightarrow E & E' \Rightarrow E+T \\ \Rightarrow E+T & \Rightarrow E+T & \Rightarrow E+T \\ \Rightarrow E+T^*F & \Rightarrow E+T^*F & \Rightarrow E+T^*F \\ & \Rightarrow E+T^*(E) & \Rightarrow E+T^*\text{id} \end{array}$$

Первое порождение показывает допустимость  $T \rightarrow T^* \cdot F$ , второе —  $F \rightarrow \cdot(E)$ , а третье —  $F \rightarrow \cdot\text{id}$  для активного префикса  $E+T^*$ . Можно показать, что для  $E+T^*$  других активных префиксов нет. Это доказательство мы оставляем читателям.  $\square$

### Таблицы SLR-анализа

Теперь покажем, как построить функции *action* и *goto* SLR-анализа по детерминированному конечному автомату, распознающему активные префиксы. Наш алгоритм не дает однозначно определенные таблицы действий синтаксического анализа для всех грамматик, но он успешно работает для многих грамматик языков программирования. Данную нам грамматику  $G$  расширяем до грамматики  $G'$ , и на основе  $G'$  строим  $C$  — каноническую систему множеств пунктов для  $G'$ . По системе  $C$  мы строим *action*, функцию действий синтаксического анализа, и *goto*, функцию переходов, в соответствии с приведенным далее алгоритмом. Он требует знания  $\text{FOLLOW}(A)$  для каждого нетерминала  $A$  грамматики (см. раздел 4.4).

#### Алгоритм 4.8. Построение таблицы SLR-анализа

*Вход.* Расширенная грамматика  $G'$ .

*Выход.* Функции  $action$  и  $goto$  таблицы SLR-анализа для грамматики  $G'$ .

*Метод.*

1. Построим  $C = \{I_0, I_1, \dots, I_n\}$  — систему множеств LR(0)-пунктов для грамматики  $G'$ .
2. Состояние  $i$  строится на основе  $I_i$ . Действия синтаксического анализа для состояния  $i$  определяются следующим образом:
  - a) если  $[A \rightarrow \alpha \cdot a\beta] \in I_i$  и  $goto(I_i, a) = I_j$ , то определить  $action[i, a]$  как “перенос  $j$ ”; здесь  $a$  должно быть терминалом;
  - b) если  $[A \rightarrow \alpha \cdot] \in I_i$ , то определить  $action[i, a]$  как “свертка  $A \rightarrow \alpha'$  для всех  $a$  из  $FOLLOW(A)$ ; здесь  $A$  не должно быть  $S'$ ;
  - c) если  $[S' \rightarrow S \cdot] \in I_i$ , то определить  $action[i, \$]$  как “допуск”.

Если по этим правилам генерируются конфликтующие действия, мы говорим, что грамматика не является SLR(1). Алгоритм не в состоянии построить синтаксический анализатор для нее.

3. Переходы  $goto$  для состояния  $i$  и всех нетерминалов  $A$  строятся по правилу: если  $goto(I_i, A) = I_j$ , то  $goto[i, A] = j$ .
4. Все записи, не определенные по правилам (2) и (3), указываются как “ошибка”.
5. Начальное состояние синтаксического анализатора представляет собой состояние, построенное из множества пунктов, содержащего  $[S' \rightarrow \cdot S]$ .  $\square$

Таблица синтаксического анализа, состоящая из функций  $action$  и  $goto$ , определяемых алгоритмом 4.8, называется *SLR(1)-таблицей грамматики G*. LR-анализатор, использующий SLR(1)-таблицу для грамматики  $G$ , называется SLR(1)-анализатором для  $G$ , а соответствующая грамматика — SLR(1)-грамматикой. Обычно при указании SLR(1) часть “(1)” опускается, поскольку мы не работаем более чем с одним символом предпросмотра.

#### Пример 4.38

Построим SLR-таблицу для грамматики (4.19). Каноническая система множеств LR(0)-пунктов для (4.19) была показана на рис. 4.35. Вначале рассмотрим множество пунктов  $I_0$ :

$$\begin{array}{ll} E' & \rightarrow \cdot E \\ E & \rightarrow \cdot E + T \\ E & \rightarrow \cdot T \\ T & \rightarrow \cdot T^* F \\ T & \rightarrow \cdot F \\ F & \rightarrow \cdot (E) \\ F & \rightarrow \cdot id \end{array}$$

Пункт  $F \rightarrow \cdot (E)$  дает запись  $action[0, ()] =$  “перенос 4”, пункт  $F \rightarrow \cdot id$  — запись  $action[0, id] =$  “перенос 5”. Прочие пункты  $I_0$  к записям  $action$  не приводят. Теперь рассмотрим  $I_1$ :

$$\begin{array}{ll} E' & \rightarrow E \cdot \\ E & \rightarrow E \cdot + T \end{array}$$

Первый пункт дает  $action[1, \$] =$  “допуск”, а второй —  $action[1, +] =$  “перенос 6”. Переайдем к  $I_2$ :

$$\begin{array}{l} E \rightarrow T \\ T \rightarrow T \cdot * F \end{array}$$

Поскольку  $FOLLOW(E) = \{\$, +, )\}$ , из первого пункта следует  $action[2, \$] = action[2, +] = action[2, )] =$  “свертка  $E \rightarrow T$ ”. Второй пункт дает  $action[2, *] =$  “перенос 7”. Продолжая таким образом рассмотрение множеств пунктов, мы получим таблицу, показанную на рис. 4.31. В ней номера продукции в свертках те же, что и номера, под которыми продукции приведены в исходной грамматике (4.18), т.е.  $E \rightarrow E+T$  имеет номер 1,  $E \rightarrow T$  — номер 2 и т.д.  $\square$

### Пример 4.39

Любая SLR(1)-грамматика однозначна, но имеется множество однозначных грамматик, не являющихся SLR(1). Рассмотрим грамматику с продукциями

$$\begin{array}{l} S \rightarrow L = R \\ S \rightarrow R \\ L \rightarrow * R \\ L \rightarrow id \\ R \rightarrow L \end{array} \quad (4.20)$$

Мы можем рассматривать  $L$  и  $R$  как обозначение  $l$ - и  $r$ -значений соответственно, а оператор  $*$  — как оператор “содержимое”<sup>9</sup>. Каноническая система множеств LR(0)-пунктов грамматики (4.20) показана на рис. 4.37.

|        |                             |        |                             |
|--------|-----------------------------|--------|-----------------------------|
| $I_0:$ | $S' \rightarrow \cdot S$    | $I_5:$ | $L \rightarrow id \cdot$    |
|        | $S \rightarrow \cdot L = R$ |        |                             |
|        | $S \rightarrow \cdot R$     | $I_6:$ | $S \rightarrow L = \cdot R$ |
|        | $L \rightarrow \cdot * R$   |        | $R \rightarrow \cdot L$     |
|        | $L \rightarrow \cdot id$    |        | $L \rightarrow \cdot * R$   |
|        | $R \rightarrow \cdot L$     |        | $L \rightarrow \cdot id$    |
| $I_1:$ | $S' \rightarrow S \cdot$    | $I_7:$ | $L \rightarrow * R \cdot$   |
| $I_2:$ | $S \rightarrow L \cdot = R$ | $I_8:$ | $R \rightarrow L \cdot$     |
|        | $R \rightarrow L \cdot$     | $I_9:$ | $S \rightarrow L = R \cdot$ |
| $I_3:$ | $S \rightarrow R \cdot$     |        |                             |
| $I_4:$ | $L \rightarrow * \cdot R$   |        |                             |
|        | $R \rightarrow \cdot L$     |        |                             |
|        | $L \rightarrow \cdot * R$   |        |                             |
|        | $L \rightarrow \cdot id$    |        |                             |

Рис. 4.37. Каноническая LR(0)-система грамматики (4.20)

<sup>9</sup> Как и в разделе 2.8,  $l$ -значение указывает местоположение, а  $r$ -значение — значение, хранящееся в определенном местоположении.

Рассмотрим множество пунктов  $I_2$ . Первый пункт в этом множестве устанавливает  $\text{action}[2, =]$  как “перенос б”. Поскольку  $\text{FOLLOW}(R)$  содержит “=” (чтобы увидеть это, рассмотрите  $S \Rightarrow L = R \Rightarrow *R = R$ ), второй пункт определяет  $\text{action}[2, =]$  как “свертка  $R \rightarrow L$ ”. Следовательно, запись  $\text{action}[2, =]$  определена дважды, а поскольку для нее имеется и запись переноса, и запись свертки, состояние 2 приводит к конфликту перенос/свертка при входном символе =.

Грамматика (4.20) не является однозначной. Обнаруженный конфликт порождается тем, что метод SLR недостаточно мощный, чтобы запомнить левый контекст, необходимый для принятия решения о действиях синтаксического анализатора при входном символе = и просмотренной строке, выводимой из  $L$ . Канонический метод и метод LALR, рассматриваемые ниже, успешно работают с большим множеством грамматик, включая грамматику (4.20). Следует, однако, отметить, что имеются однозначные грамматики, для которых любой метод построения LR-анализатора приведет к таблице с конфликтами действий синтаксического анализа. К счастью, таких грамматик в применении к языкам программирования обычно можно избежать.  $\square$

## Построение канонических таблиц LR-анализа

Сейчас рассмотрим наиболее общую технологию построения таблицы LR-анализа для грамматики. Вспомним, что в SLR-методе состояние  $i$  вызывает свертку по продукции  $A \rightarrow \alpha$ , если множество пунктов  $I_i$  содержит пункт  $[A \rightarrow \alpha]$  и  $a \in \text{FOLLOW}(A)$ . В некоторых ситуациях, однако, когда состояние  $i$  располагается на вершине стека, активный префикс  $\beta\alpha$  в стеке таков, что в правосентенциальной форме за  $\beta A$  не может следовать  $a$ . Следовательно, свертка по продукции  $A \rightarrow \alpha$  будет недопустимой при входном символе  $a$ .

### Пример 4.40

Вновь обратимся к примеру 4.39, где в состоянии 2 был пункт  $R \rightarrow L$ , который мог соответствовать приведенной выше продукции  $A \rightarrow \alpha$ , и  $a$  мог быть знаком =, принадлежащим  $\text{FOLLOW}(R)$ . Таким образом, SLR-анализатор в состоянии 2 с очередным входным символом = вызывает свертку по продукции  $R \rightarrow L$  (вызывается также и перенос — в соответствии с пунктом  $S \rightarrow L = R$  в состоянии 2). Однако в грамматике из примера 4.39 правосентенциальная форма, начинающаяся с  $R = \dots$ , отсутствует. Следовательно, состояние 2, соответствующее только активному префиксу  $L$ , не должно вызывать свертку этого  $L$  в  $R$ .  $\square$

В состоянии можно содержать дополнительную информацию, которая позволит исключить некоторые из таких некорректных сверток по продукции  $A \rightarrow \alpha$ . Разделяя при необходимости состояния, мы можем заставить каждое состояние LR-анализатора точно указывать, какой входной символ может следовать за основой  $\alpha$  (для которой существует возможная свертка в  $A$ ).

Дополнительная информация внедряется в состояние путем переопределения пунктов для включения терминального символа в качестве второго компонента. Общим видом пункта становится  $[A \rightarrow \alpha\beta, a]$ , где  $A \rightarrow \alpha\beta$  представляет собой продукцию,  $a$  — терминал или маркер конца строки \$. Такой объект назовем  $LR(1)$ -пунктом. Здесь 1 означает длину второго компонента, называемого *предпросмотром* (lookahead) пункта<sup>10</sup>. Пред-

<sup>10</sup> Естественно, возможен предпросмотр и большей длины, однако такие предпросмотры в данной книге не обсуждаются.

просмотр не играет роль в пункте вида  $[A \rightarrow \alpha\beta, a]$ , где  $\beta \neq \epsilon$ , однако пункт вида  $[A \rightarrow \alpha, a]$  вызывает свертку по продукции  $A \rightarrow \alpha$  только тогда, когда очередной входной символ —  $a$ . Следовательно, свертка по продукции  $A \rightarrow \alpha$  выполняется только для тех входных символов  $a$ , для которых  $[A \rightarrow \alpha, a]$  является LR(1)-пунктом в состоянии на вершине стека. Множество таких  $a$  всегда будет подмножеством FOLLOW( $A$ ), но может быть и собственным подмножеством, как в примере 4.40.

Формально мы говорим, что LR(1)-пункт  $[A \rightarrow \alpha\beta, a]$  допустим (valid) для активного префикса  $\gamma$ , если существует порождение  $S \xrightarrow{m} \delta A w \xrightarrow{m} \delta\alpha\beta w$ , где

1.  $\gamma = \delta\alpha$ ,
2. либо  $a$  является первым символом  $w$ , либо  $w$  представляет собой  $\epsilon$ , а  $a = \$$ .

#### Пример 4.41

Рассмотрим грамматику

$$\begin{aligned} S &\rightarrow BB \\ B &\rightarrow aB \mid b \end{aligned}$$

Имеется правое порождение  $S \xrightarrow{m} aaBab \xrightarrow{m} aaaBab$ . Мы видим, что пункт  $[B \rightarrow a\cdot B, a]$  допустим для активного префикса  $\gamma = aaa$ , полагая в приведенном выше определении  $\delta = aa$ ,  $A = B$ ,  $w = ab$ ,  $\alpha = a$  и  $\beta = B$ .

Имеется также правое порождение  $S \xrightarrow{m} BaB \xrightarrow{m} BaaB$ , в соответствии с которым пункт  $[B \rightarrow a\cdot B, \$]$  допустим для активного префикса  $Baa$ .  $\square$

Метод построения системы множеств допустимых LR(1)-пунктов, по сути, тот же, что и способ построения канонической системы множеств LR(0)-пунктов. Нам необходимо внести изменения только в две функции — *closure* и *goto*.

Чтобы разобраться в новом определении операции *closure*, рассмотрим пункт вида  $[A \rightarrow \alpha\cdot B\beta, a]$  из множества пунктов, допустимых для некоторого активного префикса  $\gamma$ . Тогда имеется правое порождение  $S \xrightarrow{m} \delta A ax \xrightarrow{m} \delta\alpha B\beta ax$ , где  $\gamma = \delta\alpha$ . Предположим, что  $\beta ax$  порождает терминальную строку  $by$ . Тогда для каждой продукции вида  $B \rightarrow \eta$  с некоторым  $\eta$  мы получаем порождение  $S \xrightarrow{m} \gamma B by \xrightarrow{m} \gamma\eta by$ . Следовательно, пункт  $[B \rightarrow \cdot\eta, b]$  допустим для активного префикса  $\gamma$ . Заметим, что  $b$  может быть первым терминалом, выведенным из  $\beta$ ; возможно также, что из  $\beta$  выводится  $\epsilon$  в порождении  $\beta ax \Rightarrow by$ , и  $b$ , следовательно, может быть  $a$ . Резюмируя обе возможности, мы говорим, что  $b$  может быть любым терминалом из FIRST( $\beta ax$ ), где FIRST — функция, описанная в разделе 4.4. Заметим, что  $x$  не может содержать первый терминал  $by$ , так что FIRST( $\beta ax$ ) = FIRST( $\beta a$ ). Теперь представим способ построения множеств LR(1)-пунктов.

#### Алгоритм 4.9. Построение множеств LR(1)-пунктов

*Вход.* Расширенная грамматика  $G'$ .

*Выход.* Множества LR(1)-пунктов, которые являются множествами пунктов, допустимых для одного или нескольких активных префиксов  $G'$ .

*Метод.* Процедуры *closure* и *goto* и основная подпрограмма *items* для построения множеств пунктов показаны на рис. 4.38.  $\square$

```

function closure(I);
begin
 repeat
 for каждый пункт $[A \rightarrow \alpha \cdot B\beta, a] \in I$,
 каждая продукция $B \rightarrow \gamma$ из G' ,
 и каждый терминал b из $\text{FIRST}(\beta a)$,
 такой, что $[B \rightarrow \cdot \gamma, b] \notin I$ do
 добавить $[B \rightarrow \cdot \gamma, b]$ в I ;
 until пунктов для добавления в I больше нет;
 return I ;
end;
function goto(I, X);
begin
 Пусть J — множество пунктов $[A \rightarrow \alpha X \cdot \beta, a]$,
 таких, что $[A \rightarrow \alpha \cdot X\beta, a] \in I$;
 return closure(J);
end;
procedure items(G');
begin
 $C := \{\text{closure}(\{[S' \rightarrow \cdot S, \$]\})\}$;
 repeat
 for каждое множество пунктов $I \in C$ и каждый
 символ грамматики X , такие, что $\text{goto}(I, X)$
 не пусто и не принадлежит C do
 добавить $\text{goto}(I, X)$ в C
 until множеств пунктов для добавления в C больше нет
end

```

Рис. 4.38. Построение множеств  $LR(1)$ -пунктов для грамматики  $G'$

#### Пример 4.42

Рассмотрим следующую расширенную грамматику.

$$\begin{aligned}
 S' &\rightarrow S \\
 S &\rightarrow CC \\
 C &\rightarrow cC \mid d
 \end{aligned} \tag{4.21}$$

Начнем с вычисления замыкания  $\{[S' \rightarrow \cdot S, \$]\}$ . Для этого сопоставим пункт  $[S' \rightarrow \cdot S, \$]$  с пунктом  $[A \rightarrow \alpha \cdot B\beta, a]$  из процедуры *closure*; таким образом,  $A = S'$ ,  $\alpha = \epsilon$ ,  $B = S$ ,  $\beta = \epsilon$  и  $a = \$$ . Функция *closure* требует добавить  $[B \rightarrow \cdot \gamma, b]$  в  $\text{FIRST}(\beta a)$  для каждой продукции  $B \rightarrow \gamma$  и терминала  $b$ . В нашей грамматике  $B \rightarrow \gamma$  соответствует  $S \rightarrow CC$ , и поскольку  $\beta$  равно  $\epsilon$ , а  $a = \$$ ,  $b$  может быть только  $\$$ . Таким образом, мы добавляем  $[S \rightarrow \cdot CC, \$]$ .

Далее мы продолжаем вычисление замыкания, добавляя все пункты  $[C \rightarrow \cdot \gamma, b]$  для  $b$  из  $\text{FIRST}(C\$)$ . Сопоставляя  $[S \rightarrow \cdot CC, \$]$  и  $[A \rightarrow \alpha \cdot B\beta, a]$ , получаем  $A = S$ ,  $\alpha = \epsilon$ ,  $B = C$ ,  $\beta = C$  и  $a = \$$ . Поскольку  $C$  не порождает пустую строку,  $\text{FIRST}(C\$) = \text{FIRST}(C)$ . Так как  $\text{FIRST}(C)$  содержит терминалы  $c$  и  $d$ , мы добавляем пункты  $[C \rightarrow \cdot cC, c]$ ,  $[C \rightarrow \cdot cC, d]$ ,  $[C \rightarrow \cdot d, c]$  и  $[C \rightarrow \cdot d, d]$ . Никакие из новых пунктов не имеют нетерминалов непосредст-

венно справа от точки, так что мы завершаем на этом построение первого множества LR(1)-пунктов. Начальное множество пунктов следующее:

$$\begin{aligned} I_0: \quad S' &\rightarrow \cdot S, \$ \\ S &\rightarrow \cdot CC, \$ \\ C &\rightarrow \cdot cC, c/d \\ C &\rightarrow \cdot d, c/d \end{aligned}$$

Здесь для удобства записи опущены квадратные скобки, а также для представления двух пунктов —  $[C \rightarrow \cdot cC, c]$  и  $[C \rightarrow \cdot cC, d]$  — используется обозначение  $[C \rightarrow \cdot cC, c/d]$ .

Теперь вычислим  $goto(I_0, X)$  для различных значений  $X$ . Для  $X = S$  мы должны замкнуть пункт  $[S' \rightarrow S \cdot, \$]$ . Никаких других пунктов в замыкании нет, поскольку точка в пункте находится на правом краю. Таким образом, следующее множество пунктов выглядит как

$$I_1: \quad S' \rightarrow S \cdot, \$$$

Для  $X = C$  замыкаем  $[S \rightarrow C \cdot C, \$]$ . Мы добавляем  $C$ -продукции со вторым компонентом  $\$$ , и после этого добавить уже нечего. Тем самым получаем

$$\begin{aligned} I_2: \quad S &\rightarrow C \cdot C, \$ \\ C &\rightarrow \cdot cC, \$ \\ C &\rightarrow \cdot d, \$ \end{aligned}$$

Далее пусть  $X = c$ , и замыкается  $\{[C \rightarrow c \cdot C, c/d]\}$ . Добавляем  $C$ -продукции со вторым компонентом  $c/d$ , получая

$$\begin{aligned} I_3: \quad C &\rightarrow c \cdot C, c/d \\ C &\rightarrow \cdot cC, c/d \\ C &\rightarrow \cdot d, c/d \end{aligned}$$

И наконец, для  $X = d$  находим

$$I_4: \quad C \rightarrow d \cdot, c/d$$

На этом мы завершаем рассмотрение  $goto$  на множестве  $I_0$ . Для  $I_1$  новые множества не образуются, но  $I_2$  имеет переходы по  $C$ ,  $c$  и  $d$ . Для  $C$  находим

$$I_5: \quad S \rightarrow CC \cdot, \$$$

Для  $c$  рассматриваем замыкание  $\{[C \rightarrow c \cdot C, \$]\}$ , получая

$$\begin{aligned} I_6: \quad C &\rightarrow c \cdot C, \$ \\ C &\rightarrow \cdot cC, \$ \\ C &\rightarrow \cdot d, \$ \end{aligned}$$

Заметим, что  $I_6$  отличается от  $I_3$  только вторыми компонентами. Мы увидим, что некоторые множества LR(1)-пунктов для грамматики могут иметь одинаковые первые и разные вторые компоненты. При построении коллекции множеств LR(0)-пунктов для той же грамматики каждое множество LR(0)-пунктов будет совпадать с множеством первых компонентов одного или нескольких множеств LR(1)-пунктов. Подробнее об этом мы поговорим при обсуждении LALR-анализа.

Продолжая рассмотрение функции  $goto$  для  $I_2$ , находим  $goto(I_2, d)$ :

$$I_7: \quad C \rightarrow d \cdot, \$$$

Переходя к  $I_3$ , обнаружим, что переходы из  $I_3$  по символам  $c$  и  $d$  представляют собой соответственно  $I_3$  и  $I_4$ , а  $goto(I_3, C)$  —

$$I_8: \quad C \rightarrow cC, c/d$$

$I_4$  и  $I_5$  переходов не имеют. Переходы из  $I_6$  по символам  $c$  и  $d$  представляют собой соответственно  $I_6$  и  $I_7$ , а  $goto(I_6, C)$  —

$$I_9: \quad C \rightarrow cC, \$$$

Остальные множества пунктов не имеют переходов, и на этом наша работа завершена. На рис. 4.39 показаны найденные десять множеств пунктов с их переходами.  $\square$

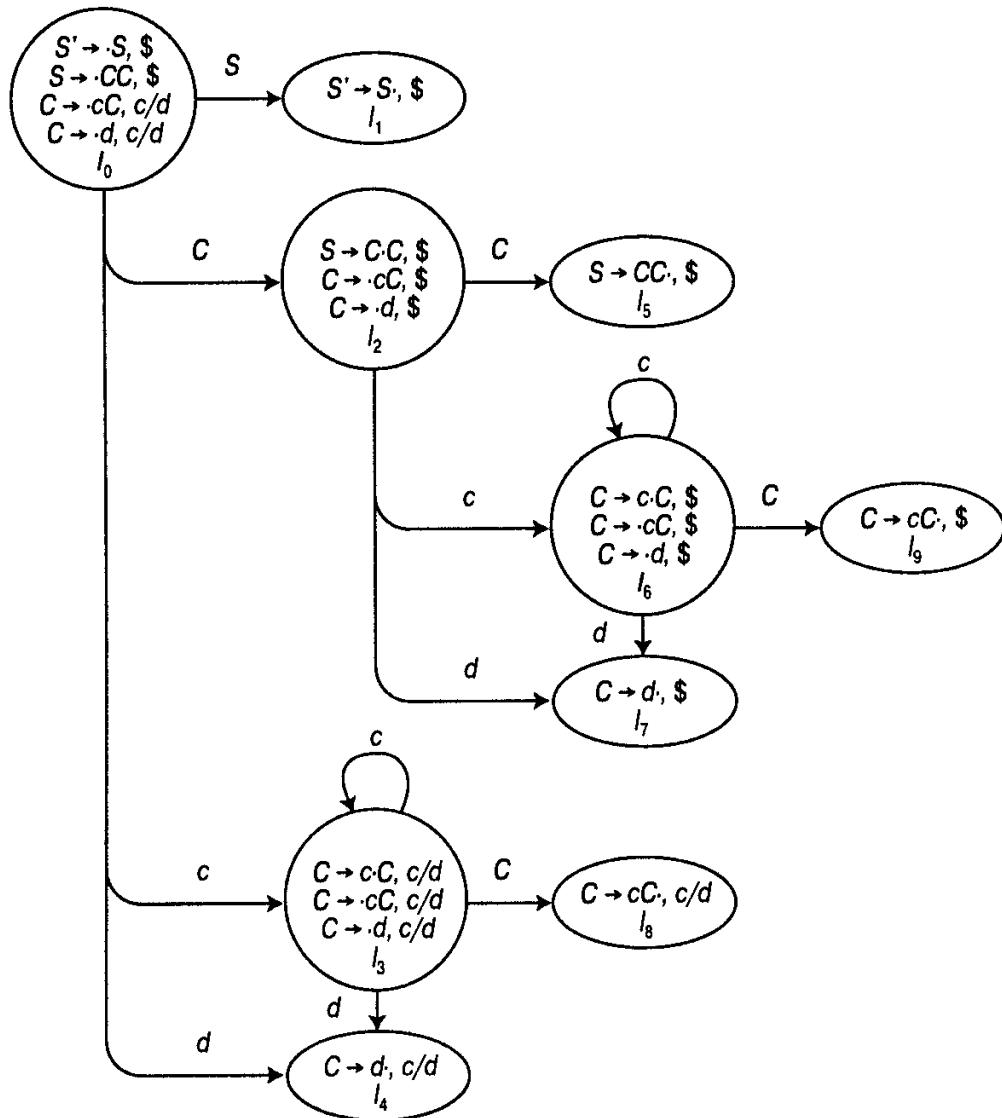


Рис. 4.39. Граф переходов для грамматики (4.21)

Теперь приведем правила, посредством которых на основе множеств LR(1)-пунктов строятся функции *action* и *goto* LR(1)-анализа. Функции представлены, как и ранее, в виде таблицы; отличие заключается в значениях записей.

#### Алгоритм 4.10. Построение канонической таблицы LR-анализа

*Вход.* Расширенная грамматика  $G'$ .

*Выход.* Каноническая таблица функций *action* и *goto* LR-анализа для грамматики  $G'$ .

*Метод.*

- Построим  $C = \{I_0, I_1, \dots, I_n\}$ , систему множеств LR(1)-пунктов для  $G'$ .
- Состояние  $i$  синтаксического анализатора строится на основе множества  $I_i$ . Действия синтаксического анализа для состояния  $i$  определяются следующим образом:
  - если  $[A \rightarrow \alpha \cdot a\beta, b] \in I_i$  и  $goto(I_i, a) = I_j$ , то установим  $action[i, a]$  равным “перенос  $j$ ” (здесь  $a$  должно быть терминалом);
  - если  $[A \rightarrow \alpha \cdot, a] \in I_i, A \neq S'$ , то определим  $action[i, a]$  как “свертка  $A \rightarrow \alpha'$ ;
  - если  $[S' \rightarrow S \cdot, \$] \in I_i$ , то установим  $action[i, \$]$  равным “допуск”.

Если применение этих правил приводит к конфликту, грамматика не является LR(1)-грамматикой и данный алгоритм к ней неприменим.

- Переходы  $goto$  для состояния  $i$  определяются следующим образом: если  $goto(I_i, A) = I_j$ , то  $goto[i, A] = j$ .
- Все записи, не определенные правилами (2) и (3), задаются как “ошибка”.
- Начальное состояние синтаксического анализатора представляет собой состояние, построенное из множества, содержащего пункт  $[S' \rightarrow \cdot S, \$]$ .  $\square$

Таблица, построенная на основе функций  $action$  и  $goto$  синтаксического анализа, полученных с использованием алгоритма 4.10, называется *канонической таблицей LR(1)-анализа*. LR-анализатор, использующий эту таблицу, называется *каноническим LR(1)-анализатором*. Если функция  $action$  синтаксического анализа не имеет множественных записей, то данная грамматика называется *LR(1)-грамматикой*. Как и ранее, мы будем опускать “(1)”, если это понятно из контекста.

#### Пример 4.43

Каноническая таблица синтаксического анализа для грамматики (4.21) показана на рис. 4.40. Продукциями 1, 2 и 3 являются соответственно  $S \rightarrow CC$ ,  $C \rightarrow cC$  и  $C \rightarrow d$ .  $\square$

| СОСТОЯНИЕ | action |    |     | goto |   |
|-----------|--------|----|-----|------|---|
|           | c      | d  | \$  | S    | C |
| 0         | s3     | s4 |     | 1    | 2 |
| 1         |        |    | acc |      |   |
| 2         | s6     | s7 |     |      | 5 |
| 3         | s3     | s4 |     |      | 8 |
| 4         | r3     | r3 |     |      |   |
| 5         |        |    | r1  |      |   |
| 6         | s6     | s7 |     |      | 9 |
| 7         |        |    | r3  |      |   |
| 8         | r2     | r2 |     |      |   |
| 9         |        |    | r2  |      |   |

Рис. 4.40. Каноническая таблица синтаксического анализа для грамматики (4.21)

Любая SLR(1)-грамматика является LR(1)-грамматикой, но для SLR(1)-грамматики канонический LR-анализатор может иметь больше состояний, чем SLR-анализатор для той же грамматики. Грамматика из предыдущего примера является SLR-грамматикой; SLR-анализатор для нее имеет семь состояний, а на рис. 4.40 их десять.

## Построение таблиц LARL-анализа

Сейчас мы переходим к последнему методу построения синтаксического анализатора, методу LALR (*lookahead LR*). Этот метод часто используется на практике, поскольку получаемые с его помощью таблицы значительно меньше канонических LR-таблиц; кроме того, с помощью LALR-грамматики легко выражается большинство стандартных синтаксических конструкций языков программирования. Почти то же самое можно сказать и о SLR-грамматиках, однако имеется ряд конструкций, с которыми SLR-технология справиться не в состоянии (см., например, упр. 4.39).

Сравнивая размеры таблиц синтаксического анализа, отметим, что SLR- и LALR-таблицы грамматики всегда имеют одинаковое число состояний (порядка нескольких сотен для языков типа Pascal<sup>11</sup>). Каноническая LR-таблица для языка такого типа обычно содержит несколько тысяч состояний. Таким образом, построение SLR- или LARL-таблиц существенно проще и экономичнее канонических LR-таблиц.

В качестве введения обратимся вновь к грамматике (4.21), множества LR(1)-пунктов которой были приведены на рис. 4.39. Возьмем пару похожих состояний, например  $I_4$  и  $I_7$ . Эти состояния имеют только пункты с первым компонентом  $C \rightarrow d$ . В  $I_4$  предпротивляемыми символами могут быть  $c$  и  $d$ , а в  $I_7$  — только  $\$$ .

Чтобы увидеть разницу между  $I_4$  и  $I_7$  в синтаксическом анализаторе, обратите внимание, что грамматика (4.21) порождает регулярное множество  $c^*dc^*d$ . При считывании входного потока  $cc\dots cdcc\dots cd$  синтаксический анализатор переносит первую группу символов  $c$  и следующий за ними  $d$  в стек, попадая после считывания символа  $d$  в состояние 4. Затем синтаксический анализатор вызывает свертку по продукции  $C \rightarrow d$ , обусловленную следующим входным символом  $c$  или  $d$ . Требование, чтобы следующим входным символом был  $c$  или  $d$ , имеет смысл, поскольку это — символы, с которых может начинаться строка  $c^*d$ . Если после первого  $d$  следует  $\$$ , то получается входной поток типа  $ccd$ , который не содержится в рассматриваемом языке, и состояние 4 совершенно справедливо обнаруживает ошибку при очередном входном символе  $\$$ .

В состояние 7 синтаксический анализатор попадает после чтения второго  $d$ . Соответственно, синтаксический анализатор должен обнаружить во входном потоке символ  $\$$ , иначе входная строка не соответствует регулярному выражению  $c^*dc^*d$ . Таким образом, состояние 7 должно приводить к свертке  $C \rightarrow d$  при входном символе  $\$$  и к ошибке при входном символе  $c$  или  $d$ .

<sup>11</sup> Вот некоторые характеристики реальных языков программирования.

| Язык           | Количество |             |           |                |
|----------------|------------|-------------|-----------|----------------|
|                | Токены     | Нетерминалы | Терминалы | Ключевые слова |
| Algol 60       | 1085       | 119         | 92        | 25             |
| Pascal         | 1012       | 110         | 84        | 35             |
| Turbo Pascal 6 | 1488       | 143         | 89        | 55             |
| Delphi         | 1825       | 180         | 90        | 83             |
| Modula-2       | 887        | 70          | 88        | 39             |
| Oberon-2       | 726        | 43          | 91        | 34             |
| C              | 917        | 917         | 123       | 27             |
| C++            | 1662       | 126         | 131       | 47             |
| Java           | 1771       | 174         | 121       | 48             |
| Ada            | 2206       | 226         | 102       | 63             |

Для полного ознакомления со сравнительным анализом различных языков программирования можно обратиться к статье Свердлов С. Арифметика синтаксиса. — РС Week/RE №42-43, 1998. — Прим. ред.

Заменим теперь состояния  $I_4$  и  $I_7$  на  $I_{47}$ , которое представляет собой объединение  $I_4$  и  $I_7$ , состоящее из трех пунктов —  $[C \rightarrow d\cdot, c/d/\$]$ . Все переходы по  $d$  в  $I_4$  или  $I_7$  из  $I_0, I_2, I_3$  и  $I_6$  ведут теперь в  $I_{47}$ . Действие в состоянии 47 — свертка при любом входном символе. Такой синтаксический анализатор в целом ведет себя так же, как и исходный, хотя и может свернуть  $d$  в  $C$  в условиях, когда исходный синтаксический анализатор объявил бы об ошибке, например при входной строке  $ccd$  или  $c dc dc$ . В конце концов, ошибка будет обнаружена — перед тем как мы получим любой символ, вызывающий перенос.

Обобщая, мы можем рассмотреть множества LR(1)-пунктов, имеющих одно и то же ядро (core), т.е. множество первых компонентов, и объединить эти множества с общими ядрами в одно множество пунктов. Например, на рис. 4.39 такую пару с ядром  $\{C \rightarrow d\cdot\}$  образуют состояния  $I_4$  и  $I_7$ . Аналогично множества  $I_3$  и  $I_6$  образуют другую пару — с ядром  $\{C \rightarrow c\cdot C, C \rightarrow \cdot cC, C \rightarrow \cdot d\}$ . Имеется и еще одна пара —  $I_8$  и  $I_9$  — с ядром  $\{C \rightarrow cC\cdot\}$ . Заметим, что, вообще говоря, ядро является множеством LR(0)-пунктов рассматриваемой грамматики и LR(1)-грамматика может давать более двух множеств пунктов с одним и тем же ядром.

Поскольку ядро множества  $goto(I, X)$  зависит только от ядра множества  $I$ , значения функции  $goto$  объединяемых множеств также могут быть объединены. Таким образом, проблем вычисления функции  $goto$  при слиянии множеств не возникает. Функция же  $action$  должна быть изменена, чтобы отражать (не ошибочные) действия всех объединяемых множеств пунктов.

Предположим, у нас есть LR(1)-грамматика, т.е. грамматика, множества LR(1)-пунктов которой не вызывают конфликтов действий синтаксического анализа. Если мы заменим все состояния, имеющие одно и то же ядро, их объединениями, возможно, что полученное в результате состояние будет иметь конфликт, хотя это маловероятно по следующей причине. Предположим, что в объединении возникает конфликт при просмотре входного символа  $a$ , поскольку существует пункт  $[A \rightarrow \alpha\cdot, a]$ , вызывающий свертку по продукции  $A \rightarrow \alpha$ , и пункт  $[B \rightarrow \beta\cdot a\gamma, b]$ , приводящий к переносу. Тогда некоторое множество пунктов, из которого было сформировано объединение, имело пункт  $[A \rightarrow \alpha\cdot, a]$ . Поскольку ядра объединяемых множеств совпадают, это множество должно также иметь пункт  $[B \rightarrow \beta\cdot a\gamma, c]$  для некоторого  $c$ . Но в таком случае это состояние имеет конфликт переноса/свертки, и, вопреки нашему предположению, грамматика не была LR(1)-грамматикой. Следовательно, объединение состояний с одинаковыми ядрами не может привести к конфликту переноса/свертки, если такой конфликт не присутствовал ни в одном из исходных состояний (поскольку переносы зависят только от ядра, но не от предпросмотра).

Тем не менее при объединении возможно появление конфликта свертка/свертки, как показано в следующем примере.

#### Пример 4.44

Рассмотрим грамматику

$$\begin{array}{lcl} S & \rightarrow & S \\ S & \rightarrow & aAd \mid bBd \mid aBe \mid bAe \\ A & \rightarrow & c \\ B & \rightarrow & c \end{array}$$

которая генерирует четыре строки —  $acd$ ,  $ace$ ,  $bcd$  и  $bce$ . Читатель может убедиться, что грамматика является LR(1), построив множества пунктов. Сделав это, мы обнаружим

множество пунктов  $\{[A \rightarrow c, d], [B \rightarrow c, e]\}$ , допустимых для активного префикса  $ac$ , и  $\{[A \rightarrow c, e], [B \rightarrow c, d]\}$  — для префикса  $bc$ . Ни одно из этих множеств не вызывает конфликта; ядра их одинаковы. Однако их объединение

$$\begin{array}{l} A \rightarrow c, d/e \\ B \rightarrow c, d/e \end{array}$$

вызывает конфликт свертка/свертка, поскольку при входных символах  $d$  и  $e$  вызываются две свертки:  $A \rightarrow c$  и  $B \rightarrow c$ .  $\square$

Теперь мы готовы рассмотреть первый из двух алгоритмов построения LALR-таблиц. Основная идея состоит в создании множеств LR(1)-пунктов и, если это не вызывает конфликтов, объединении множеств с одинаковыми ядрами. Затем на базе системы множеств пунктов строим таблицу синтаксического анализа. Описываемый метод служит, в первую очередь, определением LALR(1)-грамматик. Построение полной системы LR(1)-множеств пунктов требует слишком много памяти и времени, чтобы использоваться на практике.

#### Алгоритм 4.11. Прямое построение LALR-таблицы

*Вход.* Расширенная грамматика  $G'$ .

*Выход.* Таблица функций LALR-анализа *action* и *goto* для грамматики  $G'$ .

*Метод.*

- Построим  $C = \{I_0, I_1, \dots, I_n\}$ , систему множеств LR(1)-пунктов для  $G'$ .
- Для каждого ядра, имеющегося среди множества LR(1)-пунктов, находим все множества, имеющие это ядро, и заменяем эти множества их объединением.
- Пусть  $C' = \{J_0, J_1, \dots, J_m\}$  — полученные в результате множества LR(1)-пунктов. Функция *action* для состояния  $i$  строится из  $J_i$  так же, как и в алгоритме 4.10. Если при этом обнаруживается конфликт, алгоритм не в состоянии построить синтаксический анализатор, а грамматика не является LALR(1)-грамматикой.
- Таблица *goto* строится следующим образом. Если  $J$  — объединение одного или нескольких множеств LR(1)-пунктов, т.е.  $J = I_1 \cup I_2 \cup \dots \cup I_k$ , то ядра множеств  $goto(I_1, X), goto(I_2, X), \dots, goto(I_k, X)$  одни и те же, поскольку  $I_1, I_2, \dots, I_k$  имеют одно и то же ядро. Обозначим через  $K$  объединение всех множеств пунктов, имеющих то же ядро, что и  $goto(I_1, X)$ . Тогда  $goto(J, X) = K$ .  $\square$

Таблица, построенная по алгоритму 4.11, называется *таблицей LALR-анализа* для грамматики  $G$ . Если конфликты действий синтаксического анализа отсутствуют, такая грамматика называется *LALR(1)-грамматикой*; система множеств пунктов  $C' = \{J_0, J_1, \dots, J_m\}$ , построенная на шаге (3) алгоритма, называется *LALR(1)-системой*.

#### Пример 4.45

Вновь рассмотрим грамматику (4.21), граф *goto* которой показан на рис. 4.39. Как упоминалось ранее, в ней имеется три пары множеств пунктов, которые могут быть объединены.  $I_3$  и  $I_6$  заменяются их объединением

$$\begin{array}{ll} I_{36}: & C \rightarrow c \cdot C, c/d/\$ \\ & C \rightarrow \cdot cC, c/d/\$ \\ & C \rightarrow \cdot d, c/d/\$ \end{array}$$

$I_4$  и  $I_7$  — объединением

$$I_{47}: C \rightarrow d, c/d/\$$$

а  $I_8$  и  $I_9$  —

$$I_{89}: C \rightarrow cC, c/d/\$$$

Функции *action* и *goto* для объединенных множеств пунктов показаны на рис. 4.41.

| Состояние | <i>action</i> |          |      | <i>goto</i> |          |
|-----------|---------------|----------|------|-------------|----------|
|           | <i>c</i>      | <i>d</i> | $\$$ | <i>S</i>    | <i>C</i> |
| 0         | s36           | s47      |      | 1           | 2        |
| 1         |               |          | acc  |             |          |
| 2         | s36           | s47      |      |             | 5        |
| 36        | s36           | s47      |      |             | 89       |
| 47        | r3            | r3       | r3   |             |          |
| 5         |               |          | r1   |             |          |
| 89        | r2            | r2       | r2   |             |          |

Рис. 4.41. Таблица LALR-анализа для грамматики (4.21)

Чтобы увидеть, каким образом вычисляется функция *goto*, рассмотрим  $goto(I_{36}, C)$ . В исходном множестве LR(1)-пунктов  $goto(I_3, C) = I_8$  теперь является частью  $I_{89}$ , а потому мы определяем, что  $goto(I_{36}, C) = I_{89}$ . Тот же вывод мы можем сделать, рассматривая  $I_6$  — вторую часть  $I_{36}$ . В качестве второго примера рассмотрим  $goto(I_2, c)$ , переход, выполняемый после переноса в состоянии  $I_2$  при входном символе *c*. В исходных множествах LR(1)-пунктов  $goto(I_2, c) = I_6$ . Поскольку теперь  $I_6$  является частью  $I_{36}$ ,  $goto(I_2, c)$  становится равным  $I_{36}$ ; таким образом, запись в таблице на рис. 4.41 для состояния 2 и входного символа *c* — s36, что означает перенос и помещение состояния 36 в стек.

При входной строке из языка *c\*dc\*d* как LR-анализатор на рис. 4.40, так и LALR-анализатор на рис. 4.41 выполняют одну и ту же последовательность переносов и сверток, хотя имена состояний в стеке могут отличаться. Так, если LR-анализатор помещает в стек  $I_3$  или  $I_6$ , LALR-анализатор помещает в стек  $I_{36}$ . Это верно и в общем случае LALR-грамматики. LR- и LALR-анализаторы функционируют одинаково при корректной входной строке.

Однако при строке с ошибками LALR-анализатор может выполнить несколько сверток после того, как LR-анализатор объявил об ошибке. Вместе с тем, LALR-анализатор никогда не перенесет символ после того, как ошибка распознана LR-анализатором. Например, для входной строки *ccd\$* LR-анализатор на рис. 4.34 поместит в стек

0 c 3 c 3 d 4

и в состоянии 4 обнаружит ошибку, поскольку следующий входной символ —  $\$$ , а для состояния 4 соответствующая запись таблицы *action* — ошибка. LALR-анализатор, показанный на рис. 4.41, выполняет те же действия, помещая в стек

0 c 36 c 36 d 47

Однако состояние 47 при входном символе  $\$$  приводит к свертке  $C \rightarrow d$ . Таким образом, LALR-анализатор изменит содержимое стека на

0 c 36 c 36 C 89

Действие в состоянии 89 для входного символа  $\$$  — свертка  $C \rightarrow cC$ , после чего содержимое стека приобретает вид

После этого та же свертка выполняется повторно, что приводит к стеку

0 С 2

И наконец, мы обнаруживаем ошибку в соответствии с записью *action* для состояния 2 и входного символа \$.  $\square$

## Эффективное построение таблиц LALR-анализа

Имеется ряд модификаций, которые можно внести в алгоритм 4.11, чтобы избежать построения полной системы множеств LR(1)-пунктов в процессе создания таблицы LALR(1)-анализа. Во-первых, множество пунктов  $I$  можно представить его ядром, а значит, теми пунктами, которые либо являются начальным пунктом  $[S' \rightarrow \cdot S, \$]$ , либо имеют точку где-то не в начале правой части.

Во-вторых, действия синтаксического анализа для  $I$  можно вычислить на основе только его ядра. Любой пункт, вызывающий свертку по продукции  $A \rightarrow \alpha$ , будет находиться в ядре (за исключением  $\alpha = \epsilon$ ). Свертка по продукции  $A \rightarrow \epsilon$  вызывается при входном символе  $a$  тогда и только тогда, когда существует пункт ядра  $[B \rightarrow \gamma C \delta, b]$ , такой, что  $C \xrightarrow{r_m} A\eta$  для некоторого  $\eta$ , а  $a \in \text{FIRST}(\eta\delta b)$ . Множество нетерминалов  $A$ , таких, что  $C \xrightarrow{r_m} A\eta$ , может быть вычислено заранее для каждого нетерминала  $C$ .

Переносы, генерируемые множеством  $I$ , можно найти по ядру  $I$  следующим образом. Перенос  $a$  вычисляется, если существует базисный пункт  $[B \rightarrow \gamma C \delta, b]$ , где  $C \xrightarrow{r_m} ax$  содержится в порождении, в котором последний шаг не использует  $\epsilon$ -продукцию. Множество таких  $a$  также может быть вычислено заранее для каждого  $C$ .

На основе ядра может быть найдена и функция переходов *goto*. Если в ядре множества  $I$  имеется пункт  $[B \rightarrow \gamma X \delta, b]$ , то он есть и в ядре множества *goto*( $I, X$ ). Кроме того, в ядре *goto*( $I, X$ ) будет находиться пункт  $[A \rightarrow X \cdot \beta, a]$ , если в ядре  $I$  имеется пункт  $[B \rightarrow \gamma C \delta, b]$  и  $C \xrightarrow{r_m} A\eta$  для некоторого  $\eta$ . Если для всех пар нетерминалов  $A$  и  $C$  вычислить заранее, выполняется ли  $C \xrightarrow{r_m} A\eta$  для некоторого  $\eta$ , то вычисление множеств пунктов на основе только ядер будет лишь немногим менее эффективно, чем вычисление на базе замкнутых множеств пунктов.

При вычислении LALR(1)-множеств пунктов для расширенной грамматики  $G'$  мы начинаем с ядра  $S' \rightarrow \cdot S$  начального множества пунктов  $I_0$ . Затем вычисляем ядра переходов *goto* из  $I_0$ , как было описано выше, и продолжаем вычисления переходов *goto* для каждого вновь сгенерированного ядра до тех пор, пока не получим полную систему множеств LR(0)-пунктов.

### Пример 4.46

Обратимся вновь к расширенной грамматике

$$\begin{array}{lcl} S' & \rightarrow & S \\ S & \rightarrow & L = R \mid R \\ L & \rightarrow & * R \mid \text{id} \\ R & \rightarrow & L \end{array}$$

Ядра множеств LR(0)-пунктов для этой грамматики показаны на рис. 4.42.  $\square$

|        |                             |        |                                 |
|--------|-----------------------------|--------|---------------------------------|
| $I_0:$ | $S' \rightarrow \cdot S$    | $I_5:$ | $L \rightarrow \text{id} \cdot$ |
| $I_1:$ | $S' \rightarrow S \cdot$    | $I_6:$ | $S \rightarrow L \cdot R$       |
| $I_2:$ | $S \rightarrow L \cdot = R$ | $I_7:$ | $L \rightarrow *R \cdot$        |
|        | $R \rightarrow L \cdot$     | $I_8:$ | $R \rightarrow L \cdot$         |
| $I_3:$ | $S \rightarrow R \cdot$     | $I_9:$ | $S \rightarrow L = R \cdot$     |
| $I_4:$ | $L \rightarrow * \cdot R$   |        |                                 |

Рис. 4.42. Ядра множеств LR(0)-пунктов грамматики (4.20)

Теперь расширим ядра, добавляя к каждому LR(0)-пункту надлежащие символы предпросмотра (вторые компоненты). Для того чтобы понять, как от множества пунктов  $I$  символы распространяются на множество  $goto(I, X)$ , рассмотрим LR(0)-пункт  $B \rightarrow \gamma \cdot C\delta$  из ядра  $I$ . Предположим, что  $C \xrightarrow{m} A\eta$  для некоторого  $\eta$  (возможно,  $C = A$  и  $\eta = \epsilon$ ), а  $A \rightarrow X\beta$  — продукция. Тогда LR(0)-пункт  $A \rightarrow X \cdot \beta$  принадлежит  $goto(I, X)$ .

Предположим теперь, что мы рассматриваем не LR(0)-, а LR(1)-пункты, и пункт  $[B \rightarrow \gamma \cdot C\delta, b]$  принадлежит  $I$ . Тогда для каких значений  $a$  пункт  $[A \rightarrow X \cdot \beta, a]$  содержится в  $goto(I, X)$ ? Безусловно, если некоторое  $a \in FIRST(\eta\delta)$ , то порождение  $C \xrightarrow{m} A\eta$  свидетельствует, что  $[A \rightarrow X \cdot \beta, a]$  должен содержаться в  $goto(I, X)$ . В этом случае значение  $b$  не играет роли и мы говорим, что  $a$  в качестве предпросмотра для  $A \rightarrow X \cdot \beta$  генерируется произвольно. По определению,  $\$$  генерируется произвольно для пункта  $S' \rightarrow \cdot S$  из начального множества пунктов.

Однако существует еще один источник предпросмотров для пункта  $A \rightarrow X \cdot \beta$ . Если  $\eta\delta \xrightarrow{m} \epsilon$ , то в  $goto(I, X)$  будет находиться и  $[A \rightarrow X \cdot \beta, b]$ . В этом случае мы говорим о том, что предпросмотры *распространяются* от  $B \rightarrow \gamma \cdot C\delta$  к  $A \rightarrow X \cdot \beta$ . Приведенный ниже алгоритм предоставляет простой метод определения того, когда LR(1)-пункт в  $I$  генерирует предпросмотр в  $goto(I, X)$  произвольно и когда происходит распространение предпросмотра.

#### Алгоритм 4.12. Определение предпросмотров

*Вход.* Ядро  $K$  множества LR(0)-пунктов  $I$  и символ грамматики  $X$ .

*Выход.* Произвольно генерируемые пунктами из  $I$  предпросмотры для базисных пунктов в  $goto(I, X)$  и пункты из  $I$ , от которых предпросмотры распространяются к пунктам ядра в  $goto(I, X)$ .

*Метод.* Алгоритм приведен на рис. 4.43. Здесь используется искусственный символ предпросмотра  $\#$  для обнаружения ситуаций, когда происходит распространение предпросмотров.  $\square$

```

for каждый пункт $B \rightarrow \gamma \cdot \delta$ из K do begin
 $J' := \text{closure}(\{[B \rightarrow \gamma \cdot \delta, \#]\})$;
 if $[A \rightarrow \alpha \cdot X\beta, a]$ принадлежит J' , где a — не $\#$ then
 Предпросмотр a генерируется для пункта $A \rightarrow \alpha X \cdot \beta$
 в $goto(I, X)$ произвольно;

```

```

if [A→α·Xβ, #] принадлежит J' then
 Предпросмотры распространяются из B→γ·δ от I к
 A→α·X·β из goto(I, X)
end

```

Рис. 4.43. Поиск распространяемых и спонтанных предпросмотров

Рассмотрим теперь поиск предпросмотров, связанных с пунктами в ядрах множеств LR(0)-пунктов. Мы знаем, что \$ является предпросмотром для  $S' \rightarrow \cdot S$  в начальном множестве LR(0)-пунктов. Алгоритм 4.12 дает нам все предпросмотры, генерируемые произвольно. После перечисления всех этих предпросмотров мы должны позволить им распространяться до тех пор, пока это возможно. Имеется множество различных подходов, причем все они в некотором смысле отслеживают “новые” предпросмотры, которые “пришли” в пункт, но еще не “ушли” от него. Следующий алгоритм описывает одну из технологий распространения предпросмотров во все пункты.

#### Алгоритм 4.13. Эффективное вычисление ядер LALR(1)-систем множеств пунктов

*Вход.* Расширенная грамматика  $G'$ .

*Выход.* Ядра LALR(1)-системы множеств пунктов  $G'$ .

*Метод.*

1. Используя описанный ранее метод, построим ядра множеств LR(0)-пунктов для  $G$ .
2. Применим алгоритм 4.12 к ядру каждого множества LR(0)-пунктов и грамматическому символу  $X$ , чтобы определить, какие предпросмотры генерируются произвольно для базисных пунктов в  $goto(I, X)$  и от каких пунктов в  $I$  предпросмотры распространяются к базисным пунктам в  $goto(I, X)$ .
3. Инициализируем таблицу, которая для каждого базисного пункта в каждом множестве пунктов дает связанные предпросмотры. Изначально с каждым пунктом связаны только те предпросмотры, которые в п. 2 определены как произвольно сгенерированные.
4. Повторим проходы по базисным пунктам во всех множествах. При посещении пункта  $i$  мы с помощью таблицы, построенной в п. 2, ищем базисные пункты, к которым  $i$  распространяет свои предпросмотры. Текущее множество предпросмотров для  $i$  добавляется к множествам, связанным с каждым из пунктов, в которые  $i$  распространяет свои предпросмотры. Мы продолжаем выполнять такие проходы по базисным пунктам до тех пор, пока не останется новых предпросмотров для распространения.  $\square$

#### Пример 4.47

Построим ядра LALR(1)-пунктов для грамматики из предыдущего примера. Ядра LR(0)-пунктов были показаны на рис. 4.42. Применяя алгоритм 4.12 к ядру множества пунктов  $I_0$ , мы вычисляем  $closure(\{[S' \rightarrow \cdot S, \#]\})$ , которое представляет собой

$$\begin{aligned}
S' &\rightarrow \cdot S, \# \\
S &\rightarrow \cdot L = R, \# \\
S &\rightarrow \cdot R, \# \\
L &\rightarrow \cdot *R, \#= \\
L &\rightarrow \cdot id, \#= \\
R &\rightarrow \cdot L, \#
\end{aligned}$$

Два пункта в этом замыкании приводят к произвольной генерации предпросмотров. Пункт  $[L \rightarrow \cdot *R, =]$  приводит к тому, что предпросмотр  $=$  произвольно генерируется для базисного пункта  $L \rightarrow \cdot R$  в  $I_4$ , а пункт  $[L \rightarrow \cdot id, =]$  — что предпросмотр  $=$  произвольно генерируется для пункта  $L \rightarrow id \cdot$  в  $I_5$ .

Пример распространения предпросмотров среди базисных пунктов, определенный в шаге (2) алгоритма 4.13, приведен на рис. 4.44. Например, переходы из множества  $I_0$  по символам  $S, L, R, *$  и  $id$  представляют собой соответственно  $I_1, I_2, I_3, I_4$  и  $I_5$ . Для  $I_0$  мы вычисляем замыкание только одного базисного пункта —  $[S' \rightarrow \cdot S, \#]$ . Таким образом,  $S' \rightarrow \cdot S$  распространяет свой предпросмотр к каждому базисному пункту в множествах от  $I_1$  до  $I_5$ .

| Из     |                             | В      |                             |
|--------|-----------------------------|--------|-----------------------------|
| $I_0:$ | $S' \rightarrow \cdot S$    | $I_1:$ | $S' \rightarrow S \cdot$    |
|        |                             | $I_2:$ | $S \rightarrow L \cdot = R$ |
|        |                             | $I_3:$ | $R \rightarrow L \cdot$     |
|        |                             | $I_4:$ | $L \rightarrow * \cdot R$   |
|        |                             | $I_5:$ | $L \rightarrow id \cdot$    |
| $I_2:$ | $S \rightarrow L \cdot = R$ | $I_6:$ | $S \rightarrow L = R$       |
| $I_4:$ | $L \rightarrow * \cdot R$   | $I_4:$ | $L \rightarrow * \cdot R$   |
|        |                             | $I_5:$ | $L \rightarrow id \cdot$    |
|        |                             | $I_7:$ | $L \rightarrow * R \cdot$   |
|        |                             | $I_8:$ | $R \rightarrow L \cdot$     |
| $I_6:$ | $S \rightarrow L = R$       | $I_4:$ | $L \rightarrow * \cdot R$   |
|        |                             | $I_5:$ | $L \rightarrow * \cdot R$   |
|        |                             | $I_8:$ | $R \rightarrow L \cdot$     |
|        |                             | $I_9:$ | $S \rightarrow L = R \cdot$ |

Рис. 4.44. Распространение предпросмотров

| Множество | Пункт                       | Предпросмотры |          |          |          |
|-----------|-----------------------------|---------------|----------|----------|----------|
|           |                             | Начальный     | Проход 1 | Проход 2 | Проход 3 |
| $I_0:$    | $S' \rightarrow \cdot S$    | \$            | \$       | \$       | \$       |
| $I_1:$    | $S' \rightarrow S \cdot$    |               | \$       | \$       | \$       |
| $I_2:$    | $S \rightarrow L \cdot = R$ |               | \$       | \$       | \$       |
| $I_3:$    | $R \rightarrow L \cdot$     |               | \$       | \$       | \$       |
| $I_4:$    | $L \rightarrow * \cdot R$   | =             | =\$      | =\$      | =\$      |
| $I_5:$    | $L \rightarrow id \cdot$    | =             | =\$      | =\$      | =\$      |
| $I_6:$    | $S \rightarrow L = R$       |               |          | \$       | \$       |
| $I_7:$    | $L \rightarrow * R \cdot$   |               | =        | =\$      | =\$      |
| $I_8:$    | $R \rightarrow L \cdot$     |               | =        | =\$      | =\$      |
| $I_9:$    | $S \rightarrow L = R \cdot$ |               |          |          | \$       |

Рис. 4.45. Вычисление предпросмотров

На рис. 4.45 показаны шаги (3) и (4) алгоритма 4.13. Столбец “Начальный” представляет произвольно сгенерированные предпросмотры для каждого базисного пункта. При первом проходе предпросмотр  $\$$  распространяется от  $S' \rightarrow \cdot S$  в  $I_0$  к шести пунктам, перечисленным на рис. 4.44. Предпросмотр  $=$  распространяется от  $L \rightarrow \cdot R$  в  $I_4$  к пунктам  $L \rightarrow \cdot R$  в  $I_7$  и к  $R \rightarrow L \cdot$  в  $I_8$ . Кроме того, он также распространяется к самому себе и к  $L \rightarrow \cdot id$  из  $I_5$ , но эти предпросмотры уже есть. При втором и третьем проходах единственный распространяемый предпросмотр —  $\$$ . При четвертом проходе распространения не происходит, и окончательное множество предпросмотров показано в последнем столбце на рис. 4.45.

Обратите внимание, что конфликт переноса/свертки, обнаруженный в примере 4.39 при использовании SLR-метода, исчезает при LALR-технологии. Причина в том, что с  $R \rightarrow L \cdot$  в  $I_2$  связан только предпросмотр  $\$$ , так что нет конфликта с переносом  $=$ , генерируемым пунктом  $S \rightarrow L \cdot = R$  из  $I_2$ .  $\square$

### Уплотнение таблиц LR-анализа

Грамматика типичного языка программирования с 50–100 терминалами и 100 продукциями может иметь таблицу LALR-анализа с несколькими сотнями состояний. Функция *action* может легко иметь 20 000 записей, каждая из которых требует минимум 8 бит. Очевидно, что немаловажной задачей является поиск более эффективного, по сравнению с двухмерной таблицей, представления информации. В этом разделе будут вкратце рассмотрены некоторые технологии, используемые для сжатия полей *action* и *goto* таблицы LR-анализа.

Одна из таких технологий уплотнения части *action* таблицы основана на том, что обычно в таблице *action* многие строки идентичны. Так, на рис. 4.40 состояния 0 и 3 имеют одинаковые записи *action*; то же можно сказать и о состояниях 2 и 6. Следовательно, пространство можно сэкономить (ценой очень небольшого повышения времени работы), создавая указатель в одномерный массив для каждого состояния. Указатели для состояний с одними и теми же действиями указывают на одно и то же место. Для получения информации из этого массива присвоим каждому терминалу номер — от нуля до числа, на единицу меньшего количества терминалов, и используем затем это целое число как смещение от значения указателя для каждого состояния. При заданном состоянии действие синтаксического анализа для  $i$ -го терминала представляет собой  $i$ -ю запись после той, на которую указывает введенный нами указатель для данного состояния.

Дальнейшее повышение эффективности использования памяти достигается путем создания списка действий для каждого состояния. Несмотря на замедление работы синтаксического анализатора, такое решение имеет смысл, поскольку LR-подобные синтаксические анализаторы обычно используют незначительную часть общего времени компиляции. Список состоит из пар (терминальный символ, действие). Наиболее часто встречающиеся действия для данного состояния могут быть размещены в конце списка; в списке можно использовать специальный термин “любой”, который означает, что если в списке не найден текущий входной символ, то действие синтаксического анализатора выполняется независимо от входного символа. Кроме того, записи ошибок можно для единообразия заменить записями сверток — это не повлияет на выявление ошибок, а просто отложит его до использования переноса.

#### Пример 4.48

Рассмотрим таблицу синтаксического анализа на рис. 4.31. Заметим, что действия для состояний 0, 4, 6 и 7 совпадают. Мы можем представить их в виде списка:

| СИМВОЛ  | ДЕЙСТВИЕ |
|---------|----------|
| id      | s5       |
| (       | s4       |
| “любой” | ошибка   |

Состояние 1 имеет похожий список:

|         |        |
|---------|--------|
| +       | s6     |
| \$      | acc    |
| “любой” | ошибка |

В состоянии 2 мы можем заменить записи ошибок действием r2, так что для любого символа, кроме \*, 2 будет выполняться свертка по продукции 2.

|         |    |
|---------|----|
| *       | s7 |
| “любой” | r2 |

Состояние 3 имеет только записи ошибок и свертки r4, поэтому можем заменить первые записи последними, и список для состояния 3 будет содержать только одну пару — (“любой”, r4). Состояния 5, 10 и 11 могут рассматриваться аналогично. Список для состояния 8 представляет собой

|         |        |
|---------|--------|
| +       | s6     |
| )       | s11    |
| “любой” | ошибка |

а для состояния 9 —

|         |    |
|---------|----|
| *       | s7 |
| “любой” | r1 |

□

Мы можем закодировать список и часть *goto* таблицы, но более эффективный путь состоит в создании списка пар для каждого нетерминала *A*. Каждая пара в списке для *A* имеет вид (*текущее состояние*, *следующее состояние*), указывающий, что *goto*[*текущее состояние*, *A*] = *следующее состояние*. Такой способ предпочтительнее по следующей причине. Переход по нетерминалу *A* может быть только состоянием, достижимым из множества пунктов, в котором у некоторых пунктов символ *A* расположен непосредственно слева от точки. Ни одно множество не имеет пунктов с *X* и *Y* непосредственно слева от точки, если *X* ≠ *Y*. Следовательно, каждое состояние появляется не более чем в одном столбце таблицы переходов, а значит, в каждом столбце, как правило, совсем мало состояний.

Для еще большей экономии памяти заметим, что к записям ошибок в таблице *goto* никогда не производится обращений. Такую запись можно заменить наиболее часто встречающейся неошибочной записью в том же столбце. Эта запись становится записью по умолчанию и представляется в списке одной парой со специальным текущим состоянием “любое”.

### Пример 4.49

Вновь рассмотрим рис. 4.31. Столбец  $F$  имеет запись 10 для состояния 7, а все остальные записи — либо 3, либо ошибка. Мы можем заменить ошибку на третье состояние и создать для столбца  $F$  следующий список:

Текущее состояние Следующее состояние

|         |    |
|---------|----|
| 7       | 10 |
| “любое” | 3  |

Аналогично список для столбца  $T$  представляет собой

|         |   |
|---------|---|
| 6       | 9 |
| “любое” | 2 |

Для столбца  $E$  можно выбрать в качестве значения по умолчанию либо 1, либо 8; в любом случае необходимы две записи. Например, создадим для столбца  $E$  следующий список:

|         |   |
|---------|---|
| 4       | 8 |
| “любое” | 1 |

□

Если читатель подсчитает число записей в списках, созданных в этом и предыдущем примерах, а затем добавит указатели от состояний в списки действий и от нетерминалов в списки следующих состояний, то экономия памяти по сравнению с таблицей на рис. 4.31 не произведет на него особого впечатления. Но нас не должен вводить в заблуждение этот маленький пример — в реальных грамматиках пространство, необходимое для представления в виде списков, обычно составляет менее 10% от пространства, требуемого для матричного представления.

Следует отметить, что методы сжатия таблиц конечных автоматов, рассмотренные в разделе 3.9, также могут использоваться для представления таблиц LR-анализа. Применение таких методов обсуждается в упражнениях.

## 4.8. Использование неоднозначных грамматик

Утверждение о том, что каждая неоднозначная грамматика не может быть LR-грамматикой и, следовательно, не может относиться ни к одному из рассмотренных ранее классов грамматик, является доказанной теоремой. Ряд типов неоднозначных грамматик, однако, вполне пригоден для определения и реализации языков, как мы увидим позже в этом разделе. Для языковых конструкций типа арифметических выражений неоднозначные грамматики обеспечивают более краткую и естественную спецификацию по сравнению с эквивалентными однозначными грамматиками. Другое использование неоднозначных грамматик состоит в выделении обычных синтаксических конструкций для специальной оптимизации. Имея неоднозначную грамматику, можно определить специальные конструкции путем добавления в грамматику новых продукции.

Следует особо отметить, что хотя используемые грамматики являются неоднозначными, во всех случаях мы задаем специальные правила разрешения неоднозначности, обеспечивающие для каждого предложения только одно дерево разбора. В этом смысле полное описание языка остается однозначным. Подчеркнем также, что неоднозначные конструкции должны использоваться как можно реже и предельно аккуратно; в противном случае нет никакой гарантии, что синтаксический анализатор распознает соответствующий язык.

## Использование приоритетов и ассоциативности для разрешения конфликтов

Рассмотрим выражения в языках программирования. Следующая грамматика, описывающая арифметические выражения с операторами + и \*,

$$E \rightarrow E+E \mid E^*E \mid (E) \mid \text{id} \quad (4.22)$$

неоднозначна, поскольку не определяет ассоциативность или приоритет операторов + и \*. Однозначная грамматика

$$\begin{aligned} E &\rightarrow E+T \mid T \\ T &\rightarrow T^*F \mid F \\ F &\rightarrow (E) \mid \text{id} \end{aligned} \quad (4.23)$$

порождает тот же язык, но придает оператору + низший приоритет по сравнению с оператором \* и делает оба оператора левоассоциативными. Имеется две причины, по которым грамматика (4.22) может оказаться предпочтительнее грамматики (4.23). Во-первых, как мы увидим позже, можно легко изменить ассоциативность и уровень приоритета операторов + и \* без изменения продукции (4.22) или числа состояний итогового синтаксического анализатора. Во-вторых, синтаксический анализатор для (4.23) будет тратить значительную часть времени на свертку по продукциям  $E \rightarrow T$  и  $T \rightarrow F$ , единственная функция которых — определять приоритеты и ассоциативность операторов.

|        |                                 |        |                                 |
|--------|---------------------------------|--------|---------------------------------|
| $I_0:$ | $E' \rightarrow \cdot E$        | $I_5:$ | $E \rightarrow E^*\cdot E$      |
|        | $E \rightarrow \cdot E+E$       |        | $E \rightarrow \cdot E+E$       |
|        | $E \rightarrow \cdot E^*E$      |        | $E \rightarrow \cdot E^*E$      |
|        | $E \rightarrow \cdot (E)$       |        | $E \rightarrow \cdot (E)$       |
|        | $E \rightarrow \cdot \text{id}$ |        | $E \rightarrow \cdot \text{id}$ |
| $I_1:$ | $E' \rightarrow E\cdot$         | $I_6:$ | $E \rightarrow (E)\cdot$        |
|        | $E \rightarrow E\cdot+E$        |        | $E \rightarrow E\cdot+E$        |
|        | $E \rightarrow E\cdot^*E$       |        | $E \rightarrow E\cdot^*E$       |
| $I_2:$ | $E \rightarrow (\cdot E)$       | $I_7:$ | $E \rightarrow E+E\cdot$        |
|        | $E \rightarrow \cdot E+E$       |        | $E \rightarrow E\cdot+E$        |
|        | $E \rightarrow \cdot E^*E$      |        | $E \rightarrow E\cdot^*E$       |
|        | $E \rightarrow \cdot (E)$       | $I_8:$ | $E \rightarrow E^*E\cdot$       |
|        | $E \rightarrow \cdot \text{id}$ |        | $E \rightarrow E\cdot+E$        |
| $I_3:$ | $E \rightarrow \text{id}\cdot$  |        | $E \rightarrow E\cdot^*E$       |
| $I_4:$ | $E \rightarrow E+\cdot E$       | $I_9:$ | $E \rightarrow (E)\cdot$        |
|        | $E \rightarrow \cdot E+E$       |        |                                 |
|        | $E \rightarrow \cdot E^*E$      |        |                                 |
|        | $E \rightarrow \cdot (E)$       |        |                                 |
|        | $E \rightarrow \cdot \text{id}$ |        |                                 |

Рис. 4.46. Множества LR(0)-пунктов для расширенной грамматики (4.22)

Множества LR(0)-пунктов для грамматики (4.22), расширенной продукцией  $E' \rightarrow E$  показаны на рис. 4.46. Поскольку грамматика (4.22) неоднозначна, при попытке построить таблицу LR-анализа по этому множеству возникают конфликты действий. В частности, такие конфликты порождают состояния, соответствующие множествам пунктов  $I_7$  и  $I_8$ . Предположим, что мы используем SLR-подход для построения таблицы действий синтаксического анализа. Конфликт порождается множеством  $I_7$  — между сверткой по продукции  $E \rightarrow E+E$  и переносом + и \* — и не может быть разрешен, поскольку +, и \* принадлежат множеству  $\text{FOLLOW}(E)$ . Таким образом, для входных символов + и \* могут выполняться оба действия. Подобный конфликт порождается множеством  $I_8$  — между сверткой  $E \rightarrow E^*E$  и переносом при входных символах + и \*. На самом деле, эти конфликты порождаются при всех способах построения таблиц LR-анализа.

Однако эти проблемы могут быть решены с использованием информации о приоритетах и ассоциативности операторов + и \*. Рассмотрим входную строку  $\text{id}+\text{id}^*\text{id}$ , которая заставляет синтаксический анализатор, основанный на рис. 4.46, попасть в состояние 7 после обработки  $\text{id}+\text{id}$ ; при этом синтаксический анализатор достигает конфигурации

| СТЕК                  | ВХОД               |
|-----------------------|--------------------|
| $0\ E\ 1\ +\ 4\ E\ 7$ | $*\ \text{id}\ \$$ |

Предполагая, что приоритет оператора \* выше приоритета +, мы знаем, что синтаксический анализатор должен выполнить перенос \* в стек, подготавливая свертку \* и соседних с этим оператором id. Эти же действия должен выполнять и SLR-анализатор на рис. 4.31, и синтаксический анализатор приоритета операторов. Однако если приоритет + выше приоритета \*, синтаксический анализатор должен свернуть и  $E+E$  к  $E$ . Таким образом, относительное старшинство +, за которым следует \*, однозначно определяет, каким образом должен быть разрешен конфликт между сверткой  $E \rightarrow E+E$  и переносом \* в состоянии 7.

Если входная строка имеет вид  $\text{id}+\text{id}+\text{id}$ , то синтаксический анализатор после обработки части строки  $\text{id}+\text{id}$  достигнет конфигурации, при которой содержимое стека —  $0\ E\ 1\ +\ 4\ E\ 7$ . При очередном входном символе + в состоянии 7 вновь возникает конфликт переноса/свертки. Однако теперь способ разрешения возникшего конфликта определяется ассоциативностью оператора +. Если этот оператор левоассоциативен, правильным действием синтаксического анализатора будет свертка по продукции  $E \rightarrow E+E$ . Первыми должны быть сгруппированы id, окружающие первый оператор +. Так поступает SLR-анализатор для грамматики из примера 4.34, и так должен работать синтаксический анализатор приоритета операторов<sup>12</sup>.

Итак, если оператор + левоассоциативен, действие в состоянии 7 для входного символа + должно приводить к свертке по продукции  $E \rightarrow E+E$ , а предположение о превосходстве приоритета оператора \* над + ведет к переносу входного символа \*. Аналогично, если оператор \* левоассоциативен и старше оператора +, то можно утверждать, что состояние 8, появляющееся на вершине стека только в том случае, когда три верхних символа представляют собой  $E^*E$ , должно приводить к свертке по продукции  $E \rightarrow E^*E$  при очередном входном символе как +, так и \*. В случае входного символа + причина заклю-

<sup>12</sup> В принципе, левоассоциативность (правоассоциативность) можно рассматривать как превосходство приоритета левого (правого) оператора одного и того же вида в случае их последовательного применения (естественно, отношения приоритетов операторов этого типа с другими операторами остаются в силе). — Прим. ред.

чается в том, что оператор \* старше оператора +, а в случае входного символа \* — что оператор \* левоассоциативен.

Продолжая рассмотрение, можно получить таблицу LR-анализа, показанную на рис. 4.47. Продукции 1–4 представляют собой, соответственно,  $E \rightarrow E+E$ ,  $E \rightarrow E^*E$ ,  $E \rightarrow (E)$  и  $E \rightarrow \text{id}$ . Интересно, что подобная таблица может быть получена устранением сверток согласно цепным продукциям  $E \rightarrow T$  и  $T \rightarrow F$  из SLR-таблицы, показанной на рис. 4.31, для грамматики (4.23). В контексте LALR- и канонического LR-анализа с неоднозначными грамматиками типа (4.22) можно поступить аналогично.

| СОСТОЯНИЕ | action |    |    |    |    |     | goto |
|-----------|--------|----|----|----|----|-----|------|
|           | id     | +  | *  | (  | )  | \$  |      |
| 0         | s3     |    |    | s2 |    |     | 1    |
| 1         |        | s4 | s5 |    |    | acc |      |
| 2         | s3     |    |    | s2 |    |     | 6    |
| 3         |        | r4 | r4 |    | r4 | r4  |      |
| 4         | s3     |    |    | s2 |    |     | 8    |
| 5         | s3     |    |    | s2 |    |     | 8    |
| 6         |        | s4 | s5 |    | s9 |     |      |
| 7         |        | r1 | s5 |    | r1 | r1  |      |
| 8         |        | r2 | r2 |    | r2 | r2  |      |
| 9         |        | r3 | r3 |    | r3 | r3  |      |

Рис. 4.47. Таблица синтаксического анализа для грамматики (4.22)

## Неоднозначность „кочующего“ else

Вновь обратимся к грамматике для условных инструкций.

$$\begin{array}{lcl} \text{stmt} & \rightarrow & \text{if } \text{expr} \text{ then stmt else stmt} \\ & | & \text{if } \text{expr} \text{ then stmt} \\ & | & \text{other} \end{array}$$

Как отмечалось в разделе 4.3, эта грамматика неоднозначна, поскольку не разрешает неоднозначности кочующего else. Для упрощения рассмотрим абстрактное представление приведенной выше грамматики, где  $i$  означает if expr then,  $e$  — else,  $a$  — “все остальные продукции”. Тогда рассматриваемая грамматика с расширяющей продукцией  $S' \rightarrow S$  может быть записана как

$$\begin{array}{lcl} S' & \rightarrow & S \\ S & \rightarrow & iSeS \mid iS \mid a \end{array} \tag{4.24}$$

Множества LR(0)-пунктов для грамматики (4.24) показаны на рис. 4.48. Неоднозначность грамматики (4.24) приводит к конфликту переноса/свертки в  $I_4$ . Здесь  $S \rightarrow iSeS$  приводит к переносу  $e$ , а поскольку  $\text{FOLLOW}(S) = \{e, \$\}$ , пункт  $S \rightarrow iS$  приводит к свертке по продукции  $S \rightarrow iS$  при входном символе  $e$ .

Если перевести это обратно в термины if...then...else, то, при наличии в стеке

if expr then stmt

и else в качестве первого входного символа, должны ли мы перенести else в стек или свернуть if expr then stmt в stmt? Ответ заключается в том, что мы должны перенести else, так

как оно связано с предыдущим **then**. В терминах грамматики (4.24), входное  $e$  может представлять собой только суффикс правой части, начинающейся с  $iS$  на вершине стека.

|        |                             |        |                              |
|--------|-----------------------------|--------|------------------------------|
| $I_0:$ | $S' \rightarrow \cdot S$    | $I_3:$ | $S \rightarrow \alpha \cdot$ |
|        | $S \rightarrow \cdot iSeS$  |        |                              |
|        | $S \rightarrow \cdot iS$    | $I_4:$ | $S \rightarrow iS \cdot eS$  |
|        | $S \rightarrow \cdot a$     |        | $S \rightarrow iS \cdot$     |
| $I_1:$ | $S' \rightarrow S \cdot$    | $I_5:$ | $S \rightarrow iSe \cdot S$  |
| $I_2:$ | $S \rightarrow i \cdot SeS$ |        | $S \rightarrow \cdot iSeS$   |
|        | $S \rightarrow i \cdot S$   |        | $S \rightarrow \cdot iS$     |
|        | $S \rightarrow \cdot iSeS$  |        | $S \rightarrow \cdot a$      |
|        | $S \rightarrow \cdot iS$    | $I_6:$ | $S \rightarrow iSeS \cdot$   |
|        | $S \rightarrow \cdot a$     |        |                              |

Рис. 4.48. LR(0)-состояния для расширенной грамматики (4.24)

Итак, мы приходим к заключению, что конфликт переноса/свертки в  $I_4$  должен быть разрешен в пользу переноса входного символа  $e$ . Таблица SLR-анализа, построенная по множествам пунктов на рис. 4.48 с использованием такого разрешения описанного конфликта, показана на рис. 4.49. Здесь продукции 1–3 представляют собой, соответственно,  $S \rightarrow iSeS$ ,  $S \rightarrow iS$  и  $S \rightarrow a$ .

| СОСТОЯНИЕ | action |     |     |      | $goto$ |
|-----------|--------|-----|-----|------|--------|
|           | $i$    | $e$ | $a$ | $\$$ |        |
| 0         | s2     |     | s3  |      | 1      |
| 1         |        |     |     | acc  |        |
| 2         | s2     |     | s3  |      | 4      |
| 3         |        | r3  |     | r3   |        |
| 4         |        | s5  |     | r2   |        |
| 5         | s2     |     | s3  |      | 6      |
| 6         |        | r1  |     | r1   |        |

Рис. 4.49. Таблица LR-анализа для абстрактной грамматики “кочующего else”

|      | СТЕК                    | ВХОД            |
|------|-------------------------|-----------------|
| (1)  | 0                       | $iiae\alpha \$$ |
| (2)  | $0 i 2$                 | $iae\alpha \$$  |
| (3)  | $0 i 2 i 2$             | $ae\alpha \$$   |
| (4)  | $0 i 2 i 2 a 3$         | $e\alpha \$$    |
| (5)  | $0 i 2 i 2 S 4$         | $e\$$           |
| (6)  | $0 i 2 i 2 S 4 e 5$     | $a\$$           |
| (7)  | $0 i 2 i 2 S 4 e 5 a 3$ | $\$$            |
| (8)  | $0 i 2 i 2 S 4 e 5 S 6$ | $\$$            |
| (9)  | $0 i 2 S 4$             | $\$$            |
| (10) | $0 S 1$                 | $\$$            |

Рис. 4.50. Действия синтаксического анализа для входной строки  $iiae\alpha$

Например, при входной строке *iiae* синтаксический анализатор выполняет действия, показанные на рис. 4.50, в соответствии с корректным разрешением проблемы кочующего *else*. В строке (5) в состоянии 4 происходит перенос входного символа *e*, а в строке (9) в состоянии (4) при входном  $\$$  выполняется свертка по продукции  $S \rightarrow iS$ .

В случае, когда мы не можем использовать неоднозначную грамматику для условных инструкций, нам придется использовать более громоздкую грамматику, записанную в (4.9).

## Неоднозначности особых продукции частных случаев

Представим заключительный пример, доказывающий полезность неоднозначных грамматик. Его суть в том, что к грамматике добавляются продукции для определения особого случая синтаксической конструкции, которая в более общем виде порождается остальной частью грамматики. Добавляя такую особую продукцию, мы порождаем конфликт. Он часто разрешается согласно правилу, гласящему о необходимости свертки по такой особой продукции. Связанные с дополнительной продукцией семантические действия позволяют провести специальную обработку особого случая конструкции.

Керниган и Черри [246] очень интересно использовали особые продукции в своем препроцессоре математических формул EQN (который использовался при работе над оригинальным изданием данной книги). В EQN синтаксис математических выражений описывается грамматикой, использующей операторы нижнего индекса *sub* и верхнего *sup*, как показано в фрагменте грамматики (4.25). Фигурные скобки используются для группирования выражений, а *c* означает токен, представляющий любую строку текста.

- (1)  $E \rightarrow E \text{ sub } E \text{ sup } E$
  - (2)  $E \rightarrow E \text{ sub } E$
  - (3)  $E \rightarrow E \text{ sup } E$
  - (4)  $E \rightarrow \{E\}$
  - (5)  $E \rightarrow c$
- (4.25)

Грамматика (4.25) неоднозначна в силу ряда причин. Она не определяет ассоциативность и приоритеты операторов *sub* и *sup*. Даже если мы разрешим эту неоднозначность, определив, например, операторы как правоассоциативные и имеющие одинаковые приоритеты, грамматика все равно останется неоднозначной. Дело в том, что продукция (1) выделяет особый случай выражений, порождаемых продукциями (2) и (3), а именно — выражения вида  $E \text{ sub } E \text{ sup } E$ . Причина особой трактовки выражений этого вида в том, что выражения типа  $a \text{ sub } i \text{ sup } 2$  набираются чаще как  $a_i^2$ , а не как  $a_i^2$  (что, кстати, может иметь различный смысл, например, в тензорном анализе. — Прим. ред.). Керниган и Черри решили эту задачу простым добавлением одной продукции.

Чтобы увидеть, как обрабатывается такая неоднозначность LR-технологией, построим SLR-анализатор для грамматики (4.25). Множества LR(0)-пунктов для этой грамматики показаны на рис. 4.51. В этой системе три множества пунктов приводят к конфликтам —  $I_7$ ,  $I_8$  и  $I_{11}$  вызывают конфликты переноса/свертки для токенов *sub* и *sup*, поскольку не определены ассоциативность и приоритет этих операторов. Этот конфликт разрешается посредством назначения операторам *sub* и *sup* одинакового приоритета и определения их как правоассоциативных; соответственно, в каждом из приведенных случаев используется перенос.

|        |                                                                                                                                                                                                                                                                                                        |                                                                                                                                                                                                                                                                                                                     |
|--------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| $I_0:$ | $E' \rightarrow \cdot E$<br>$E \rightarrow \cdot E \text{ sub } E \text{ sup } E$<br>$E \rightarrow \cdot E \text{ sub } E$<br>$E \rightarrow \cdot E \text{ sup } E$<br>$E \rightarrow \cdot \{E\}$<br>$E \rightarrow \cdot c$                                                                        | $I_6:$<br>$E \rightarrow E \cdot \text{sub } E \text{ sup } E$<br>$E \rightarrow E \cdot \text{sub } E$<br>$E \rightarrow E \cdot \text{sup } E$<br>$E \rightarrow \{E\} \cdot$                                                                                                                                     |
| $I_1:$ | $E' \rightarrow E \cdot$<br>$E \rightarrow E \cdot \text{sub } E \text{ sup } E$<br>$E \rightarrow E \cdot \text{sub } E$<br>$E \rightarrow E \cdot \text{sup } E$                                                                                                                                     | $I_7:$<br>$E \rightarrow E \cdot \text{sub } E \text{ sup } E$<br>$E \rightarrow E \text{ sub } E \cdot \text{sup } E$<br>$E \rightarrow E \cdot \text{sub } E$<br>$E \rightarrow E \text{ sub } E \cdot$<br>$E \rightarrow E \cdot \text{sup } E$                                                                  |
| $I_2:$ | $E \rightarrow \{\cdot E\}$<br>$E \rightarrow \cdot E \text{ sub } E \text{ sup } E$<br>$E \rightarrow \cdot E \text{ sub } E$<br>$E \rightarrow \cdot E \text{ sup } E$<br>$E \rightarrow \cdot \{E\}$<br>$E \rightarrow \cdot c$                                                                     | $I_8:$<br>$E \rightarrow E \cdot \text{sub } E \text{ sup } E$<br>$E \rightarrow E \cdot \text{sub } E$<br>$E \rightarrow E \cdot \text{sup } E$<br>$E \rightarrow E \text{ sup } E \cdot$                                                                                                                          |
| $I_3:$ | $E \rightarrow c \cdot$                                                                                                                                                                                                                                                                                | $I_9:$<br>$E \rightarrow \{E\} \cdot$                                                                                                                                                                                                                                                                               |
| $I_4:$ | $E \rightarrow E \text{ sub } \cdot E \text{ sup } E$<br>$E \rightarrow E \text{ sub } \cdot E$<br>$E \rightarrow \cdot E \text{ sub } E \text{ sup } E$<br>$E \rightarrow \cdot E \text{ sub } E$<br>$E \rightarrow \cdot E \text{ sup } E$<br>$E \rightarrow \cdot \{E\}$<br>$E \rightarrow \cdot c$ | $I_{10}:$<br>$E \rightarrow E \text{ sub } E \text{ sup } \cdot E$<br>$E \rightarrow E \text{ sup } \cdot E$<br>$E \rightarrow \cdot E \text{ sub } E \text{ sup } E$<br>$E \rightarrow \cdot E \text{ sub } E$<br>$E \rightarrow \cdot E \text{ sup } E$<br>$E \rightarrow \cdot \{E\}$<br>$E \rightarrow \cdot c$ |
| $I_5:$ | $E \rightarrow E \text{ sup } \cdot E$<br>$E \rightarrow \cdot E \text{ sub } E \text{ sup } E$<br>$E \rightarrow \cdot E \text{ sub } E$<br>$E \rightarrow \cdot E \text{ sup } E$<br>$E \rightarrow \cdot \{E\}$<br>$E \rightarrow \cdot c$                                                          | $I_{11}:$<br>$E \rightarrow E \cdot \text{sub } E \text{ sup } E$<br>$E \rightarrow E \text{ sub } E \text{ sup } E \cdot$<br>$E \rightarrow E \cdot \text{sub } E$<br>$E \rightarrow E \cdot \text{sup } E$<br>$E \rightarrow E \text{ sup } E \cdot$                                                              |

Рис. 4.51. Множество LR(0)-пунктов грамматики (4.25)

$I_{11}$  порождает также конфликт свертка/свертка для входных символов } и \$ между двумя продукциями

$$E \rightarrow E \text{ sub } E \text{ sup } E$$

$$E \rightarrow E \text{ sup } E$$

При считывании строки, которая свернута к  $E \text{ sub } E \text{ sup } E$ , на вершине стека будет находиться состояние  $I_{11}$ . Если конфликт свертка/свертка разрешается в пользу продукции (1), то выражение вида  $E \text{ sub } E \text{ sup } E$  обрабатывается особым образом. Используя

данное правило разрешения неоднозначности, можно получить таблицу SLR-анализа, приведенную на рис. 4.52.

| Состояние | action |     |    |    |    |     | goto |
|-----------|--------|-----|----|----|----|-----|------|
|           | sub    | sup | {  | }  | c  | \$  |      |
| 0         |        |     | s2 |    | s3 |     | 1    |
| 1         | s4     | s5  |    |    |    | acc |      |
| 2         |        |     | s2 |    | s3 |     | 6    |
| 3         | r5     | r5  |    | r5 |    | r5  |      |
| 4         |        |     | s2 |    | s3 |     | 7    |
| 5         |        |     | s2 |    | s3 |     | 8    |
| 6         | s4     | s5  |    | s9 |    |     |      |
| 7         | s4     | s10 |    | r2 |    | r2  |      |
| 8         | s4     | s5  |    | r3 |    | r3  |      |
| 9         | r4     | r4  |    | r4 |    | r4  |      |
| 10        |        |     | s2 |    | s3 |     | 11   |
| 11        | s4     | s5  |    | r1 |    | r1  |      |

Рис. 4.52. Таблица синтаксического анализа для грамматики (4.25)

Написание однозначной грамматики, которая может выделить частный случай синтаксической конструкции, — очень сложная задача. Чтобы почувствовать эту сложность, предлагаем читателю построить однозначную грамматику, эквивалентную (4.25), которая обеспечит выделение частного случая выражений вида  $E \text{ sub } E \text{ sup } E$ .

### Восстановление после ошибок при LR-анализе

LR-анализатор обнаруживает ошибку при обращении к таблице *action* синтаксического анализа и нахождении там записи об ошибке (при обращении к таблице *goto* ошибки не выявляются). В отличие от синтаксического анализатора приоритета операторов, LR-анализатор сообщает об ошибке, как только не сможет обнаружить корректного продолжения уже отсканированной части входного потока. Канонический LR-анализ в этом случае даже не выполнит ни одной свертки перед объявлением об ошибке; SLR- и LALR-анализаторы могут произвести ряд сверток перед тем, как сообщить об ошибке, но никогда не будут переносить в стек неверный символ.

При LR-анализе восстановление после ошибок “в режиме паники” реализуется следующим образом. Мы сканируем стек от вершины до состояния  $s$  с записью в таблице *goto* для некоторого нетерминала  $A$ . После этого пропускаем нуль или несколько символов входного потока, пока не будет найден символ  $a$ , который при отсутствии ошибки может следовать за  $A$ . После этого синтаксический анализатор переносит в стек состояние  $goto[s, A]$  и продолжает обычный разбор. Для выбора нетерминала  $A$  возможны несколько вариантов. Обычно это нетерминалы, представляющие крупные фрагменты программы, такие как выражение, инструкция или блок. Например, если  $A$  — нетерминал *stmt*,  $a$  может быть точкой с запятой или *end*.

При таком методе восстановления производится попытка выделить фразу, содержащую синтаксическую ошибку. Синтаксический анализатор определяет, что строка, выведенная из  $A$ , содержит ошибку. Часть этой строки уже обработана, и результат этой обработки — последовательность состояний, находящаяся на вершине стека. Остаток строки все еще находится во входном потоке, и синтаксический анализатор пытается пропустить этот остаток, находя символ, который может следовать за  $A$ . Удаляя состояния из

стека, пропуская часть входного потока и помещая в стек  $goto[s, A]$ , синтаксический анализатор полагает, что им найден экземпляр  $A$ , и продолжает обычный процесс анализа.

Восстановление на уровне фразы реализуется путем проверки каждой ошибочной записи в таблице LR-анализа и принятия решения (на основе знания особенностей языка) о том, какая наиболее вероятная ошибка программиста могла привести к данной ситуации. После этого можно построить подходящую процедуру восстановления после ошибки; возможно, при этом придется изменить вершину стека и/или первые символы входного потока способом, соответствующим данной записи ошибки.

По сравнению с синтаксическими анализаторами приоритета операторов, создание специализированных программ обработки ошибок для LR-анализатора — относительно легкая задача. В частности, не надо беспокоиться о неверных свертках — любая свертка, выполненная LR-анализатором, гарантированно корректна. Таким образом, мы можем заполнить каждую пустую ячейку в таблице *action* указателем на программу обработки ошибок, которая будет выполнять некоторые действия, определенные разработчиком компилятора. Эти действия могут включать вставку символов в стек или входной поток и удаление их оттуда или изменение и перестановку входных символов, как в случае синтаксического анализатора приоритета операторов. Как и в упомянутом случае, мы не должны допустить возможности зацикливания LR-анализатора. Стратегия, гарантирующая отсутствие зацикливания, требует либо удаления (или переноса) из входного потока по меньшей мере одного символа, либо уменьшения стека при достижении конца входного потока. Снятия со стека состояния над нетерминалом следует избегать, поскольку такое изменение удаляет из стека успешно разобранную конструкцию.

#### Пример 4.50

Обратимся еще раз к грамматике выражений

$$E \rightarrow E+E \mid E^*E \mid (E) \mid id$$

На рис. 4.53 показана таблица LR-анализа для этой грамматики, представляющая собой дополненную программами обработки ошибок таблицу на рис. 4.47. Для каждого состояния,зывающего свертку при том или ином входном символе, все записи ошибок заменены на записи некоторых сверток. Такая замена приводит к отложенному обнаружению ошибок, после выполнения одной или нескольких лишних сверток; ошибка в любом случае будет найдена до выполнения первого переноса. Пустые ячейки заполнены указателями на подпрограммы обработки ошибок.

| СОСТОЯНИЕ | <i>action</i> |    |    |    |    |     | <i>goto</i> |
|-----------|---------------|----|----|----|----|-----|-------------|
|           | <i>id</i>     | +  | *  | (  | )  | \$  |             |
| 0         | s3            | e1 | e1 | s2 | e2 | e1  | 1           |
| 1         | e3            | s4 | s5 | e3 | e2 | acc |             |
| 2         | s3            | e1 | e1 | s2 | e2 | e1  | 6           |
| 3         | r4            | r4 | r4 | r4 | r4 | r4  |             |
| 4         | s3            | e1 | e1 | s2 | e2 | e1  | 7           |
| 5         | s3            | e1 | e1 | s2 | e2 | e1  | 8           |
| 6         | e3            | s4 | s5 | e3 | s9 | e4  |             |
| 7         | r1            | r1 | s5 | r1 | r1 | r1  |             |
| 8         | r2            | r2 | r2 | r2 | r2 | r2  |             |
| 9         | r3            | r3 | r3 | r3 | r3 | r3  |             |

Рис. 4.53. Таблица LR-анализа с программами обработки ошибок

Далее приведены описания подпрограмм обработки ошибок. Следует отметить сходство этих подпрограмм и подпрограмм, представленных в примере 4.32.

- e1: /\* Эта подпрограмма вызывается из состояний 0, 2, 4 и 5; все они ожидают начало операнда — **id** или левую скобку. Вместо этого обнаруживается оператор + или \* либо окончание входного потока. \*/  
 Поместить воображаемый **id** в стек, а над ним разместить состояние 3 (переход из состояний 0, 2, 4 и 5 при входном символе **id**)<sup>13</sup>.  
 Вывести сообщение “Отсутствует операнд”.
- e2: /\* Эта подпрограмма вызывается из состояний 0, 1, 2, 4 и 5 при обнаружении правой скобки. \*/  
 Удалить правую скобку из входного потока.  
 Вывести сообщение “Несбалансированная правая скобка”.
- e3: /\* Эта подпрограмма вызывается из состояний 1 и 6, когда ожидается оператор, а обнаруживается **id** или правая скобка. \*/  
 Поместить + в стек, а над ним разместить состояние 4.  
 Вывести сообщение “Отсутствует оператор”.
- e4: /\* Эта подпрограмма вызывается из состояния 6 при обнаружении конца входного потока. В состоянии 6 ожидается появление оператора или правой скобки. \*/  
 Поместить в стек правую скобку, а над ней разместить состояние 9.  
 Вывести сообщение “Отсутствует правая скобка”.

При ошибочной входной строке **id + )**, рассмотренной в примере 4.32, последовательность конфигураций синтаксического анализатора показана на рис. 4.54. □

| СТЕК                  | ВХОДНОЙ ПОТОК    | СООБЩЕНИЯ ОБ ОШИБКАХ И ДЕЙСТВИЯ СИНТАКСИЧЕСКОГО АНАЛИЗАТОРА |
|-----------------------|------------------|-------------------------------------------------------------|
| 0                     | <b>id</b> + ) \$ |                                                             |
| 0 <b>id</b> 3         | + ) \$           |                                                             |
| 0 E 1                 | + ) \$           |                                                             |
| 0 E 1 + 4             | ) \$             |                                                             |
| 0 E 1 + 4             | \$               | “Несбалансированная правая скобка”                          |
| 0 E 1 + 4 <b>id</b> 3 | \$               | e2 удаляет правую скобку                                    |
| 0 E 1 + 4 <b>id</b> 3 | \$               | “Отсутствует операнд”                                       |
| 0 E 1 + 4 E 7         | \$               | e1 помещает в стек <b>id</b> 3                              |
| 0 E 1                 | \$               |                                                             |

Рис. 4.54. Анализ и восстановление после ошибок в LR-анализаторе

## 4.9. Генераторы синтаксических анализаторов

В этом разделе будет рассмотрен генератор синтаксических анализаторов, используемый для облегчения построения начальной фазы компилятора. Мы обратимся к генератору LALR-анализаторов Yacc, поскольку он реализует многие концеп-

<sup>13</sup> Заметим, что на практике символы грамматики в стеке не размещаются. Мы просто представляем их там, чтобы напомнить себе о символах, “представленных” состояниями.

ции из числа рассмотренных в двух предыдущих разделах, а также широко распространен. Название Yacc означает “Yet another compiler-compiler” (еще один компилятор компиляторов), что отражает популярность генераторов синтаксических анализаторов в начале 1970-х годов, когда С. Джонсоном была создана первая версия Yacc. Этот генератор доступен в UNIX и использовался при разработке сотен компиляторов.

## Генератор синтаксических анализаторов Yacc

Создание транслятора с использованием Yacc схематично показано на рис. 4.55. Вначале создается файл, скажем, `translate.y`, содержащий Yacc-спецификацию разрабатываемого транслятора. Команда UNIX

```
yacc translate.y
```

преобразует файл `translate.y` в программу `y.tab.c` на языке С с использованием LALR-метода, описанного в алгоритме 4.13. Программа `y.tab.c` является синтаксическим анализатором, написанным на языке С и объединенным с другими подпрограммами на языке С, которые могут быть подготовлены пользователем. Таблица LALR-анализа уплотнена с помощью технологии, описанной в разделе 4.7. С помощью компиляции `y.tab.c` вместе с библиотекой `ly`, содержащей программу LR-анализа, по команде

```
cc y.tab.c -ly
```

получаем требуемую объектную программу `a.out`, которая выполняет трансляцию, определенную исходной программой Yacc<sup>14</sup>. Если необходимы другие процедуры, они могут быть скомпилированы или загружены вместе с `y.tab.c`, как и с любой другой программой на языке С.

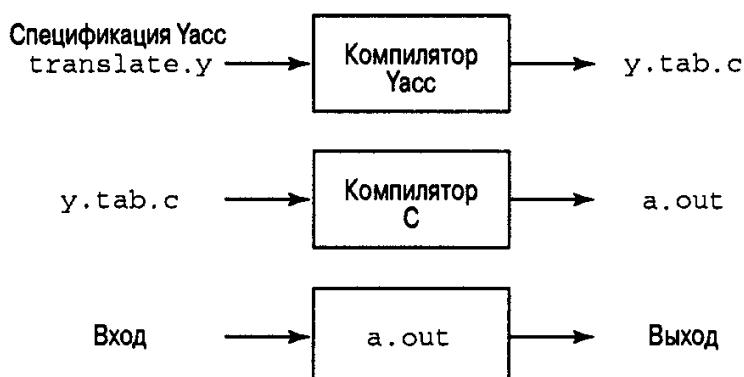


Рис. 4.55. Создание транслятора с помощью Yacc

Исходная Yacc-программа имеет три части:

Объявления

%%

Правила трансляции

%%

С-подпрограммы поддержки

<sup>14</sup> Имя библиотеки (указанное параметром `-ly`) системно зависит (т.е. может быть разным в разных системах).

## Пример 4.51

Чтобы проиллюстрировать подготовку Yacc-программы, построим простой калькулятор, который считывает арифметические выражения, вычисляет их и выводит соответствующие числовые значения. Построение калькулятора начнем со следующей грамматики для арифметических выражений:

$$\begin{array}{l} E \rightarrow E + T \mid T \\ T \rightarrow T * F \mid F \\ F \rightarrow (E) \mid \text{digit} \end{array}$$

Токен **digit** представляет отдельную цифру от 0 до 9. Yacc-калькулятор на основе этой грамматики показан на рис. 4.56.  $\square$

```
%{
#include <ctype.h>
%}

%token DIGIT

%%
line : '\n' { printf("%d\n", $1); }
;
expr : expr '+' term { $$ = $1 + $3; }
| term
;
term : term '*' factor { $$ = $1 * $3; }
| factor
;
factor : '(' expr ')'
| DIGIT
;

%%

yylex() {
 int c;
 c = getchar();
 if (isdigit(c)) {
 yylval = c - '0';
 return DIGIT;
 }
 return c;
}
```

Рис. 4.56. Yacc-спецификация простейшего калькулятора

**Объявления.** В Yacc-программе имеется два необязательных раздела объявлений. В первом размещаются обычные объявления С, ограниченные `%{` и `%}`. На рис. 4.56 этот раздел содержит только одну директиву `#include <ctype.h>`, которая заставляет препроцессор С включить стандартный заголовочный файл `<ctype.h>`, содержащий описание предиката `isdigit`.

В части объявлений специфицируются также токены грамматики. На рис. 4.56 инструкция

```
%token DIGIT
```

объявляет токен DIGIT. Токены, объявленные в этом разделе, могут использоваться во второй и третьей частях спецификации Yacc.

*Правила трансляции.* В этой части спецификации Yacc после первой пары `%` мы размещаем правила трансляции. Каждое правило состоит из продукции грамматики и связанных семантических действий. Множество продукции, которые мы записывали как

```
<Левая часть> → <Alt 1> | <Alt 2> | ... | <Alt n>
```

в Yacc записываются следующим образом:

```
<Левая часть> : <Alt 1> { Семантическое действие 1 }
 | <Alt 2> { Семантическое действие 2 }
 .
 .
 | <Alt n> { Семантическое действие n }
;
```

В продукциях Yacc отдельный символ в одинарных кавычках 'с' представляет терминальный символ с, а строка из букв и цифр, не взятая в кавычки и не объявлена как токен, считается нетерминалом. Альтернативные правые части продукции разделяются вертикальной чертой, а за каждой продукцией с ее альтернативами и семантическими действиями ставится точка с запятой. Первая левая часть считается стартовым символом.

Семантические действия Yacc представляют собой последовательности инструкций C. В них могут использоваться специальные символы: `$$` представляет значение атрибута, связанного с нетерминалом в левой части, а `$i` — значение, связанное с *i*-м грамматическим символом (терминалом или нетерминалом) справа. Семантическое действие выполняется при свертке связанной с ним продукции, так что обычно семантическое действие вычисляет `$$` по `$i`. В спецификации Yacc две *E*-продукции ( $E \rightarrow E+T \mid T$ ) и связанные с ними семантические действия выглядят как

```
expr : expr '+' term { $$ = $1 + $3; }
 | term
;
```

Заметим, что нетерминал `term` в первой продукции представляет собой третий грамматический символ (вторым является оператор '+'). Семантическое действие, связанное с первой продукцией, суммирует значения `expr` и `term` правой части продукции и присваивает полученную сумму атрибуту нетерминала `expr` в левой части. Мы опускаем семантическое действие для второй продукции, поскольку для продукции с одним символом в правой части семантическое действие по умолчанию состоит в копировании значения атрибута. В общем случае `{ $$ = $1; }` является действием по умолчанию.

Заметьте, что в спецификации Yacc добавлена новая стартовая продукция

```
line : expr '\n' { printf("%d\n", $1); }
```

Эта продукция говорит о том, что входной поток калькулятора представляет собой выражение, за которым следует символ новой строки. Семантическое действие, связанное с этой продукцией, состоит в выводе десятичного значения выражения, за которым следует символ новой строки.

*C-подпрограммы поддержки.* Третья часть спецификации Yacc содержит C-подпрограммы поддержки. Среди них в обязательном порядке должна находиться функция

лексического анализатора `yylex()`. Другие функции, такие как подпрограммы обработки ошибок, могут быть добавлены при необходимости.

Лексический анализатор `yylex()` производит пары токен-значение. Если функция возвращает токен типа `DIGIT`, этот токен должен быть объявлен в первой части спецификации Yacc. Значение связанного с токеном атрибута передается синтаксическому анализатору через предопределенную переменную Yacc `yyval`.

Лексический анализатор на рис. 4.56 очень “сырой”. Он считывает входной поток по одному символу с использованием функции `getchar()`. Если считанный символ — цифра, ее значение сохраняется в переменной `yyval`, а лексический анализатор возвращает токен `DIGIT`. В противном случае возвращается сам считанный символ.

## Использование Yacc с неоднозначной грамматикой

Модифицируем спецификацию Yacc так, чтобы полученный в результате калькулятор стал немного полезнее. Вначале обеспечим возможность работы с несколькими выражениями, по одному в строке (разрешив также пустые строки между выражениями). Для этого изменим первое правило следующим образом.

```
lines : lines expr '\n' {printf("%g\n", $2); }
 | lines '\n'
 |
 ;
```

В Yacc пустая альтернатива (третья в данном примере) обозначает  $\epsilon$ .

Теперь расширим класс обрабатываемых выражений, включив числа вместо одиночных цифр, а также все арифметические операторы — +, - (как бинарные, так и унарные), \* и /. Простейший путь определения такого класса выражений — неоднозначная грамматика

$$E \rightarrow E+E \mid E-E \mid E^*E \mid E/E \mid (E) \mid -E \mid \text{number}$$

Полученная в результате спецификация Yacc приведена на рис. 4.57.

```
%{
#include <ctype.h>
#include <stdio.h>
#define YYSTYPE double /* Тип double для стека Yacc */
%}

%token NUMBER
%left '+' '-'
%left '*' '/'
%right UMINUS

%%
lines : lines expr '\n' {printf("%g\n", $2); }
 | lines '\n'
 | /* ε */
 ;
expr : expr '+' expr { $$ = $1 + $3; }
 | expr '-' expr { $$ = $1 - $3; }
 | expr '*' expr { $$ = $1 * $3; }
 | expr '/' expr { $$ = $1 / $3; }
```

```

| '(' expr ')' { $$ = $2; }
| '-' expr %prec UMINUS { $$ = - $2; }
| NUMBER
;

%%

yylex() {
 int c;
 while((c = getchar()) == ' ')
 if ((c == '.') || (isdigit(c))) {
 ungetc(c, stdin);
 scanf("%lf", &yyval);
 return NUMBER;
 }
 return c;
}

```

*Рис. 4.57. Спецификация Yacc улучшенного калькулятора*

Поскольку данная грамматика неоднозначна, LALR-алгоритм будет генерировать конфликты действий синтаксического анализа. При этом Yacc сообщает о количестве имеющихся конфликтов. Описание множеств пунктов и конфликтов можно получить при запуске Yacc с опцией командной строки `-v`. При этой опции Yacc выводит дополнительный файл `y.output`, содержащий ядра множеств пунктов, найденные при синтаксическом анализе, описание конфликтов и удобочитаемое представление таблицы LR-анализа, показывающее, каким образом разрешаются конфликты. Если Yacc выявляет конфликты, файл `y.output` с информацией о конфликтах создается “без подсказки” со стороны пользователя.

Если Yacc не получает иных указаний в своей спецификации, все конфликты разрешаются им исходя из следующих двух правил.

1. Конфликт свертка/свертка разрешается выбором продукции, находящейся первой в спецификациях Yacc. Таким образом, для корректного разрешения неоднозначности EQN-грамматики (4.25) достаточно разместить продукцию (1) перед продукцией (3).
2. Конфликт перенос/свертка разрешается в пользу переноса. Это правило корректно разрешает конфликт, возникающий из-за неоднозначности кочующего `else`.

Поскольку эти правила по умолчанию не всегда представляют именно то, что требуется создателю компилятора, Yacc обеспечивает общий механизм для разрешения конфликтов перенос/свертка. В части объявлений терминалам можно назначить приоритеты и ассоциативность. Объявление

```
%left '+' '-'
```

задает операторы `+` и `-` как левоассоциативные, имеющие один и тот же уровень приоритета. Объявить оператор как правоассоциативный можно следующим образом:

```
%right '^'
```

Кроме того, можно обеспечить неассоциативность оператора (т.е. два оператора не могут быть скомбинированы) следующим объявлением.

```
%nonassoc '<'
```

Токены получают уровни приоритетов в том порядке, в котором они встречаются в части объявлений, начиная с наименее сложного. Таким образом, объявление

```
%right UMINUS
```

на рис. 4.57 дает токену UMINUS наивысший приоритет по сравнению с пятью предшествующими терминалами.

Yacc разрешает конфликты перенос/свертка назначением приоритета и ассоциативности каждой продукции, участвующей в конфликте (так же, как и каждому терминалу конфликтной ситуации). Если необходимо осуществить выбор между переносом входного символа  $a$  и сверткой в соответствии с продукцией  $A \rightarrow a$ , то Yacc выполняет свертку, если приоритет продукции выше приоритета  $a$ , или если приоритеты одинаковы, и ассоциативность продукции — левая. В противном случае выбирается перенос.

Обычно приоритет продукции устанавливается таким же, как у ее крайнего правого терминала. В большинстве случаев это разумное решение. Например, в продукции  $E \rightarrow E+E \mid E^*E$  мы предпочитаем свертку  $E+E$ , если очередной входной символ  $+$ , поскольку  $+$  в правой части имеет тот же приоритет, что и входной символ, и при этом он левоассоциативен. В случае очередного входного символа  $*$  мы предпочитаем перенос, поскольку входной символ имеет более высокий приоритет, чем  $+$  в продукции.

В тех ситуациях, когда крайний правый терминал не обеспечивает корректный приоритет продукции, его можно изменить путем добавления к продукции дескриптора

```
%prec <терминал>
```

При этом приоритет и ассоциативность продукции будут такими же, как у использованного в дескрипторе терминала, который, как правило, определен в разделе объявлений. Yacc не сообщает о конфликтах перенос/свертка, разрешенных с использованием механизма приоритета и ассоциативности.

Используемый “терминал” может быть искусственно введенным, как, например, UMINUS в рассматриваемом калькуляторе. Такой терминал не возвращается лексическим анализатором и объявляется исключительно для того, чтобы определить приоритет продукции. Объявление на рис. 4.57

```
%right UMINUS
```

назначает токену UMINUS наивысший приоритет, больший, чем у операторов  $*$  и  $/$ . При описании правил трансляции дескриптор

```
%prec UMINUS
```

в конце продукции

```
expr : '-' expr
```

делает унарный минус, определяемый этой продукцией, оператором, имеющим высший, по сравнению с остальными операторами, приоритет.

## Создание лексического анализатора в Yacc с помощью Lex

Lex предназначен для создания лексических анализаторов, которые могут использоваться в Yacc. Библиотека Lex 11 предоставляет подпрограмму с именем `yylex()`. Yacc использует это имя при вызове лексического анализатора. При использовании Lex в качестве генератора лексического анализатора, функция `yylex()` в третьей части спецификации Yacc заменяется инструкцией

```
#include "lex.yy.c"
```

и каждый вызов лексического анализатора возвращает известный Yacc терминал. Использование указанной инструкции обеспечивает функции `yylex()` доступ к именам, используемым Yacc для токенов, поскольку выходной файл Lex компилируется как составная часть выходного файла Yacc `y.tab.c`.

В UNIX, если спецификации Lex находятся в файле `first.l`, а спецификации Yacc — в файле `second.y`, транслятор может быть скомпилирован следующей последовательностью команд:

```
lex first.l
yacc second.y
cc y.tab.c -ly -ll
```

Приведенная на рис. 4.58 спецификация Lex может использоваться для получения лексического анализатора взамен используемого на рис. 4.57. Последний шаблон спецификаций — `\n|.`, поскольку в Lex символ “.” означает любой символ, но не символ новой строки.

```
number [0-9]+\.\.?|[0-9]*\.[0-9]+
%%
[] { /* пропуск пробелов */ }
{number} { sscanf(yytext,"%lf",&yyval);
 return NUMBER; }
\n|. { return yytext[0]; }
```

Рис. 4.58. Спецификация Lex для `yylex()` на рис. 4.57

## Восстановление после ошибок в Yacc

В Yacc восстановление после ошибок может быть выполнено с использованием специальных продукции для ошибок. Вначале следует решить, какие “крупные” нетерминалы будут иметь связанные с ними продукции ошибок. Типичным выбором является некоторое подмножество нетерминалов, порождающих выражения, инструкции, блоки и процедуры. Затем пользователь добавляет к грамматике продукции ошибок в виде  $A \rightarrow \text{error } \alpha$ , где  $A$  — “крупный” нетерминал, а  $\alpha$  — строка символов грамматики, возможно, пустая; `error` является в Yacc зарезервированным словом. Yacc генерирует синтаксический анализатор с использованием таких спецификаций, рассматривая продукцию ошибки как обычную.

Однако когда синтаксический анализатор, сгенерированный Yacc, сталкивается с ошибкой, он рассматривает состояния, множества пунктов которых содержат продукции ошибок, особым образом. При обнаружении ошибки Yacc снимает символы со стека до тех пор, пока на вершине стека не окажется состояние, множество пунктов которого включает пункт  $A \rightarrow \cdot \text{error } \alpha$ . После этого синтаксический анализатор выполняет “перенос” фиктивного токена `error` в стек, как если бы такой токен присутствовал во входном потоке.

Если  $\alpha$  представляет собой  $\epsilon$ , немедленно производится свертка в  $A$  и выполняется семантическое действие, связанное с продукцией  $A \rightarrow \text{error}$  (которое может представлять собой пользовательскую программу обработки ошибок). После этого синтаксический анализатор отбрасывает входные символы, пока не будет найден символ, позволяющий продолжить нормальный анализ.

Если  $\alpha$  — не пустая строка, то Yacc пропускает символы входного потока в поисках подстроки, которая может быть свернута в  $\alpha$ . Если  $\alpha$  полностью состоит из терминалов, Yacc ищет соответствующую строку во входном потоке и “сворачивает” ее, перенося в стек. После этого синтаксический анализатор сворачивает **error**  $\alpha$  в  $A$  и продолжает нормальный анализ.

Например, продукция ошибки

*stmt* → **error** ;

говорит синтаксическому анализатору о том, что он должен пропустить весь входной поток до символа ";" и считать, что был найден нетерминал *stmt*. Семантической подпрограмме для такой ошибки не нужно работать с входным потоком, но она может, например, вывести диагностическое сообщение и выставить флаг, запрещающий генерацию объектного кода.

### Пример 4.52

На рис. 4.59 показана спецификация калькулятора с продукцией ошибки

*lines* : **error** '\n'

Эта продукция заставляет калькулятор приостановить нормальную работу при обнаружении ошибки во входной строке. Обнаружив ошибку, синтаксический анализатор начинает снимать символы со стека, пока не встретит состояние, имеющее перенос по токену **error**. Состояние 0 является таковым (и, в нашем примере, единственным), поскольку его пункты включают

*lines* → · **error** '\n'

Состояние 0 всегда находится на дне стека. Синтаксический анализатор переносит **error** в стек и пропускает все символы входного потока, пока не обнаружит символ новой строки. Синтаксический анализатор переносит его в стек, сворачивает **error** '\n' в *lines* и выводит диагностическое сообщение “Повторите ввод последней строки:”. Специальная подпрограмма Yacc **yyerrok**<sup>15</sup> переводит синтаксический анализатор в режим обычной работы. □

```
%{
#include <ctype.h>
#include <stdio.h>
#define YYSTYPE double /* Тип double для стека Yacc */
%}

%token NUMBER
%left '+' '-'
%left '*' '/'
%right UMINUS

%%
lines : lines expr '\n' {printf("%g\n", $2); }
| lines '\n'
| /* ε */
```

<sup>15</sup> В листинге **yyerrok** приводится без скобок, указывающих, что это функция, поскольку реально **yyerrok** — макроопределение, вносимое Yacc в текст программы на языке C. — Прим. ред.

```

| error '\n' { yyerror("Повторите ввод"
| " последней строки:");
| yyerrok; }
;
expr : expr '+' expr { $$ = $1 + $3; }
| expr '-' expr { $$ = $1 - $3; }
| expr '*' expr { $$ = $1 * $3; }
| expr '/' expr { $$ = $1 / $3; }
| '(' expr ')' { $$ = $2; }
| '-' expr %prec UMINUS { $$ = - $2; }
| NUMBER
;
%%
```

```
#include "lex.yyy.c"
```

*Рис. 4.59. Калькулятор с восстановлением после ошибок*

## Упражнения

### 4.1. Рассмотрим грамматику

$$\begin{aligned} S &\rightarrow (L) \mid a \\ L &\rightarrow L, S \mid S \end{aligned}$$

- a) Что в ней являются терминалами, нетерминалами и стартовым символом?
- b) Постройте деревья разбора для следующих предложений:
  - i)  $(a, a)$
  - ii)  $(a, (a, a))$
  - iii)  $(a, ((a, a), (a, a)))$
- c) Постройте левое порождение для каждого предложения из (b).
- d) Постройте правое порождение для каждого предложения из (b).
- \*e) Какой язык порождает данная грамматика?

### 4.2. Рассмотрим грамматику $S \rightarrow aSbS \mid bSaS \mid \epsilon$ .

- a) Покажите, что эта грамматика неоднозначна, построив два разных левых порождения для предложения  $abab$ .
- b) Постройте соответствующие правые порождения для  $abab$ .
- c) Постройте соответствующие деревья разбора для  $abab$ .
- \*d) Какой язык порождает данная грамматика?

### 4.3. Рассмотрим грамматику

$$\begin{aligned} bexpr &\rightarrow bexpr \text{ or } bterm \mid bterm \\ bterm &\rightarrow bterm \text{ and } bfactor \mid bfactor \\ bfactor &\rightarrow \text{not } bfactor \mid (bexpr) \mid \text{true} \mid \text{false} \end{aligned}$$

- a) Постройте дерево разбора для предложения  $\text{not } (\text{true or false})$ .
- b) Покажите, что эта грамматика порождает все логические выражения.
- \*c) Является ли эта грамматика неоднозначной? Почему?

#### 4.4. Рассмотрим грамматику

$$R \rightarrow R' | 'R | RR | R^* | (R) | a | b$$

Обратите внимание, что первая вертикальная черта представляет собой символ “или”, а не разделитель альтернатив продукции.

a) Покажите, что эта грамматика порождает все регулярные выражения над символами  $a$  и  $b$ .

b) Покажите, что эта грамматика неоднозначна.

\*c) Постройте эквивалентную однозначную грамматику, у которой операторам  $*$ , конкатенации и  $|$  присвоены приоритеты и ассоциативность, определенные в разделе 3.3.

d) Постройте дерево разбора предложения  $a|b^*a$  для обеих грамматик.

#### 4.5. Для устранения неоднозначности кочующего `else` предлагается следующая грамматика инструкций `if-then-else`.

$$\begin{array}{lcl} stmt & \rightarrow & \text{if } expr \text{ then } stmt \\ & | & \text{matched\_stmt} \\ \text{matched\_stmt} & \rightarrow & \text{if } expr \text{ then matched\_stmt else } stmt \\ & | & \text{other} \end{array}$$

Покажите, что эта грамматика остается неоднозначной.

\*4.6. Попытайтесь разработать грамматику для каждого из следующих языков. Какие языки из приведенных являются регулярными?

- a) Множество всех строк из нулей и единиц, таких, что непосредственно за каждым 0 следует по крайней мере одна 1.
- b) Строки из нулей и единиц с одинаковым количеством нулей и единиц.
- c) Строки из нулей и единиц с неодинаковым количеством нулей и единиц.
- d) Строки из нулей и единиц, в которых отсутствует подстрока 011.
- e) Строки из нулей и единиц вида  $xy$ , где  $x \neq y$ .
- f) Строки из нулей и единиц вида  $xx$ .

4.7. Постройте грамматику для выражений каждого из следующих языков.

- a) Pascal
- b) C
- c) Fortran 77
- d) Ada
- e) Lisp

4.8. Постройте однозначную грамматику для инструкций каждого языка из упр. 4.7.

4.9. В правой части продукции грамматики можно записывать операторы, подобные используемым в регулярных выражениях. Квадратные скобки применяются для указания необязательной части продукции. Например, можно записать

$$stmt \rightarrow \text{if } expr \text{ then } stmt [\text{else } stmt]$$

для описания необязательной части `else`. В целом,  $A \rightarrow \alpha[\beta]\gamma$  эквивалентно двум продукциям —  $A \rightarrow \alpha\beta\gamma$  и  $A \rightarrow \alpha\gamma$ .

Фигурные скобки используются для обозначения фразы, которая может повторяться нуль и более раз. Например,

$stmt \rightarrow begin\ stmt\ \{\ ;\ stmt\ \}\ end$

описывает список разделенных точкой с запятой  $stmt$  между **begin** и **end**. В целом,  $A \rightarrow \alpha\{\beta\}$  эквивалентно  $A \rightarrow \alpha B \gamma$  и  $B \rightarrow \beta B |\epsilon$ .

Запись  $[\beta]$  означает регулярное выражение  $\beta | \epsilon$ , а  $\{\beta\} = \beta^*$ . Мы можем обобщить этот способ записи, допуская в правой части продукции любые регулярные выражения из символов грамматики.

- Модифицируйте приведенную выше  $stmt$ -продукцию так, чтобы в ее правой части находился завершающий точкой с запятой список из  $stmt$ .
- Приведите набор контекстно-свободных продукции, порождающих тот же набор строк, что и  $A \rightarrow B^*a(C|D)$ .
- Покажите, каким образом можно заменить любую продукцию  $A \rightarrow r$ , где  $r$  — регулярное выражение, конечным множеством контекстно-свободных продукции.

#### 4.10. Следующая грамматика порождает объявления для одиночного идентификатора.

```
stmt → declare id option_list
option_list → option_list option | ε
option → mode | scale | precision | base
mode → real | complex
scale → fixed | floating
precision → single | double
base → binary | decimal
```

- Покажите, каким образом эта грамматика может быть обобщена для  $n$  опций  $A_i$ ,  $1 \leq i \leq n$ , каждая из которых может быть либо  $a_i$ , либо  $b_i$ .
- Приведенная выше грамматика допускает повторные или противоречивые объявления, такие как

declare zap real fixed real floating

Можно добиться, чтобы синтаксис языка запрещал такие объявления. Таким образом, у нас останется конечное число синтаксически корректных последовательностей токенов. Очевидно, что эти корректные объявления образуют контекстно-свободный язык и, более того, регулярное множество. Разработайте грамматику для объявлений с  $n$  опциями, каждая из которых может использоваться только один раз.

- \*\*c) Покажите, что грамматика для п. (b) должна иметь как минимум  $2^n$  символов.
- d) Какой вывод следует из п. (c) в смысле обеспечения отсутствия повторных и противоречивых опций в объявлениях с помощью синтаксического определения языка?

- #### 4.11. a) Устранит левую рекурсию в грамматике из упр. 4.1.
- b) Постройте предиктивный синтаксический анализатор для грамматики из п. (a). Покажите работу синтаксического анализатора на примере предложений из упр. 4.1b.

- 4.12. Постройте синтаксический анализатор с откатом, работающий методом рекурсивного спуска, для грамматики из упр. 4.2. Можете ли вы построить для этой грамматики предиктивный синтаксический анализатор?
- 4.13. Грамматика  $S \rightarrow aSa \mid aa$  порождает все строки четной длины из  $a$ , за исключением пустой строки.
- Постройте для данной грамматики синтаксический анализатор с откатом, работающий методом рекурсивного спуска, который вначале проверяет альтернативу  $aSa$ , а уже затем —  $aa$ . Покажите, что процедура для  $S$  корректно обрабатывает на строках из 2, 4 и 8 символов  $a$ , но не сможет обработать строку из 6 символов  $a$ .
  - \*b) Какой язык распознает такой синтаксический анализатор?
- 4.14. Постройте предиктивный синтаксический анализатор для грамматики из упр. 4.3.
- 4.15. Постройте предиктивный синтаксический анализатор на основе однозначной грамматики для регулярных выражений из упр. 4.4.
- \*4.16. Покажите, что никакая леворекурсивная грамматика не является LL(1).
- \*4.17. Покажите, что никакая LL(1)-грамматика не может быть неоднозначной.
- 4.18. Покажите, что грамматика без  $\epsilon$ -продукций, в которой каждая альтернатива начинается со своего терминала, отличающегося от начальных терминалов всех остальных альтернатив, всегда является LL(1)-грамматикой.
- 4.19. Назовем символ  $X$  грамматики *бесполезным*, если не существует порождения вида  $S \xrightarrow{*} wXy \xrightarrow{*} wxy$ , т.е.  $X$  никогда не появляется в порождении некоторого предложения.
- Разработайте алгоритм для удаления из грамматики всех продукции, содержащих бесполезные символы.
  - Примените ваш алгоритм к грамматике
- $$S \rightarrow 0 \mid A$$
- $$A \rightarrow AB$$
- $$B \rightarrow 1$$
- 4.20. Определим  $\epsilon$ -свободную грамматику как грамматику, либо не имеющую  $\epsilon$ -продукций, либо имеющую ровно одну  $\epsilon$ -продукцию  $S \rightarrow \epsilon$ ; при этом стартовый символ  $S$  не появляется в правой части ни одной из продукции.
- Разработайте алгоритм преобразования данной грамматики в эквивалентную  $\epsilon$ -свободную грамматику. Указание: вначале определите все нетерминалы, которые могут порождать пустую строку.
  - Примените разработанный алгоритм к грамматике из упр. 4.2.
- 4.21. Назовем продукцию *цепной*, если в ее правой части содержится одиночный нетерминал.
- Разработайте алгоритм для преобразования грамматики в эквивалентную грамматику без цепных продукции.
  - Примените разработанный алгоритм к грамматике выражений (4.10).

- 4.22.** Грамматика без циклов не имеет порождений вида  $A \xrightarrow{+} A$  ни для какого нетерминала  $A$ .
- Разработайте алгоритм для преобразования грамматики в эквивалентную грамматику без циклов.
  - Примените разработанный алгоритм к грамматике  $S \rightarrow SS \mid (S) \mid \epsilon$ .
- 4.23.** a) Используя грамматику из упр. 4.1, постройте правое порождение для  $(a, (a, a))$  и покажите основы каждой правосентенциальной формы.  
 b) Покажите шаги ПС-анализатора, соответствующего правому порождению из (a).  
 c) Покажите шаги при восходящем построении дерева разбора в процессе ПС-анализа (b).
- 4.24.** На рис. 4.60 показаны отношения приоритета операторов для грамматики из упр. 4.1. Используя приведенные отношения приоритетов, разберите предложения из упр. 4.1b.

|      | $a$ | $($ | $)$      | ,   | $\$$ |
|------|-----|-----|----------|-----|------|
| $a$  |     |     | $>$      | $>$ | $>$  |
| $($  | $<$ | $<$ | $\doteq$ | $<$ |      |
| $)$  |     |     | $>$      | $>$ | $>$  |
| ,    | $<$ | $<$ | $>$      | $>$ |      |
| $\$$ | $<$ | $<$ |          |     |      |

Рис. 4.60. Отношения приоритетов операторов для грамматики из упр. 4.1

- 4.25.** Найдите функции приоритета операторов для таблицы на рис. 4.60.
- 4.26.** Имеется механический способ получения отношений приоритета операторов из операторной грамматики, даже если она содержит много различных нетерминалов. Определим  $leading(A)$  для нетерминала  $A$  как множество терминалов  $a$ , являющихся крайними слева в строках, выводимых из  $A$ , а  $trailing(A)$  — как множество терминалов  $a$ , являющихся крайними справа в тех же строках. Тогда о терминалах  $a$  и  $b$  мы говорим, что  $a \doteq b$ , если существует правая часть вида  $\alpha a \beta \gamma$ , где  $\beta$  — пустое слово или одиночный нетерминал, а  $\alpha$  и  $\gamma$  — произвольны. Говорим, что  $a < b$ , если имеется правая часть  $\alpha a A \beta$  и  $b \in leading(A)$ ;  $a > b$ , если имеется правая часть  $\alpha A b \beta$  и  $a \in trailing(A)$ . В обоих случаях  $\alpha$  и  $\beta$  являются произвольными строками. Кроме того,  $\$ < b$ , если  $b \in leading(S)$ , где  $S$  — стартовый символ, и  $a > \$$ , если  $a \in trailing(S)$ .
- Для грамматики из упр. 4.1 вычислите  $leading$  и  $trailing$  для  $S$  и  $T$ .
  - Убедитесь, что отношения приоритетов на рис. 4.60 порождаются этой грамматикой.
- 4.27.** Постройте отношения приоритета операторов для следующих грамматик.
- Грамматика из упр. 4.2.
  - Грамматика из упр. 4.3.
  - Грамматика выражений (4.10).

**4.28.** Постройте синтаксический анализатор приоритета операций для регулярных выражений.

**4.29.** Грамматика называется (однозначно обратимой) *грамматикой приоритета операторов*<sup>16</sup>, если это операторная грамматика, у которой нет двух правых частей, имеющих строки терминалов одного вида, и метод из упр. 4.26 приводит не более чем к одному отношению приоритетов для любой пары терминалов. Какая из грамматик, перечисленных в упр. 4.27, удовлетворяет этим условиям?

**4.30.** Грамматика имеет *нормальную форму Грейбах* (Greibach normal form, GNF), если она  $\epsilon$ -свободна и каждая продукция (за исключением  $S \rightarrow \epsilon$ , если таковая существует) имеет вид  $A \rightarrow a\alpha$ , где  $a$  — терминал, а  $\alpha$  — строка нетерминалов, возможно пустая.

\*\*a) Разработайте алгоритм преобразования грамматики в эквивалентную GNF-грамматику.

b) Примените разработанный алгоритм к грамматике выражений (4.10).

\***4.31.** Покажите, что любая грамматика может быть преобразована в эквивалентную операторную грамматику. Указание: вначале преобразуйте грамматику в нормальную форму Грейбах.

\***4.32.** Покажите, что любая грамматика может быть преобразована в операторную грамматику, в которой каждая продукция имеет одну из следующих форм:

$$A \rightarrow aBcC$$

$$A \rightarrow aBb$$

$$A \rightarrow aB$$

$$A \rightarrow a$$

Если в языке имеется  $\epsilon$ , то  $S \rightarrow \epsilon$  также является продукцией.

**4.33.** Рассмотрим неоднозначную грамматику

$$\begin{array}{l} S \rightarrow AS \mid b \\ A \rightarrow SA \mid a \end{array}$$

a) Постройте систему множеств LR(0)-пунктов для этой грамматики.

b) Постройте НКА, в котором каждое состояние представляет собой LR(0)-пункт из (a). Покажите, что граф переходов канонической системы LR(0)-пунктов для этой грамматики такой же, как и ДКА, построенный из НКА с использованием алгоритма построения подмножества.

c) Постройте таблицу синтаксического анализа с использованием SLR-алгоритма 4.8.

d) Покажите все шаги, определяемые таблицей из п. (c) для входной строки  $abab$ .

e) Постройте каноническую таблицу синтаксического анализа.

f) Постройте таблицу синтаксического анализа с использованием LALR-алгоритма 4.11.

g) Постройте таблицу синтаксического анализа с использованием LALR-алгоритма 4.13.

**4.34.** Постройте таблицу SLR-анализа для грамматики из упр. 4.3.

**4.35.** Рассмотрим грамматику

$$E \rightarrow E+T \mid T$$

<sup>16</sup> В русскоязычной литературе был принят термин “грамматика оперативного предшествования”. — Прим. ред.

$$\begin{array}{lcl} T & \rightarrow & TF \mid F \\ F & \rightarrow & F^* \mid a \mid b \end{array}$$

- a) Постройте таблицу SLR-анализа для этой грамматики.
- b) Постройте таблицу LALR-анализа.
- 4.36.** Выполните сжатие таблиц синтаксического анализа, построенных в упр. 4.33–4.35, используя метод из раздела 4.7.
- 4.37.** a) Покажите, что следующая грамматика является LL(1), но не SLR(1):
- $$\begin{array}{lcl} S & \rightarrow & AaAb \mid BbBa \\ A & \rightarrow & \epsilon \\ B & \rightarrow & \epsilon \end{array}$$
- \*\*b) Покажите, что каждая LL(1)-грамматика является LR(1)-грамматикой.
- \*4.38.** Покажите, что ни одна LR(1)-грамматика не может быть неоднозначной.
- 4.39.** Покажите, что следующая грамматика является LALR(1), но не SLR(1):
- $$\begin{array}{lcl} S & \rightarrow & Aa \mid bAc \mid dc \mid bda \\ A & \rightarrow & d \end{array}$$
- 4.40.** Покажите, что следующая грамматика является LR(1), но не LALR(1):
- $$\begin{array}{lcl} S & \rightarrow & Aa \mid bAc \mid Bc \mid bBa \\ A & \rightarrow & d \\ B & \rightarrow & d \end{array}$$
- \*4.41.** Рассмотрим семейство грамматик  $G_n$ , определяемое как
- $$\begin{array}{ll} S \rightarrow A_i b_i & 1 \leq i \leq n \\ A_i \rightarrow a_j A_i \mid a_j & 1 \leq i, j \leq n \text{ и } j \neq i \end{array}$$
- a) Покажите, что  $G_n$  имеет  $2n^2 - n$  производств и  $2^n + n^2 + n$  множеств LR(0)-пунктов. Что говорит этот результат о сравнении размеров LR-анализатора и грамматики?
- b) Является ли  $G_n$  SLR(1)-грамматикой?
- c) Является ли  $G_n$  LALR(1)-грамматикой?
- 4.42.** Разработайте алгоритм, по которому для каждого нетерминала  $A$  данной грамматики вычисляется множество нетерминалов  $B$ , таких, что  $A \xrightarrow{*} B\alpha$  для некоторой строки грамматических символов  $\alpha$ .
- 4.43.** Разработайте алгоритм, по которому для каждого нетерминала  $A$  данной грамматики вычисляется множество терминалов  $a$ , таких, что  $A \xrightarrow{*} aw$  для некоторой строки терминалов  $w$ , причем последний шаг порождения не использует  $\epsilon$ -продукцию.
- 4.44.** Постройте таблицу SLR-анализа для грамматики из упр. 4.4. Разрешите конфликты синтаксического анализа так, чтобы обеспечить корректный разбор регулярных выражений.
- 4.45.** Постройте SLR-анализатор для грамматики с кочующим *else* (4.7), рассматривая *expr* как терминал. Разрешите конфликт синтаксического анализа обычным способом.
- 4.46.** a) Постройте таблицу SLR-анализа для грамматики

$$E \rightarrow E \text{ sub } R \mid E \text{ sup } E \mid \{E\} \mid c$$

$$R \rightarrow E \text{ sup } E \mid E$$

Разрешите конфликт синтаксического анализа таким образом, чтобы выражения разбирались так же, как и LR-анализатором на рис. 4.52.

- b) Можно ли преобразовать все конфликты свертка/свертка, возникающие в процессе построения таблицы LR-анализа, в конфликты перенос/свертка, изменив грамматику?

\*4.47. Постройте LR-грамматику, эквивалентную грамматике (4.25), которая выделяет выражение вида  $E \text{ sub } E \text{ sup } E$  как особый случай.

\*4.48. Рассмотрим следующую неоднозначную грамматику для  $n$  бинарных инфиксных операторов:  $E \rightarrow E \theta_1 E \mid E \theta_2 E \mid \dots \mid E \theta_n E \mid (E) \mid id$

Полагаем, что все операторы левоассоциативны и приоритет  $\theta_i$  выше приоритета  $\theta_j$ , если  $i > j$ .

- a) Постройте SLR-множества пунктов для приведенной грамматики. Выразите количество множеств в виде функции от  $n$ .
- b) Постройте таблицу SLR-анализа для данной грамматики и уплотните ее с использованием представления в виде списка из раздела 4.7. Выразите общую длину всех списков, используемых в представлении, в виде функции от  $n$ .
- c) Сколько шагов требуется для разбора строки  $id \theta_i id \theta_j id$ ?

\*4.49. Повторите упр. 4.48 для однозначной грамматики

$$E_1 \rightarrow E_1 \theta_1 E_2 \mid E_2$$

$$E_2 \rightarrow E_2 \theta_2 E_3 \mid E_3$$

...

$$E_n \rightarrow E_n \theta_n E_{n+1} \mid E_{n+1}$$

$$E_{n+1} \rightarrow (E_1) \mid id$$

После решения упр. 4.48 и 4.49, что вы можете сказать об относительной эффективности синтаксических анализаторов для эквивалентных грамматик — неоднозначной и однозначной? Об относительной эффективности построения синтаксических анализаторов?

4.50. Напишите Yacc-программу, которая в качестве входного потока будет получать арифметические выражения и преобразовывать их в постфиксные.

4.51. Напишите Yacc-программу калькулятора для вычисления логических выражений.

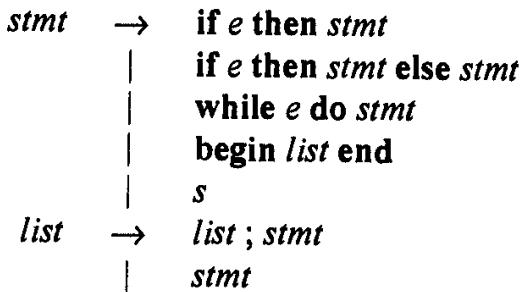
4.52. Напишите Yacc-программу, которая в качестве входного потока получает регулярные выражения и выводит их деревья разбора.

4.53. Отследите шаги, выполняемые предиктивным синтаксическим анализатором, синтаксическим анализатором приоритета операторов и LR-анализатором из упр. 4.20, 4.32 и 4.50 для следующих ошибочных строк.

a) (*id* + (\* *id* ))

b) \* + *id* ) + ( *id* \*

\*4.54. Постройте синтаксический анализатор приоритета операторов и LR-анализатор, исправляющие ошибки, для следующей грамматики.



- \*4.55. Грамматику из упр. 4.54 можно сделать LL-грамматикой, заменив продукцию для *list* следующими продукциями.

$$\begin{array}{l}
 \text{list} \rightarrow \text{stmt list}' \\
 \text{list}' \rightarrow ; \text{stmt} | \epsilon
 \end{array}$$

Постройте исправляющий ошибки предиктивный синтаксический анализатор для измененной грамматики.

- 4.56. Покажите, каким образом ведут себя синтаксические анализаторы из упр. 4.54 и 4.55 при ошибочных входных строках:

- a) **if** *e* **then** *s* ; **if** *e* **then** *s* **end**
- b) **while** *e* **do** **begin** *s* ; **if** *e* **then** *s* ; **end**

- 4.57. Разработайте предиктивный синтаксический анализатор, синтаксический анализатор приоритета операторов и LR-анализатор с восстановлением после ошибок “в режиме паники” для грамматик из упр. 4.54 и 4.55, используя точку с запятой и **end** в качестве синхронизирующих токенов. Покажите, как эти синтаксические анализаторы будут обрабатывать ошибочные входные строки из упр. 4.56.

- 4.58. В разделе 4.6 предложен метод для определения множества строк, которые могут быть сняты со стека при свертке синтаксическим анализатором приоритета операторов.

- \*a) Приведите алгоритм для поиска регулярных выражений, описывающих все такие строки.
- b) Приведите алгоритм для определения, является ли рассматриваемое множество строк бесконечным или конечным, и вывода его в последнем случае.
- c) Примените предложенные вами в пп. (a) и (b) алгоритмы к грамматике из упр. 4.54.

- \*\*4.59. К исправляющим ошибки синтаксическим анализаторам на рис. 4.18, 4.28 и 4.53 предъявлялось требование, чтобы при любой коррекции ошибок в конечном счете удалялся по меньшей мере один символ из входного потока (или, по достижении его конца, осуществлялось снятие со стека). Однако выбранные исправления не всегда приводят к немедленному удалению символа из входного потока. Можете ли вы доказать, что в этих синтаксических анализаторах бесконечные циклы невозможны? Указание: при использовании синтаксического анализатора приоритета операторов последовательные терминалы в стеке связаны отношением  $\leq$ , даже если произошла ошибка. В случае LR-анализатора даже при наличии ошибки стек содержит активный префикс.

- \*\*4.60. Приведите алгоритм обнаружения недостижимых записей в таблицах предиктивного анализа, анализа приоритета операторов и LR-анализа.

- 4.61. LR-анализатор на рис. 4.53 одинаково обрабатывает четыре ситуации, при которых верхнее состояние — 4 или 5 (возникающее при размещении на вершине стека соответственно + и \*), а следующий входной символ — + или \*. При этом вызывается подпрограмма `e1`, вставляющая между ними `id`. Легко представить себе LR-анализатор для выражений, работающий со всеми арифметическими операторами тем же способом, а именно — вставкой `id` между соседствующими операторами. В некоторых языках (таких, как PL/I или C, но не Fortran или Pascal) разумно трактовать / на вершине стека при очередном входном символе \* как особый случай, требующий отдельной обработки. Почему? Какие действия должна предпринять подпрограмма исправления ошибок в этом случае?
- 4.62. Грамматика называется грамматикой в *нормальной форме Хомского* (Chomsky normal form, CNF), если она  $\epsilon$ -свободна и каждая не- $\epsilon$ -продукция имеет вид  $A \rightarrow BC$  или  $A \rightarrow a$ .
- \*a) Приведите алгоритм преобразования грамматики в эквивалентную CNF-грамматику.
  - b) Примените предложенный вами алгоритм к грамматике выражений (4.10).
- 4.63. Разработайте алгоритм определения, верно ли для данной грамматики  $G$  в нормальной форме Хомского и входной строки  $w=a_1a_2\dots a_n$ , что  $w \in L(G)$ . Указание: с помощью динамического программирования заполните таблицу  $T$  размером  $n \times n$ , в которой  $T[i, j] = \{A \mid A \xrightarrow{*} a_i a_{i+1} \dots a_j\}$ . Входная строка  $w \in L(G)$  тогда и только тогда, когда  $S \in T[1, n]$ .
- \*4.64. a) Для данной CNF-грамматики  $G$  покажите, каким образом добавить в нее продукции для ошибок однократного удаления, вставки и замены символов, чтобы расширенная грамматика порождала все возможные строки токенов.
- b) Модифицируйте алгоритм синтаксического анализа из упр. 4.63 так, чтобы для произвольной строки  $w$  он находил разбор, использующий минимальное число продукции ошибок.
- 4.65. Разработайте Yacc-анализатор для арифметических выражений, использующий механизм восстановления после ошибок из примера 4.50.

## Библиографические примечания

В знаменитом сообщении об Algol 60 [328] использовалась форма Бэкуса-Наура. Эквивалентность БНФ и контекстно-свободных грамматик была быстро замечена, и в 1960-х годах теория формальных языков стала объектом пристального внимания. Ее основы исчерпывающе изложены в [194].

Методы синтаксического анализа стали гораздо более систематичными после развития контекстно-свободных грамматик. Был разработан ряд общих технологий для анализа контекстно-свободных языков; одной из первых являлась методика динамического программирования из упр. 4.63, независимо открытая Коком, Янгером [462] и Касами [232]. Эрли в своей диссертации [118] разработал еще один универсальный алгоритм синтаксического анализа контекстно-свободных грамматик. Детальное рассмотрение этих и других методов можно найти в работах [18, 19].

В компиляторах используется множество различных методов синтаксического анализа. В работе [401] описан метод синтаксического анализа, использованный в самом первом компиляторе Fortran, где вводились дополнительные скобки для операторов, чтобы обеспечить разбор выражений. Идея приоритета операторов и использования функций приоритетов принадлежит Флойду [135]. В 1960-х годах были предложены различные стратегии восходящего разбора, исходные версии которых можно найти в работах [136, 165, 203, 311, 458].

Анализ методом рекурсивного спуска и предиктивный синтаксический анализ широко применялись на практике. Благодаря своей гибкости синтаксический анализ методом рекурсивного спуска использовался во многих ранних системах разработки компиляторов, таких как META [393] и TMG [306]. Решение упр. 4.13 может быть найдено в работе Бирмана и Ульмана [57] вместе с теоретическим обоснованием этого метода синтаксического анализа. Метод нисходящего синтаксического анализа приоритета операторов был предложен Праттом [353].

Изучение LL-грамматик можно найти в [289, 382]. Предиктивные анализаторы подробно рассмотрены в [255]; их использование в компиляторах описано в работе [288]. Алгоритмы преобразования грамматик в LL(1)-грамматики представлены в работах [140, 407, 409, 459].

LR-грамматики и синтаксические анализаторы были введены Кнутом [253], который описал построение канонических таблиц LR-анализа. LR-метод не рассматривался как практический до тех пор, пока Кореньяк [265] не показал, что с его помощью можно разработать синтаксический анализатор вполне разумного размера для разбора грамматики языка программирования. После разработки де Ремером [106, 107] SLR- и LALR-методов, которые оказались проще предложенного Кореньяком, LR-технология стала основным методом автоматической генерации синтаксических анализаторов. Сегодня использование генераторов LR-анализаторов при разработке компиляторов — повсеместное явление.

Множество научных исследований существенно расширили возможности и применимость технологии LR-анализа; в первую очередь следует упомянуть такие работы, как [11, 20, 34, 38, 109, 119, 223, 338, 406, 433]. То же относится и к LALR-технологиям, развитым в работах [34, 108, 270, 272, 337, 342]. Общие обзоры LR-анализа, алгоритмов, лежащих в основе Yacc, включая использование продукции ошибок, имеются в работах [9, 18, 19].

Наряду с технологиями синтаксического анализа было разработано множество различных методов восстановления после ошибок, обзор которых можно найти в работах [84, 404]. Грамматический подход к восстановлению после ошибок предложен Айронсом [207]; продукции ошибок использованы Виртом [455] в компиляторе PL360. Стратегия восстановления на уровне фразы предложена в работе [280]. Следует также упомянуть о важных для развития этой области синтаксического анализа работах [15, 169, 170, 302]. Кроме того, можно порекомендовать обратиться к таким работам, как [195], где рассматривается вопрос качества сообщений об ошибках; [405], в которой сравниваются методы восстановления из работ [362] и [345]; [168], где представлена LR-технология восстановления после ошибок; и [339], которая посвящена технологии восстановления “глобального контекста”.

Исправление ошибок в процессе синтаксического анализа обсуждается в работах [94, 95, 286, 325, 378, 419]. Решение упр. 4.63 имеется в работе Ахо и Петерсона [15].

# ГЛАВА 5

## Синтаксически управляемая трансляция

В данной главе развивается затронутая в разделе 2.3 тема — трансляция языков, управляемая контекстно-свободной грамматикой. Мы связываем информацию с конструкциями языка программирования с помощью атрибутов грамматических символов, представляющих данную конструкцию. Значения атрибутов вычисляются согласно “семантическим правилам”, связанным с продукциями грамматики.

Существует два вида записи для связанных с продукциями семантических правил — синтаксически управляемые определения и схемы трансляции. Синтаксически управляемые определения представляют собой высокоуровневые спецификации трансляции, скрывающие множество деталей реализации и освобождающие пользователя от явного указания порядка выполнения трансляции. Схемы трансляции указывают порядок, в котором выполняются семантические правила; так что эти схемы показывают определенную часть деталей реализации. Обе записи используются в главе 6, “Проверка типов”, для указания семантических проверок, в частности определения типов, и в главе 8, “Генерация промежуточного кода”, для генерации промежуточного кода.

Концептуально при обоих методах мы разбираем входной поток токенов, строим дерево разбора и обходим его так, как необходимо для выполнения семантических правил в узлах дерева разбора (рис. 5.1). Выполнение семантических правил может генерировать код, сохранять информацию в таблице символов, выводить сообщения об ошибках или выполнять какие-либо другие действия. Результат трансляции потока токенов будет получен путем выполнения указанных семантических правил.



Рис. 5.1. Концепция синтаксически управляемой трансляции

Реализация не всегда следует приведенной на рис. 5.1 схеме. Частные случаи синтаксически управляемых определений могут быть реализованы за один проход выполнением семантических правил в процессе синтаксического анализа, без явного построения дерева разбора или графа, показывающего взаимосвязи между атрибутами. Поскольку однопроходная реализация важна с точки зрения скорости компиляции, большая часть этой главы посвящена изучению именно таких частных случаев. Один важный подкласс синтаксически управляемых определений, именуемый L-атрибутными определениями, включает практически все трансляции, которые могут выполняться без явного построения дерева разбора.

## 5.1. Синтаксически управляемые определения

Синтаксически управляемое определение представляет собой обобщение контекстно-свободной грамматики, в которой каждый грамматический символ имеет связанное множество атрибутов, разделенное на два подмножества, — синтезируемые и наследуемые атрибуты этого грамматического символа. Если рассматривать узел грамматического символа в дереве разбора как запись с полями для хранения информации, то атрибут соответствует имени поля.

Атрибут может представлять собой все, что угодно, — строку, число, тип, адрес памяти и т.д. Значение атрибута в узле дерева разбора определяется семантическими правилами, связанными с используемой в данном узле продукцией. Значение синтезируемого атрибута в узле вычисляется по значениям атрибутов в дочерних по отношению к данному узлах; значения наследуемых атрибутов определяются значениями атрибутов соседних (т.е. узлов, дочерних по отношению к родительскому узлу данного) и родительского узлов.

Семантические правила определяют зависимости между атрибутами, которые представляются графом. Граф зависимости определяет порядок выполнения семантических правил, что в свою очередь дает значения атрибутов в узлах дерева разбора входной строки. Семантические правила могут иметь и побочные действия, например вывод значения или изменение глобальной переменной. Естественно, при реализации не обязательно явным образом строить дерево разбора или граф зависимостей; главное, чтобы для каждой входной строки выполнялись верные действия в правильном порядке.

Дерево разбора, показывающее значения атрибутов в каждом узле, называется *анnotated*, а процесс вычисления значений атрибутов в узлах дерева — *аннотированием* дерева разбора.

### Вид синтаксически управляемого определения

В синтаксически управляемом определении каждая продукция грамматики  $A \rightarrow \alpha$  имеет связанное с ней множество семантических правил вида  $b := f(c_1, c_2, \dots, c_k)$ , где  $f$  — функция,  $c_1, c_2, \dots, c_k$  — атрибуты грамматических символов продукции, а  $b$  — синтезируемый атрибут символа  $A$  или наследуемый атрибут одного из грамматических символов правой части продукции.

В любом случае мы говорим, что атрибут  $b$  зависит от атрибутов  $c_1, c_2, \dots, c_k$ . Атрибутная грамматика является синтаксически управляемым определением, в котором функции в семантических правилах не имеют побочных эффектов.

Функции в семантических правилах зачастую записываются как выражения. Случается, что единственная цель семантического правила в синтаксически управляемом определении состоит именно в создании побочного эффекта. Такие семантические правила записываются как вызовы процедур или фрагменты программ. Их можно рассматривать как правила, определяющие значения фиктивных синтезируемых атрибутов нетерминала в левой части связанной продукции; фиктивный атрибут и знак присвоения  $:=$  при этом не указываются.

### Пример 5.1

На рис. 5.2 приведено синтаксически управляемое определение программы настольного калькулятора. Это определение связывает с каждым из нетерминалов  $E$ ,  $T$  и  $F$  целочисленный синтезируемый атрибут  $val$ . Для каждой  $E$ -,  $T$ - и  $F$ -продукции семантическое правило вычисляет значение атрибута  $val$  нетерминала из левой части продукции по значениям атрибутов нетерминалов правой части.

| ПРОДУКЦИЯ                    | СЕМАНТИЧЕСКИЕ ПРАВИЛА                                |
|------------------------------|------------------------------------------------------|
| $L \rightarrow E \text{ n}$  | $print(E.\text{val})$                                |
| $E \rightarrow E_1 + T$      | $E.\text{val} := E_1.\text{val} + T.\text{val}$      |
| $E \rightarrow T$            | $E.\text{val} := T.\text{val}$                       |
| $T \rightarrow T_1 * F$      | $T.\text{val} := T_1.\text{val} \times F.\text{val}$ |
| $T \rightarrow F$            | $T.\text{val} := F.\text{val}$                       |
| $F \rightarrow ( E )$        | $F.\text{val} := E.\text{val}$                       |
| $F \rightarrow \text{digit}$ | $F.\text{val} := \text{digit}.lexval$                |

Рис. 5.2. Синтаксически управляемое определение простого калькулятора

Токен **digit** имеет синтезируемый атрибут *lexval*, значение которого предоставляется лексическим анализатором. Правило, связанное с продукцией  $L \rightarrow E \text{ n}$  для стартового нетерминала  $L$ , представляет собой процедуру вывода значения арифметического выражения, порождаемого  $E$  (это правило можно рассматривать как определяющее фиктивный атрибут для нетерминала  $L$ ). Спецификация Yacc этого калькулятора, иллюстрирующая трансляцию в процессе LR-разбора, представлена на рис. 4.56.  $\square$

В синтаксически управляемом определении предполагается, что терминалы могут иметь только синтезируемые атрибуты, поскольку определение не дает никаких семантических правил для терминалов (обычно значения атрибутов терминалов предоставляются лексическим анализатором, как описывалось в разделе 3.1). Кроме того, если иное не оговорено особо, стартовый символ не имеет наследуемых атрибутов.

## Синтезируемые атрибуты

Синтезируемые атрибуты очень часто используются на практике. Синтаксически управляемое определение, использующее только синтезируемые атрибуты, называется *S-атрибутным определением*. Дерево разбора для S-атрибутного определения всегда может быть аннотировано путем выполнения семантических правил для атрибутов в каждом узле снизу вверх, от листьев к корню. В разделе 5.3 описывается, как генератор LR-анализаторов может быть адаптирован для механической реализации S-атрибутного определения, основанного на LR-грамматике.

### Пример 5.2

S-атрибутное определение в примере 5.1 описывает калькулятор,читывающий арифметическое выражение из цифр, скобок, операторов  $+$  и  $*$ , за которым следует символ новой строки **n**, и выводит значение выражения. Например, получив выражение  $3 * 5 + 4$ , за которым следует символ новой строки, программа выводит значение 19. На рис. 5.3 показано аннотированное дерево разбора для входной строки  $3 * 5 + 4 \text{n}$ . Выход программы, печатаемый в корне дерева, представляет собой значение  $E.\text{val}$  в первом дочернем узле корня дерева.

Чтобы увидеть, как производится вычисление значений атрибутов, рассмотрим внутренний крайний слева снизу узел, соответствующий использованию продукции  $F \rightarrow \text{digit}$ . Семантическое правило  $F.\text{val} := \text{digit}.lexval$  определяет атрибут  $F.\text{val}$  в этом узле как имеющий значение 3, поскольку значение  $\text{digit}.lexval$  в дочернем узле равно 3. Аналогично в родительском по отношению к данному узлу атрибут  $T.\text{val}$  также имеет значение 3.

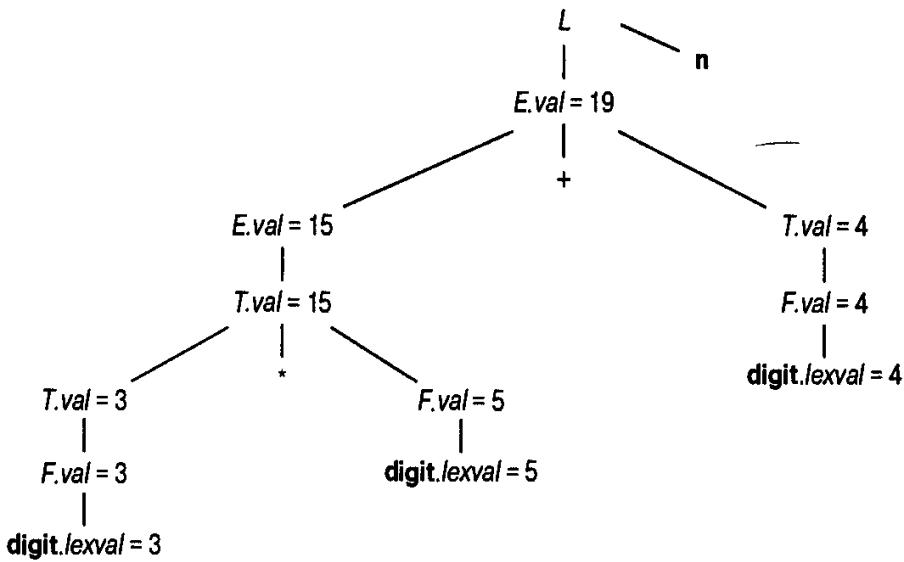


Рис. 5.3. Аннотированное дерево разбора для  $3 * 5 + 4\pi$

Теперь рассмотрим узел продукции  $T \rightarrow T * F$ . Значение атрибута  $T.val$  в этом узле определяется следующим образом:

ПРОДУКЦИЯ      СЕМАНТИЧЕСКОЕ ПРАВИЛО

$T \rightarrow T_1 * F \quad T.val := T_1.val \times F.val$

При использовании этого семантического правила в данном узле  $T_1.val$  имеет значение 3, полученное от левого наследника, а  $F.val$  — значение 5, полученное от правого наследника. Следовательно, в этом узле  $T.val$  составляет 15.

Правило, связанное с продукцией для стартового нетерминала  $L \rightarrow E \pi$ , выводит значение выражения, порожденного  $E$ .  $\square$

## Наследуемые атрибуты

Наследуемые атрибуты представляют собой атрибуты, значения которых в узле дерева разбора определяются атрибутами родительского и/или дочерних по отношению к родительскому узлов. Наследуемые атрибуты удобны для выражения зависимости конструкций языка программирования от контекста, в котором они появляются. Например, мы можем использовать наследуемые атрибуты для отслеживания, появляется ли идентификатор слева или справа от знака присвоения, чтобы определить, потребуется ли адрес или значение данного идентификатора. Хотя всегда можно переписать синтаксически управляемое определение таким образом, чтобы использовать только синтезируемые атрибуты, зачастую более естественно воспользоваться синтаксически управляемым определением с наследуемыми атрибутами.

В следующем примере наследуемый атрибут распространяет информацию о типе на различные идентификаторы в объявлении.

### Пример 5.3

Объявление, порожданное нетерминалом  $D$  в синтаксически управляемом определении на рис. 5.4, состоит из ключевого слова `int` или `real`, за которым следует список идентификаторов. Нетерминал  $T$  имеет синтезируемый атрибут `type`, значение которого определяется ключевым словом объявления. Семантическое правило  $L.in := T.type$ , связанное с продукцией  $D \rightarrow T L$ , определяет наследуемый атрибут  $L.in$  как тип объявления. Затем приведенные правила распространяют этот тип вниз по дереву разбора с исполь-

зованием атрибута  $L.in$ . Правила, связанные с продукциями для  $L$ , вызывают процедуру  $addtype$  для добавления типа каждого идентификатора к его записи в таблице символов (определенной атрибутом  $entry$ ).

| ПРОДУКЦИЯ               | СЕМАНТИЧЕСКИЕ ПРАВИЛА                         |
|-------------------------|-----------------------------------------------|
| $D \rightarrow TL$      | $L.in := T.type$                              |
| $T \rightarrow int$     | $T.type := integer$                           |
| $T \rightarrow real$    | $T.type := real$                              |
| $L \rightarrow L_1, id$ | $L_1.in := L.in$<br>$addtype(id.entry, L.in)$ |
| $L \rightarrow id$      | $addtype(id.entry, L.in)$                     |

Рис. 5.4. Синтаксически управляемое определение с наследуемым атрибутом  $L.in$

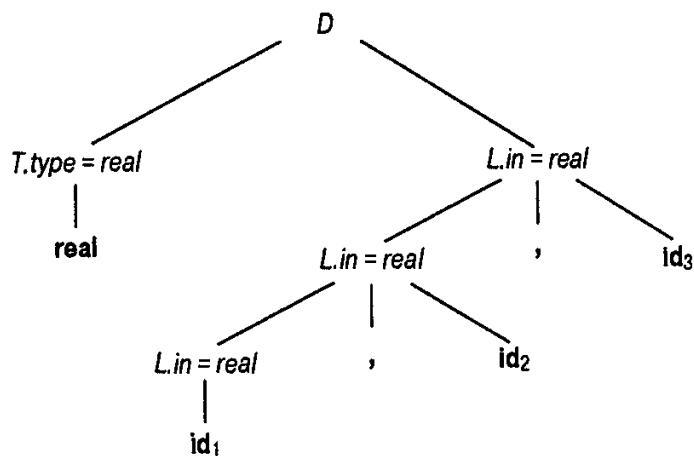


Рис. 5.5. Дерево разбора с наследуемыми атрибутами  $in$  в каждом узле, помеченном  $L$

На рис. 5.5 показано аннотированное дерево разбора для предложения **real**  $id_1$ ,  $id_2$ ,  $id_3$ . Значение  $L.in$  в трех  $L$ -узлах дает тип идентификаторов  $id_1$ ,  $id_2$  и  $id_3$ . Эти значения определяются вычислением значения атрибута  $T.type$  в левом дочернем по отношению к корню узле, а затем вычислением  $L.in$  в исходящем порядке в трех  $L$ -узлах правого поддерева корневого узла. В каждом  $L$ -узле мы вызываем также процедуру  $addtype$  для записи в таблицу символов информации о том, что идентификатор в правом дочернем узле имеет тип **real**. □

## Графы зависимости

Если атрибут  $b$  в узле дерева разбора зависит от атрибута  $c$ , то семантическое правило для  $b$  в этом узле должно выполняться после семантического правила, определяющего  $c$ . Зависимости между наследуемыми и синтезируемыми атрибутами в узлах дерева разбора могут быть показаны с помощью направленного графа, называемого *графом зависимости* (dependency graph).

Перед построением графа зависимости для дерева разбора мы приводим каждое семантическое правило к виду  $b := f(c_1, c_2, \dots, c_k)$  введением фиктивного синтезируемого атрибута  $b$  для каждого семантического правила, состоящего из вызова процедуры (или представляющего собой фрагмент кода — *Прим. ред.*). Граф имеет узел для каждого атрибута и дугу, ведущую в узел  $b$  из узла  $c$ , если атрибут  $b$  зависит от атрибута  $c$ . Более детально график для данного дерева разбора строится следующим образом.

```

for каждый узел p в дереве разбора do
 for каждый атрибут a грамматического символа в узле p do
 создать узел для a в графе зависимости;
for каждый узел p в дереве разбора do
 for каждое семантическое правило $b := f(c_1, c_2, \dots, c_k)$
 связанное с продукцией, используемой в p do
 for $i := 1$ to k do
 построить дугу от узла для c_i к узлу для b ;

```

Предположим, что  $A.a := f(X.x, Y.y)$  является семантическим правилом для продукции  $A \rightarrow XY$ . Это правило определяет синтезируемый атрибут  $A.a$ , который зависит от атрибутов  $X.x$  и  $Y.y$ . Если эта продукция используется в дереве разбора, то в графе зависимости будет три узла  $A.a$ ,  $X.x$  и  $Y.y$  с дугами от  $X.x$  к  $A.a$ , поскольку  $A.a$  зависит от  $X.x$ , и от  $Y.y$  к  $A.a$ , поскольку  $A.a$  зависит от  $Y.y$ .

Если продукция  $A \rightarrow XY$  имеет семантическое правило  $X.i := g(A.a, Y.y)$ , связанное с ней, то в графе зависимости будут дуги от  $A.a$  к  $X.i$  и  $Y.y$  к  $X.i$ , поскольку  $X.i$  зависит как от  $A.a$ , так и  $Y.y$ .

#### Пример 5.4

Когда в дереве разбора используется следующая продукция, мы добавляем показанные на рис. 5.6 дуги в граф зависимости.

|                           |                              |
|---------------------------|------------------------------|
| ПРОДУКЦИЯ                 | СЕМАНТИЧЕСКОЕ ПРАВИЛО        |
| $E \rightarrow E_1 + E_2$ | $E.val := E_1.val + E_2.val$ |

Три узла, помеченные в графе зависимости символом  $\bullet$ , представляют синтезируемые атрибуты  $E.val$ ,  $E_1.val$  и  $E_2.val$  в соответствующих узлах дерева разбора. Дуги, направленные от  $E_1.val$  и  $E_2.val$  к  $E.val$ , показывают, что  $E.val$  зависит от  $E_1.val$  и  $E_2.val$ . Пунктирные линии на рисунке представляют дерево разбора и не являются частью графа зависимости.  $\square$

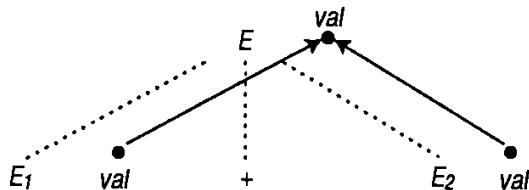


Рис. 5.6. Атрибут  $E.val$  синтезирован из  $E_1.val$  и  $E_2.val$

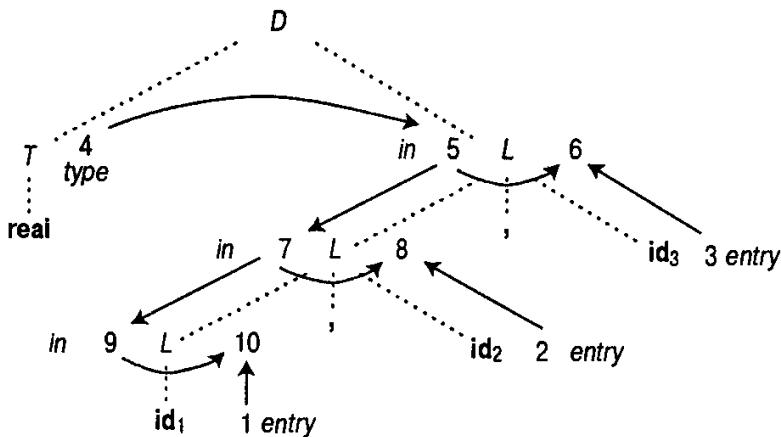


Рис. 5.7. Граф зависимости для дерева разбора на рис. 5.5

### Пример 5.5

На рис. 5.7 показан граф зависимости для дерева разбора на рис. 5.5. Узлы в графе зависимости помечены числами, которые будут использоваться далее в тексте. Имеется дуга из узла 4 для  $T.type$  в узел 5  $L.in$ , поскольку наследуемый атрибут  $L.in$  зависит от атрибута  $T.type$  согласно семантическому правилу  $L.in := T.type$  для продукции  $D \rightarrow TL$ . Две исходящие дуги в узлы 7 и 9 возникают в силу зависимости  $L_1.in$  от  $L.in$  в соответствии с семантическим правилом  $L_1.in := L.in$  для продукции  $L \rightarrow L_1, id$ . Каждое из семантических правил  $addtype(id.entry, L.in)$ , связанных с  $L$ -продукциями, приводит к созданию фиктивного атрибута. Узлы 6, 8 и 10 построены для таких фиктивных атрибутов.  $\square$

### Порядок выполнения

*Топологическая сортировка* направленного ациклического графа является упорядочением узлов графа  $m_1, m_2, \dots, m_k$ , таким, что дуги направлены от узлов, расположенных в упорядоченной последовательности раньше, к узлам, расположенным позже, т.е. если  $m_i \rightarrow m_j$  — дуга от  $m_i$  к  $m_j$ , то  $m_i$  встречается в упорядоченной последовательности раньше  $m_j$ .

Любая топологическая сортировка графа зависимости дает правильный порядок выполнения семантических правил, связанных с узлами дерева разбора, т.е. к моменту выполнения семантического правила  $b := f(c_1, c_2, \dots, c_k)$  атрибуты  $c_1, c_2, \dots, c_k$  доступны для вычисления  $f$ .

Трансляция, определяемая синтаксически управляемым определением, может быть уточнена следующим образом. Вначале для построения дерева разбора входного потока используется грамматика, лежащая в основе синтаксически управляемого определения. Затем строится граф зависимости, после топологической сортировки которого мы получаем порядок выполнения семантических правил. Выполнение этих правил в полученном порядке приводит к трансляции входной строки.

### Пример 5.6

Каждая из дуг графа зависимости на рис. 5.7 направлена от узла с меньшим номером к узлу с большим номером. Следовательно, топологическая сортировка графа зависимости может быть выполнена просто записью узлов в порядке их номеров. После такой топологической сортировки мы получим следующую программу (здесь  $a_n$  означает атрибут, связанный с узлом номер  $n$  в графе зависимости).

```
a4 := real;
a5 := a4;
addtype(id3.entry, a5);
a7 := a5;
addtype(id2.entry, a7);
a9 := a7;
addtype(id1.entry, a9);
```

Выполнение этих семантических правил вносит тип *real* в записи таблицы символов для каждого из идентификаторов.  $\square$

Для выполнения семантических правил был предложен ряд различных методов.

1. *Методы дерева разбора.* Порядок выполнения семантических правил определяется во время компиляции с помощью топологической сортировки графа зависимости, построенного по дереву разбора для каждой входной строки. Эти методы не позво-

ляют определить порядок выполнения только в том случае, когда граф зависимости для конкретного дерева разбора имеет цикл.

2. *Методы, основанные на правилах.* В процессе создания компилятора анализируются семантические правила, связанные с продукциями, либо вручную, либо с помощью специализированного инструментария. Порядок вычисления атрибутов, связанных с каждой продукцией, предопределяется в процессе разработки компилятора.
3. *Игнорирующие методы.* Выбор порядка выполнения происходит без рассмотрения семантических правил. Например, если трансляция происходит в процессе синтаксического анализа, то порядок выполнения задается методом синтаксического анализа, независимо от семантических правил. Игнорирующие методы ограничивают класс реализуемых синтаксически управляемых определений.

Методы, основанные на правилах, и игнорирующие методы не требуют явного построения графа зависимости в процессе компиляции, так что они могут оказаться более эффективными с точки зрения времени работы и используемой памяти.

Синтаксически управляемое определение называется *циклическим* (*circular*), если граф зависимости для некоторого дерева разбора, порождаемого грамматикой определения, имеет цикл. В разделе 5.10 обсуждается, каким образом можно проверить цикличность синтаксически управляемого определения.

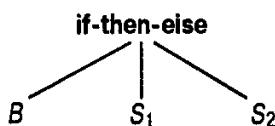
## 5.2. Построение синтаксических деревьев

В этом разделе мы покажем, каким образом для построения синтаксических деревьев и других графических представлений языковых конструкций могут использоваться синтаксически управляемые определения.

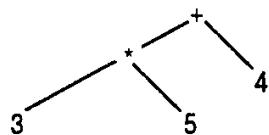
Использование синтаксических деревьев в качестве промежуточного представления позволяет отделить трансляцию от синтаксического анализа. Программы трансляции, вызываемые в процессе синтаксического анализа, должны соответствовать двум видам ограничений. Во-первых, грамматика, пригодная для синтаксического анализа, может не отражать естественную иерархическую структуру конструкций языка. Например, в грамматике для языка Fortran подпрограмма может рассматриваться просто как список инструкций, однако анализ подпрограмм может быть упрощен при использовании представления в виде дерева, отражающего вложенность циклов DO. Во-вторых, метод синтаксического анализа ограничивает порядок рассмотрения узлов дерева разбора. Этот порядок может не соответствовать порядку, в котором становится доступной информация о конструкциях. По этой причине компиляторы С обычно строят синтаксические деревья для объявлений.

### Синтаксические деревья

(Абстрактное) синтаксическое дерево представляет собой дерево разбора в сжатом виде, удобном для представления языковых конструкций. Продукция  $S \rightarrow \text{if } B \text{ then } S_1 \text{ else } S_2$  может быть представлена в синтаксическом дереве как



В синтаксическом дереве операции и ключевые слова не представлены в качестве листьев, а связываются с внутренним узлом, который выступает в дереве разбора родительским для этих листьев. Другое упрощение, используемое в синтаксических деревьях, заключается в том, что цепочки для одиночных продуктов могут быть свернуты; так, дерево разбора, показанное на рис. 5.3, превращается в следующее синтаксическое дерево.



Синтаксически управляемая трансляция может основываться как на синтаксических деревьях, так и на деревьях разбора. В обоих случаях мы присоединяем атрибуты к узлам дерева.

## Построение синтаксических деревьев для выражений

Построение синтаксических деревьев для выражений подобно трансляции выражений в постфиксный вид. Мы строим поддеревья для подвыражений путем создания узлов для каждой операции и operandов. Узлы, дочерние по отношению к узлу операции, являются корнями поддеревьев для подвыражений, образующих operandы данной операции.

Каждый узел в синтаксическом дереве может быть реализован как запись с несколькими полями. В узле операции одно поле идентифицирует операцию, а остальные поля содержат указатели на узлы operandов. Знак операции часто называется *меткой* узла. Узлы синтаксического дерева, используемого при трансляции, могут иметь дополнительные поля для хранения значений (или указателей на значения) атрибутов, связанных с узлом. В этом разделе мы используем следующие функции для создания узлов синтаксических деревьев выражений с бинарными операциями. Каждая функция возвращает указатель на вновь созданный узел.

1. *mknode(op, left, right)* создает узел операции с меткой *op* и двумя полями, содержащими указатели на *left* и *right*.
2. *mkleaf(id, entry)* создает узел идентификатора с меткой *id* и полем, содержащим *entry*, указатель на запись для этого идентификатора в таблице символов.
3. *mkleaf(num, val)* создает узел числа с меткой *num* и полем, содержащим *val* — значение числа.

### Пример 5.7

Следующая последовательность вызовов функций создает синтаксическое дерево для выражения *a - 4 + c* на рис. 5.8. В этой последовательности *p<sub>1</sub>*, *p<sub>2</sub>*, ..., *p<sub>5</sub>* являются указателями на узлы, а *entry<sub>a</sub>* и *entry<sub>c</sub>* — указателями на записи в таблице символов для идентификаторов *a* и *c* соответственно.

- |                                                                        |                                                                        |
|------------------------------------------------------------------------|------------------------------------------------------------------------|
| (1) <i>p<sub>1</sub> := mkleaf(id, entry<sub>a</sub>);</i>             | (4) <i>p<sub>4</sub> := mkleaf(id, entry<sub>c</sub>);</i>             |
| (2) <i>p<sub>2</sub> := mkleaf(num, 4);</i>                            | (5) <i>p<sub>5</sub> := mknode('+', p<sub>3</sub>, p<sub>4</sub>);</i> |
| (3) <i>p<sub>3</sub> := mknode('-', p<sub>1</sub>, p<sub>2</sub>);</i> |                                                                        |

Дерево строится снизу вверх. Вызовы функций *mkleaf(id, entry<sub>a</sub>)* и *mkleaf(num, 4)* создают листья для *a* и *4*; указатели на эти узлы сохраняются как *p<sub>1</sub>* и *p<sub>2</sub>*. После этого вы-

зов  $mknodc(' ', p_1, p_2)$  создает внутренний узел с дочерними листьями для а и 4. Еще после двух шагов  $p_5$  указывает на корень дерева.  $\square$

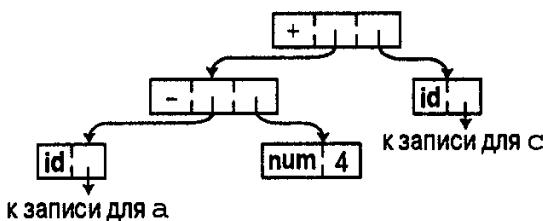


Рис. 5.8. Синтаксическое дерево для  $a - 4 + c$

## Синтаксически управляемое определение для построения синтаксических деревьев

На рис. 5.9 содержится S-атрибутное определение для построения синтаксического дерева выражения, содержащего операции + и -. Это определение использует продукцию грамматики, чтобы задать порядок вызовов функций  $mknodc$  и  $mkleaf$  для построения дерева. Синтезируемый атрибут  $nptr$  у  $E$  и  $T$  используется для хранения указателей, возвращаемых вызовами функций.

| ПРОДУКЦИЯ               | СЕМАНТИЧЕСКИЕ ПРАВИЛА                     |
|-------------------------|-------------------------------------------|
| $E \rightarrow E_1 + T$ | $E.nptr := mknodc('+', E_1.nptr, T.nptr)$ |
| $E \rightarrow E_1 - T$ | $E.nptr := mknodc('-', E_1.nptr, T.nptr)$ |
| $E \rightarrow T$       | $E.nptr := T.nptr$                        |
| $T \rightarrow ( E )$   | $T.nptr := E.nptr$                        |
| $T \rightarrow id$      | $T.nptr := mkleaf(id, id.entry)$          |
| $T \rightarrow num$     | $T.nptr := mkleaf(num, num.val)$          |

Рис. 5.9. Синтаксически управляемое определение для построения синтаксического дерева выражения

### Пример 5.8

Аннотированное дерево разбора, изображающее построение синтаксического дерева для выражения  $a - 4 + c$ , показано на рис. 5.10. Дерево разбора представлено на рисунке пунктирными линиями. Узлы дерева разбора, помеченные нетерминалами  $E$  и  $T$ , используют синтезируемый атрибут  $nptr$  для хранения указателя на узел синтаксического дерева. Этот узел соответствует выражению, представленному нетерминалом.

Семантические правила, связанные с продукциями  $T \rightarrow id$  и  $T \rightarrow num$ , определяют атрибут  $T.nptr$  как указатель на новый лист идентификатора и числа соответственно. Атрибуты  $id.entry$  и  $num.val$  являются лексическими значениями, возвращаемыми лексическим анализатором для токенов  $id$  и  $num$ .

На рис. 5.10, когда выражение  $E$  является одиночным термом (слагаемым), соответствующим использованию продукции  $E \rightarrow T$ , атрибут  $E.nptr$  получает значение  $T.nptr$ . К моменту вызова семантического правила  $E.nptr := mknodc('-', E_1.nptr, T.nptr)$ , связанного с продукцией  $E \rightarrow E_1 - T$ , предыдущие правила уже определили  $E_1.nptr$  и  $T.nptr$  как указатели на листья  $a$  и  $4$  соответственно.

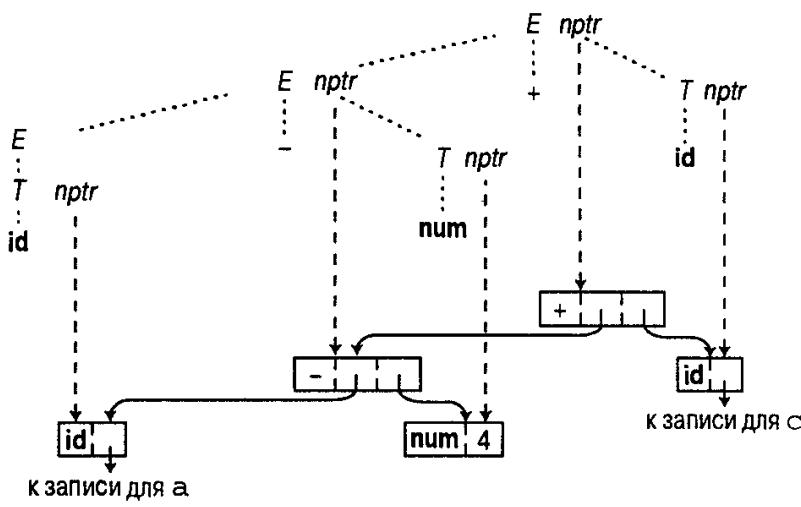


Рис. 5.10. Построение синтаксического дерева для  $a - 4 + c$

При интерпретации рис. 5.10 важно осознавать, что нижнее дерево, построенное из записей, является “реальным” синтаксическим деревом, формирующим выход, в то время как пунктирное дерево над ним — дерево разбора, которое может существовать только в переносном смысле. В следующем разделе мы покажем, как можно легко реализовать S-атрибутное определение с использованием стека восходящего синтаксического анализатора для отслеживания значений атрибутов. При такой реализации функции построения узлов вызываются в том же порядке, что и в примере 5.7. □

## Направленные ациклические графы выражений

Направленный ациклический граф (Directed Acyclic Graph, dag, далее сокращенно называемый дагом) для выражения идентифицирует его общие подвыражения. Подобно синтаксическому дереву, даг имеет узлы для каждого подвыражения своего выражения; внутренний узел представляет операцию, а дочерние узлы — операнды. Отличие заключается в том, что узел дага, представляющий общее подвыражение, имеет более чем одного “родителя”; в синтаксическом дереве общие подвыражения представлены отдельными поддеревьями.

На рис. 5.11 показан даг выражения  $a + a^* (b - c) + (b - c)^* d$ . Лист  $a$  имеет двух родителей, поскольку  $a$  является общим для двух подвыражений:  $a$  и  $a^* (b - c)$ . Аналогично оба появления подвыражения  $b - c$  представлены одним узлом, который также имеет два родительских узла.

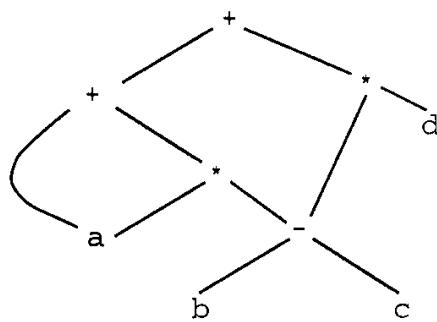


Рис. 5.11. Даг выражения  $a + a^* (b - c) + (b - c)^* d$

Синтаксически управляемое определение на рис. 5.9 вместо создания синтаксического дерева будет строить даг, если мы внесем соответствующие изменения в операции по-

строения узлов. Даг получится, если функция построения узла вначале будет проверять наличие идентичного узла. Например, перед построением нового узла с меткой *op* и полями с указателями *left* и *right*, вызов *mnode(op, left, right)* может проверить, не был ли уже построен такой узел, и если это так, то вызов может возвратить указатель на ранее построенный узел. Измененная функция построения листьев *mkleaf* ведет себя аналогично.

### Пример 5.9

Последовательность инструкций на рис. 5.12 строит даг, приведенный на рис. 5.11 (функции *mnode* и *mkleaf* создают новые узлы только при необходимости, возвращая указатели на уже существующие узлы с корректными метками и дочерними узлами, когда это возможно). На рис. 5.12 *a*, *b*, *c* и *d* указывают на записи в таблице символов для идентификаторов *a*, *b*, *c* и *d*.

- |                                             |                                                       |
|---------------------------------------------|-------------------------------------------------------|
| (1) $p_1 := \text{mkleaf}(\text{id}, a);$   | (8) $p_8 := \text{mkleaf}(\text{id}, b);$             |
| (2) $p_2 := \text{mkleaf}(\text{id}, a);$   | (9) $p_9 := \text{mkleaf}(\text{id}, c);$             |
| (3) $p_3 := \text{mkleaf}(\text{id}, b);$   | (10) $p_{10} := \text{mnode}(' - ', p_8, p_9);$       |
| (4) $p_4 := \text{mkleaf}(\text{id}, c);$   | (11) $p_{11} := \text{mkleaf}(\text{id}, d);$         |
| (5) $p_5 := \text{mnode}(' - ', p_3, p_4);$ | (12) $p_{12} := \text{mnode}(' * ', p_{10}, p_{11});$ |
| (6) $p_6 := \text{mnode}(' * ', p_2, p_5);$ | (13) $p_{13} := \text{mnode}(' + ', p_7, p_{12});$    |
| (7) $p_7 := \text{mnode}(' + ', p_1, p_6);$ |                                                       |

Рис. 5.12. Инструкции построения дага на рис. 5.11

При повторном вызове *mkleaf(id, a)* в строке (2) возвращается узел, построенный предыдущим вызовом *mkleaf(id, a)*, т.е.  $p_1 = p_2$ . Аналогично, узлы, возвращаемые в строках 8 и 9, соответственно те же, что и в строках 3 и 4. Следовательно, узел, возвращаемый в строке 10, должен быть тем же, что и построенный вызовом *mnode* в пятой строке.  $\square$

Во многих приложениях узлы реализованы как записи, хранящиеся в массиве, как показано на рис. 5.13. На этом рисунке каждая запись имеет поле метки, которое определяет природу узла. Обращаться к узлу можно по его индексу или позиции в массиве. Целый индекс узла зачастую называется *номером значения*. Например, используя номера значений, мы можем сказать, что узел 3 имеет метку +, его левым потомком является узел 1, а правым — узел 2. Для создания узлов дага, представляющего выражение, может использоваться следующий алгоритм.

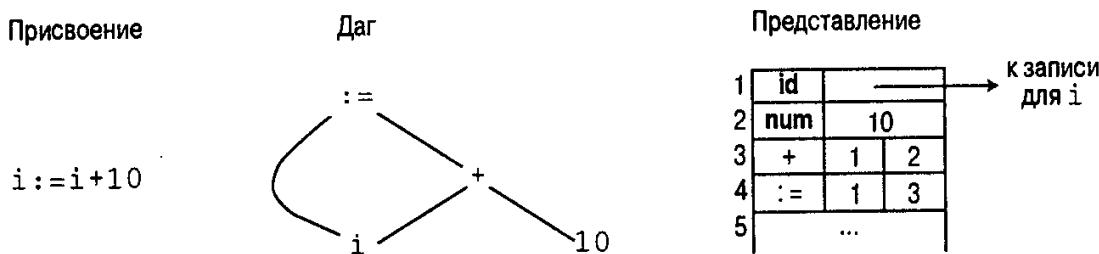


Рис. 5.13. Узлы дага для  $i := i + 10$

### Алгоритм 5.1. Метод номера значения построения узла дага

Предположим, что узлы хранятся в массиве, как показано на рис. 5.13, и каждый узел определяется своим номером значения. Назовем *сигнатурой* узла операции тройку  $\langle op, l, r \rangle$ , состоящую из его метки *op*, а также левого и правого потомков *l* и *r*.

*Вход.* Метка  $op$ , узлы  $l$  и  $r$ .

*Выход.* Узел с сигнатурой  $\langle op, l, r \rangle$ .

*Метод.* Ищем в массиве узел  $m$  с меткой  $op$ , левым потомком  $l$  и правым потомком  $r$ . Если такой узел имеется, возвращаем  $m$ ; в противном случае создаем новый узел  $n$  с меткой  $op$ , левым потомком  $l$  и правым потомком  $r$  и возвращаем  $n$ .

Очевидный путь определения, имеется ли в массиве искомый узел  $m$ , состоит в хранении списка всех ранее созданных узлов и проверке, не имеет ли какой-либо из них интересующую нас сигнатуру. Поиск узла  $m$  может быть реализован более эффективно, если использовать  $k$  списков, называемых блоками, и хеш-функцию  $h$  для определения, в каком блоке следует искать узел<sup>1</sup>.

Хеш-функция  $h$  вычисляет номер блока по значениям  $op$ ,  $l$  и  $r$ . Она всегда возвращает один и тот же номер блока при одних и тех же аргументах. Если  $m$  отсутствует в блоке  $h(op, l, r)$ , то создается новый узел  $n$  и добавляется в этот блок, чтобы последующий поиск мог обнаружить его там. Несколько сигнатур могут хешироваться в один и тот же блок, но на практике блоки содержат небольшое число узлов<sup>2</sup>.

Каждый блок может быть реализован в виде связанного списка, показанного на рис. 5.14. Каждая ячейка связанного списка представляет узел. Заголовки блоков хранятся в массиве; при этом каждый заголовок является указателем на первую ячейку списка. Номер блока, возвращаемый хеш-функцией  $h(op, l, r)$ , представляет собой индекс в этом массиве.

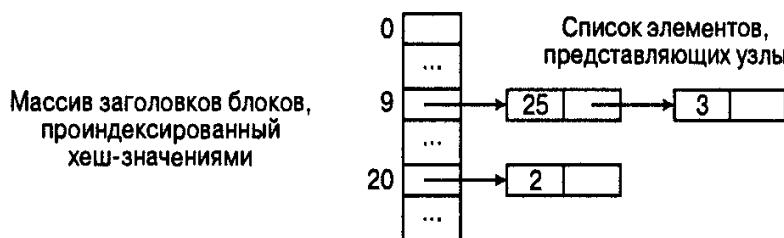


Рис. 5.14. Структура данных для блоков поиска

Этот алгоритм может быть адаптирован для применения к узлам, которые могут не размещаться последовательно в одном массиве. Во многих компиляторах память для узлов выделяется по мере необходимости, чтобы избежать ее перерасхода. В этом случае мы не можем считать, что узлы расположены в памяти последовательно, и должны использовать для обращения к ним указатели. Если можно создать хеш-функцию для вычисления номера блока по метке и указателям на дочерние узлы, то вместо номеров значений можем применять указатели. В противном случае можно каким-либо образом перенумеровать узлы и использовать эти номера в качестве номеров значений узлов. □

Даги могут использоваться и для представления множеств выражений, поскольку даг может иметь несколько корней. В главах 9, “Генерация кода”, и 10, “Оптимизация кода”, вычисления, выполняемые последовательностью операторов присвоений, будут представлены в виде дага.

<sup>1</sup> Здесь подходит любая структура данных, реализующая словари в смысле работы [8]. Важное свойство такой структуры заключается в наличии ключа, т.е. метки  $op$  и двух узлов  $l$  и  $r$ , по которому мы можем быстро обнаружить узел  $m$  с сигнатурой  $\langle op, l, r \rangle$  или выяснить, что его не существует.

<sup>2</sup> Более детально с технологией хеширования можно познакомиться в разделе 6.4 книги Киут Д. Искусство программирования. Т. 3. Сортировка и поиск. — М.: Издательский дом “Вильямс”, 2000. — Прим. ред.

## 5.3. Восходящее выполнение S-атрибутных определений

Теперь, когда мы познакомились, как для определения трансляций можно использовать синтаксически управляемые определения, приступим к изучению реализации таких трансляторов. Создание транслятора для произвольного синтаксически управляемого определения может оказаться сложной задачей; однако имеются большие классы полезных синтаксически управляемых определений, трансляторы для которых строятся достаточно просто. В этом разделе мы рассмотрим один такой класс, а именно — S-атрибутные определения, т.е. синтаксически управляемые определения, в которых применяются исключительно синтезируемые атрибуты. В последующих разделах будет рассмотрена реализация определений, в которых используются и наследуемые атрибуты.

Синтезируемые атрибуты могут быть вычислены восходящим синтаксическим анализатором в процессе разбора входной строки. Синтаксический анализатор может хранить значения синтезируемых атрибутов, связанных с грамматическими символами, в своем стеке. При выполнении свертки по хранящимся в стеке атрибутам символов из правой части сворачиваемой продукции вычисляются значения новых синтезируемых атрибутов. В этом разделе мы рассмотрим, каким образом можно расширить стек синтаксического анализатора для хранения значений этих синтезируемых атрибутов. В разделе 5.6 мы увидим, что эта реализация поддерживает и некоторые наследуемые атрибуты.

В синтаксически управляемом определении на рис. 5.9 для построения синтаксического дерева выражения используются только синтезируемые атрибуты. Следовательно, технология, рассматриваемая в этом разделе, может быть использована для построения синтаксических деревьев в процессе восходящего синтаксического анализа. Как мы увидим в разделе 5.5, при трансляции выражений в процессе нисходящего синтаксического анализа часто используются наследуемые атрибуты. Поэтому мы отложим рассмотрение такой трансляции, пока не изучим зависимости “слева направо” в следующем разделе.

### Синтезируемые атрибуты в стеке синтаксического анализатора

Транслятор для S-атрибутного определения зачастую может быть реализован с помощью генератора LR-анализаторов типа Yacc, рассмотренного в разделе 4.9. По S-атрибутному определению генератор синтаксических анализаторов может создать транслятор, вычисляющий атрибуты в процессе разбора входной строки.

Восходящий синтаксический анализатор для хранения информации о разобраных поддеревьях использует стек. Мы можем использовать дополнительные поля в стеке синтаксического анализатора для хранения значений синтезируемых атрибутов. На рис. 5.15 показан пример стека синтаксического анализатора с пространством для хранения одного значения атрибута. Предположим, что, как на рисунке, стек реализован с помощью пары массивов — *state* и *val*. Каждая запись *state* является указателем (или индексом) на запись в таблице LR(1)-анализа (обратите внимание, что грамматический символ неявно входит в состояние и нет необходимости в его явном хранении в стеке). Тем не менее, удобно ссылаться на состояния с помощью грамматических символов, которым они соответствуют при помещении в стек, как описано в разделе 4.7. Если символ *i*-го состояния — *A*, то *val[i]* содержит значение атрибута, связанного с узлом дерева разбора, соответствующим этому *A*.

| state | val |
|-------|-----|
| ...   | ... |
| X     | X.x |
| Y     | Y.y |
| Z     | Z.z |
| ...   | ... |

Рис. 5.15. Стек синтаксического анализатора с полем для синтезируемого атрибута

Текущая вершина стека определяется указателем  $top$ . Мы предполагаем, что синтезируемые атрибуты вычисляются непосредственно перед каждой сверткой. Предположим, что с продукцией  $A \rightarrow XYZ$  связано семантическое правило  $A.a := f(X.x, Y.y, Z.z)$ . Перед сверткой  $XYZ$  в  $A$  значение атрибута  $Z.z$  находится в  $val[top]$ , значение  $Y.y$  — в  $val[top-1]$  и  $X.x$  — в  $val[top-2]$ . Если символ не имеет атрибутов, то соответствующая запись в массиве  $val$  не определена. После свертки  $top$  уменьшается на 2, состояние, соответствующее  $A$ , помещается в  $state[top]$  (т.е. на место  $X.x$ ), а значение синтезируемого атрибута  $A.a$  — в  $val[top]$ .

### Пример 5.10

Вновь рассмотрим синтаксически управляемое определение калькулятора, приведенное на рис. 5.2. Синтезируемые атрибуты в аннотированном дереве разбора на рис. 5.3 могут быть вычислены LR-анализатором в процессе восходящего разбора входной строки  $3 * 5 + 4 \text{ n}$ . Как и ранее, мы полагаем, что лексический анализатор обеспечивает значение атрибута  $\text{digit}.lexval$  — числовое значение каждого токена, представляющего цифру. При переносе синтаксическим анализатором токена  $\text{digit}$  в стек, он помещается в  $state[top]$ , а значение атрибута — в  $val[top]$ .

Мы можем использовать технологии, описанные в разделе 4.7, для построения LR-анализатора грамматики, лежащей в основе калькулятора. Для вычисления атрибутов модифицируем синтаксический анализатор, чтобы он выполнял фрагменты кода, показанные на рис. 5.16, непосредственно перед выполнением соответствующей свертки. Заметим, что мы можем связать вычисление атрибутов со свертками, поскольку каждая свертка определяет используемую продукцию. Фрагменты кода получены из семантических правил на рис. 5.2 путем замены каждого атрибута позицией в массиве  $val$ .

| ПРОДУКЦИЯ                    | ФРАГМЕНТ КОДА                               |
|------------------------------|---------------------------------------------|
| $L \rightarrow E \text{ n}$  | <code>print(val[top])</code>                |
| $E \rightarrow E_1 + T$      | <code>val[ntop]:=val[top-2]+val[top]</code> |
| $E \rightarrow T$            |                                             |
| $T \rightarrow T_1 * F$      | <code>val[ntop]:=val[top-2]*val[top]</code> |
| $T \rightarrow F$            |                                             |
| $F \rightarrow ( E )$        | <code>val[ntop]:=val[top-1]</code>          |
| $F \rightarrow \text{digit}$ |                                             |

Рис. 5.16. Реализация калькулятора с помощью LR-анализатора

Фрагменты кода не показывают, каким образом происходит работа с  $ntop$  и  $top$ . При свертке продукции с  $r$  символами в правой части значение  $ntop$  устанавливается равным  $top-r+1$ , а после выполнения фрагмента кода  $top$  становится равным  $ntop$ .

На рис. 5.17 показана последовательность действий, выполняемых синтаксическим анализатором для входной строки  $3 * 5 + 4 \text{ n}$ . В таблице представлено содержимое полей *state* и *val* стека после каждого действия синтаксического анализатора. Здесь мы расширим оговоренную ранее возможность заменять состояния стека соответствующими грамматическими символами, и вместо токена **digit** будем использовать введенную цифру.

| ВХОД                  | <i>state</i> | <i>val</i> | ИСПОЛЬЗУЕМАЯ ПРОДУКЦИЯ       |
|-----------------------|--------------|------------|------------------------------|
| $3 * 5 + 4 \text{ n}$ | —            | —          |                              |
| $* 5 + 4 \text{ n}$   | 3            | 3          |                              |
| $* 5 + 4 \text{ n}$   | F            | 3          |                              |
| $* 5 + 4 \text{ n}$   | T            | 3          |                              |
| $* 5 + 4 \text{ n}$   | $T *$        | 3          |                              |
| $+ 4 \text{ n}$       | $T * 5$      | 3 — 5      |                              |
| $+ 4 \text{ n}$       | $T * F$      | 3 — 5      | $F \rightarrow \text{digit}$ |
| $+ 4 \text{ n}$       | T            | 15         | $T \rightarrow T * F$        |
| $+ 4 \text{ n}$       | E            | 15         | $E \rightarrow T$            |
| $+ 4 \text{ n}$       | $E +$        | 15 —       |                              |
| $\text{n}$            | $E + 4$      | 15 — 4     |                              |
| $\text{n}$            | $E + F$      | 15 — 4     | $F \rightarrow \text{digit}$ |
| $\text{n}$            | $E + T$      | 15 — 4     | $T \rightarrow F$            |
| $\text{n}$            | E            | 19         | $E \rightarrow E + T$        |
| $E \text{ n}$         | 19           | —          |                              |
| L                     | 19           |            | $L \rightarrow E \text{ n}$  |

Рис. 5.17. Действия транслятора для входной строки  $3 * 5 + 4 \text{ n}$

Рассмотрим последовательность событий для входного символа 3. На первом шаге синтаксический анализатор переносит в стек состояние, соответствующее токену **digit** (значение его атрибута равно 3). Состояние представлено тройкой; в поле *val* находится значение 3. На следующем шаге синтаксический анализатор выполняет свертку в соответствии с продукцией  $F \rightarrow \text{digit}$  и семантическое правило  $F.\text{val} := \text{digit}.lexval$ . Третий шаг состоит в свертке по продукции  $T \rightarrow F$ . С этой продукцией не связан никакой фрагмент кода, поэтому массив *val* остается неизменным. Заметим, что после каждой свертки вершина стека *val* содержит значение атрибута, связанное с левой частью сворачивающей продукции.  $\square$

При реализации, схематически представленной выше, фрагменты кода выполняются непосредственно перед сверткой. Свертки предоставляют “крючок”, на который можно повесить действия, состоящие из произвольных фрагментов кода<sup>3</sup>. Таким образом, мы можем разрешить пользователю связать с продукцией действие, выполняемое в процессе свертки в соответствии с данной продукцией. Схема трансляции, рассматриваемая в следующем разделе, предоставляет способ записи действий, выполняемых в процессе синтаксического анализа. В разделе 5.6 мы увидим, каким образом в процессе восходящего разбора может быть реализован больший класс синтаксически управляемых определений.

<sup>3</sup> Обычно “hook” в применении к программированию переводится как “перехват”, но в данном случае игра слов оказалась вполне переводимой и не лишенной смысла. — Прим. перев.

## 5.4. L-атрибутные определения

Когда трансляция выполняется в процессе синтаксического анализа, порядок вычисления атрибутов привязан к порядку, в котором узлы дерева разбора “создаются” методом анализа. Естественный порядок, который характеризует многие нисходящие и восходящие методы трансляции, получается при применении процедуры *dfvisit*, приведенной на рис. 5.18, к корню дерева разбора. Этот порядок вычисления мы называем “в глубину” (depth-first order). Даже если в действительности дерево разбора не строится, полезно изучить трансляцию в процессе синтаксического анализа, рассматривая именно этот метод вычисления атрибутов в узлах дерева разбора.

```
procedure dfvisit(n:node);
begin
 for каждый потомок m узла n слева направо do begin
 вычислить наследуемые атрибуты m;
 dfvisit(m)
 end;
 вычислить синтезируемые атрибуты n
end
```

Рис. 5.18. Порядок вычисления атрибутов в дереве разбора в глубину

Теперь мы введем новый класс синтаксически управляемых определений, называемых L-атрибутными определениями, атрибуты которых всегда могут быть вычислены в порядке в глубину (здесь L означает left (левый), поскольку информация об атрибутах появляется слева направо). Реализация всех больших классов L-атрибутных определений рассматривается в трех последующих разделах данной главы. Эти определения включают все синтаксически управляемые определения, основанные на LL(1)-грамматиках. В разделе 5.5 дан метод реализации таких определений за один проход с использованием методов предиктивного синтаксического анализа. Большой класс L-атрибутных определений реализуется в разделе 5.6 в процессе восходящего синтаксического анализа путем расширения методов трансляции из раздела 5.3. Набросок общего метода реализации всех L-атрибутных определений приведен в разделе 5.7.

### L-атрибутные определения

Синтаксически управляемое определение является *L-атрибутным*, если каждый наследуемый атрибут символа  $X_j$ ,  $1 \leq j \leq n$ , из правой части продукции  $A \rightarrow X_1X_2\dots X_n$  зависит только от

1. атрибутов символов  $X_1, X_2, \dots, X_{j-1}$ , расположенных в продукции слева от  $X_j$ ;
2. наследуемых атрибутов  $A$ .

Заметьте, что каждое S-атрибутное определение является L-атрибутным, так как ограничения (1) и (2) относятся только к наследуемым атрибутам.

#### Пример 5.11

Синтаксически управляемое определение на рис. 5.19 не является L-атрибутным, поскольку наследуемый атрибут  $Q.i$  грамматического символа  $Q$  зависит от атрибута  $R.s$  грамматического символа справа. Другие примеры определений, не являющихся L-атрибутными, можно найти в разделах 5.8 и 5.9.  $\square$

| ПРОДУКЦИЯ           | СЕМАНТИЧЕСКИЕ ПРАВИЛА                                 |
|---------------------|-------------------------------------------------------|
| $A \rightarrow L M$ | $L.i := l(A.i)$<br>$M.i := m(L.s)$<br>$A.s := f(M.s)$ |
| $A \rightarrow Q R$ | $R.i := r(A.i)$<br>$Q.i := q(R.s)$<br>$A.s := f(Q.s)$ |

Рис. 5.19. Синтаксически управляемое определение, не являющееся L-атрибутным

## Схемы трансляции

Схема трансляции представляет собой контекстно-свободную грамматику, в которой атрибуты, связанные с символами грамматики, и семантические действия заключены в фигурные скобки ({} ) и вставлены в правые части продукции (как было сделано в разделе 2.3). В этой главе мы воспользуемся схемами трансляции как удобным способом записи определения трансляции, выполняемой в процессе синтаксического анализа.

Схемы трансляции, рассматриваемые в этой главе, имеют как наследуемые, так и синтезируемые атрибуты. В простых схемах трансляции, рассмотренных в главе 2, “Простой однопроходный компилятор”, атрибуты имели строковый тип, у каждого символа грамматики был ровно один атрибут и для каждой продукции  $A \rightarrow X_1X_2\dots X_n$  семантическое правило формировало строку атрибута для  $A$  конкатенацией строк для  $X_1, X_2, \dots, X_n$  в приведенном порядке, с некоторыми дополнительными строками между ними. Мы видели, что трансляция могла быть выполнена просто выводом строк в порядке их появления в семантических правилах.

### Пример 5.12

Перед вами простая схема трансляции, преобразующая инфиксные выражения со сложением и вычитанием в соответствующие постфиксные выражения. Она представляет собой немного переработанную схему трансляции (2.14).

$$\begin{aligned} E &\rightarrow TR \\ R &\rightarrow \text{addop } T \{ \text{print}(\text{addop.lexeme}) \} R_1 \mid \epsilon \\ T &\rightarrow \text{num} \{ \text{print}(\text{num.val}) \} \end{aligned} \tag{5.1}$$

На рис. 5.20 показано дерево разбора для входной строки  $9 - 5 + 2$ , на котором семантические действия показаны как дочерние узлы по отношению к узлам, представляющим левые части соответствующих продукции. По сути, мы рассматриваем действия как терминальные символы, что удобно для определения момента выполнения этих действий. Вместо токенов num и addop мы указываем реальные числа и операцию сложения. При выполнении действий в порядке обхода в глубину действия на рис. 5.20 приводят к выводу  $95 - 2 +$ . □

При разработке схемы трансляции мы должны соблюдать некоторые ограничения для того, чтобы гарантировать, что значение атрибута доступно при обращении к нему. Эти ограничения, накладываемые L-атрибутными определениями, гарантируют, что действие не использует атрибут, который еще не вычислен.

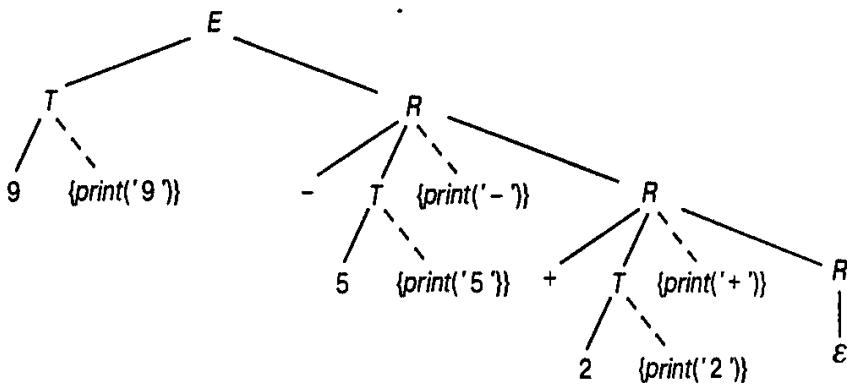


Рис. 5.20. Дерево разбора с семантическими действиями для выражения  $9 - 5 + 2$

Простейший случай — когда используются только синтезируемые атрибуты. В этом случае можно построить схему трансляции просто путем создания действий для каждого семантического правила, состоящих из присвоения, и размещения этих действий в конце правой части связанных продукции. Например, продукция и семантическое правило

| ПРОДУКЦИЯ               | СЕМАНТИЧЕСКОЕ ПРАВИЛО           |
|-------------------------|---------------------------------|
| $T \rightarrow T_1 * F$ | $T.val := T_1.val \times F.val$ |

дают следующую продукцию и семантическое действие.

$$T \rightarrow T_1 * F \{ T.val := T_1.val \times F.val \}$$

Если у нас имеются и синтезируемые, и наследуемые атрибуты, мы должны быть более осторожны.

1. Наследуемый атрибут для символа из правой части продукции должен вычисляться в действии перед этим символом.
2. Действие не должно обращаться к синтезируемому атрибуту символа справа от действия.
3. Синтезируемый атрибут для нетерминала в левой части продукции может вычисляться только после того, как будут вычислены все атрибуты, от которых он зависит. Действие, вычисляющее такой атрибут, обычно может размещаться в конце правой части продукции.

В двух следующих разделах мы покажем, как путем обобщения исходящих и восходящих синтаксических анализаторов можно реализовать схему трансляции, удовлетворяющую этим трем требованиям.

Следующая схема трансляции не удовлетворяет первому из трех требований.

$$\begin{array}{lcl} S & \rightarrow & A_1 A_2 \quad \{ A_1.in := 1; A_2.in := 2 \} \\ A & \rightarrow & a \quad \{ print(A.in) \} \end{array}$$

Наследуемый атрибут  $A.in$  во второй продукции не определен, когда мы пытаемся вывести его значение в процессе обхода в глубину дерева разбора входной строки  $aa$ . Обход в глубину начинается в  $S$  и проходит поддеревья  $A_1$  и  $A_2$  до того, как устанавливаются значения  $A_1.in$  и  $A_2.in$ . Если действие, определяющее значения  $A_1.in$  и  $A_2.in$ , вставить перед символами  $A$  в правой части продукции  $S \rightarrow A_1 A_2$ , то  $A.in$  будет определено при каждом вызове  $print(A.in)$ .

Начав с L-атрибутного синтаксически управляемого определения, всегда можно построить схему трансляции, удовлетворяющую трем приведенным выше требованиям. Следующий пример иллюстрирует такое построение. Пример основан на языке форми-

рования математических формул EQN, вкратце описанном в разделе 1.2. При получении входной строки

`E sub 1 .val`

EQN размещает  $E$ ,  $1$  и  $.val$  так, как показано на рис. 5.21. Обратите внимание, что нижний индекс  $1$  сдвинут вниз и имеет меньший размер по отношению к остальным символам.

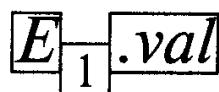


Рис. 5.21. Синтаксически управляемое размещение блоков

### Пример 5.13

По L-атрибутному определению на рис. 5.22 мы строим схему трансляции, приведенную на рис. 5.23. На этих рисунках нетерминал  $B$  (блок) представляет формулу. Продукция  $B \rightarrow B B$  описывает размещение двух блоков “бок о бок”, а  $B \rightarrow B \text{ sub } B$  означает, что второй блок (нижний индекс) имеет меньший размер и размещается ниже основной строки.

| ПРОДУКЦИЯ                            | СЕМАНТИЧЕСКИЕ ПРАВИЛА                                                                        |
|--------------------------------------|----------------------------------------------------------------------------------------------|
| $S \rightarrow B$                    | $B.ps := 10$<br>$S.ht := B.ht$                                                               |
| $B \rightarrow B_1 B_2$              | $B_1.ps := B.ps$<br>$B_2.ps := B.ps$<br>$B.ht := \max(B_1.ht, B_2.ht)$                       |
| $B \rightarrow B_1 \text{ sub } B_2$ | $B_1.ps := B.ps$<br>$B_2.ps := \text{shrink}(B.ps)$<br>$B.ht := \text{disp}(B_1.ht, B_2.ht)$ |
| $B \rightarrow \text{text}$          | $B.ht := \text{text}.h \times B.ps$                                                          |

Рис. 5.22. Синтаксически управляемое определение размера и высоты шрифта блоков

|     |               |                                                       |
|-----|---------------|-------------------------------------------------------|
| $S$ | $\rightarrow$ | $\{ B.ps := 10 \}$                                    |
|     |               | $B \quad \{ S.ht := B.ht \}$                          |
| $B$ | $\rightarrow$ | $\{ B_1.ps := B.ps \}$                                |
|     |               | $B_1 \quad \{ B_2.ps := B.ps \}$                      |
|     |               | $B_2 \quad \{ B.ht := \max(B_1.ht, B_2.ht) \}$        |
| $B$ | $\rightarrow$ | $\{ B_1.ps := B.ps \}$                                |
|     |               | $B_1 \quad \{ B_2.ps := \text{shrink}(B.ps) \}$       |
|     |               | $B_2 \quad \{ B.ht := \text{disp}(B_1.ht, B_2.ht) \}$ |
| $B$ | $\rightarrow$ | $\{ B.ht := \text{text}.h \times B.ps \}$             |

Рис. 5.23. Схема трансляции, построенная на основе рис. 5.22

Наследуемый атрибут  $ps$  (point size — размер шрифта) определяет высоту используемого шрифта и влияет на размер формулы. Правило для продукции  $B \rightarrow \text{text}$  умножает нормализованную высоту текста на высоту шрифта для определения реальной высоты текста. Атрибут  $h$  определяется из таблицы по символам, входящим в состав токена `text`.

При применении продукции  $B \rightarrow B_1 B_2$ , согласно правилам копирования,  $B_1$  и  $B_2$  наследуют размер шрифта  $B$ . Высота  $B$ , представленная синтезируемым атрибутом  $ht$ , является максимальной из высот  $B_1$  и  $B_2$ .

При использовании продукции  $B \rightarrow B_1 \text{ sub } B_2$  функция *shrink* уменьшает размер шрифта  $B_2$  на 30%. Функция *disp* позволяет переместить блок  $B_2$  вниз и вычисляет высоту  $B$ . Правила, которые порождают реальные команды вывода формулы, здесь не приведены.

Определение на рис. 5.22 — L-атрибутное. У нетерминала  $B$  имеется только один наследуемый атрибут —  $ps$ . Каждое семантическое правило определяет  $ps$  только через наследуемый атрибут нетерминала из левой части продукции, так что рассматриваемое определение — L-атрибутное.

Схема трансляции на рис. 5.23 получена посредством вставки в продукцию присвоений, соответствующих семантическим правилам на рис. 5.22, в соответствии с тремя приведенными ранее требованиями. Для удобочитаемости каждый грамматический символ продукции записывается в отдельной строке, а действия приведены немного правее. Так,

$$S \rightarrow \{ B.ps := 10 \} B \{ S.ht := B.ht \}$$

записывается как

$$\begin{array}{ll} S & \rightarrow \{ B.ps := 10 \} \\ & B \quad \{ S.ht := B.ht \} \end{array}$$

Обратите внимание, что действия, устанавливающие наследуемые атрибуты  $B_1.ps$  и  $B_2.ps$ , вставлены непосредственно перед  $B_1$  и  $B_2$  в правых частях продукции.  $\square$

## 5.5. Нисходящая трансляция

В этом разделе L-атрибутные определения будут реализованы в процессе предиктивного синтаксического анализа. Мы работаем не с синтаксически управляемыми определениями, а схемами трансляции, так что можно явным образом определить порядок, в котором выполняются действия и вычисляются атрибуты. Мы также расширим алгоритм устранения левых рекурсий для работы со схемами трансляции с синтезируемыми атрибутами.

### Устранение левой рекурсии из схемы трансляции

Поскольку большинство арифметических операторов левоассоциативны, естественно использовать для выражений леворекурсивные грамматики. Сейчас мы расширим алгоритм устранения левой рекурсии из разделов 2.4 и 4.3, обеспечивая возможность работы алгоритма с грамматикой, использующей атрибуты. Преобразования применяются к схемам трансляции с синтезируемыми атрибутами и обеспечивают реализацию многих синтаксически управляемых определений из разделов 5.1 и 5.2 с использованием предиктивного синтаксического анализатора. Следующий пример поясняет преобразование.

#### Пример 5.14

Схема трансляции, представленная на рис. 5.24, преобразуется в схему (рис. 5.25), которая порождает для выражения  $9 - 5 + 2$  аннотированное дерево разбора, показанное на рис. 5.26. Стрелки на рисунке указывают путь определения значения выражения.

|                            |                                |
|----------------------------|--------------------------------|
| $E \rightarrow E_1 + T$    | { $E.val := E_1.val + T.val$ } |
| $E \rightarrow E_1 - T$    | { $E.val := E_1.val - T.val$ } |
| $E \rightarrow T$          | { $E.val := T.val$ }           |
| $T \rightarrow ( E )$      | { $T.val := E.val$ }           |
| $T \rightarrow \text{num}$ | { $T.val := \text{num}.val$ }  |

Рис. 5.24. Схема трансляции с леворекурсивной грамматикой

|                            |                               |
|----------------------------|-------------------------------|
| $E \rightarrow T$          | { $R.i := T.val$ }            |
| $R$                        | { $E.val := R.s$ }            |
| $R \rightarrow +$          |                               |
| $T$                        | { $R_1.i := R.i + T.val$ }    |
| $R_1$                      | { $R.s := R_1.s$ }            |
| $R \rightarrow -$          |                               |
| $T$                        | { $R_1.i := R.i - T.val$ }    |
| $R_1$                      | { $R.s := R_1.s$ }            |
| $R \rightarrow \epsilon$   | { $R.s := R.i$ }              |
| $T \rightarrow ($          |                               |
| $E$                        |                               |
| $)$                        | { $T.val := E.val$ }          |
| $T \rightarrow \text{num}$ | { $T.val := \text{num}.val$ } |

Рис. 5.25. Преобразованная схема трансляции с праворекурсивной грамматикой

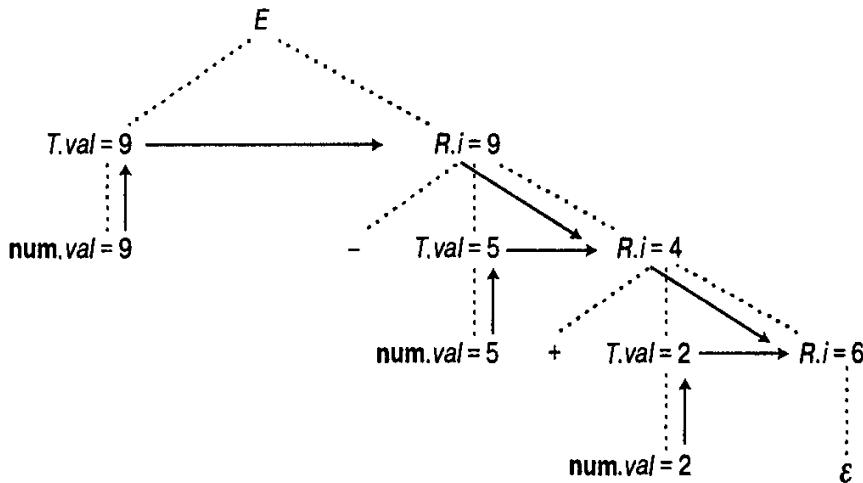


Рис. 5.26. Вычисление выражения  $9 - 5 + 2$

На рис. 5.26 отдельные числа порождаются  $T$ , и  $T.val$  получает их значения от лексических значений чисел, представляемых атрибутом  $\text{num}.val$ . Значение 9 в подвыражении  $9 - 5$  порождается крайним слева  $T$ , но оператор  $-$  и значение 5 порождаются  $R$  в правом потомке корня. Наследуемый атрибут  $R.i$  получает значение 9 от  $T.val$ . Вычитание  $9 - 5$  и передача результата 4 вниз к среднему узлу для  $R$  выполняется вставкой следующего действия между  $T$  и  $R_1$  в продукции  $R \rightarrow - T R_1$ : {  $R_1.i := R.i - T.val$  }.

Аналогичное действие добавляет 2 к значению  $9 - 5$ ; в результате получаем  $R.i = 6$  в нижнем узле для  $R$ . Окончательный результат должен быть получен в корневом узле, как значение  $E.val$ . Для этого используется синтезируемый атрибут  $R.s$ , передающий результат в корневой узел (не показанный на рис. 5.26). □

Для нисходящего разбора мы можем считать, что действие выполняется в тот момент, когда развертывается символ из этой же позиции. Таким образом, во второй продукции на рис. 5.25 первое действие (присвоение  $R_1.i$ ) выполняется после того, как  $T$  полностью развернут в последовательность терминалов, а второе действие — после полного развертывания  $R_1$ . Как указывалось при рассмотрении L-атрибутных определений в разделе 5.4, наследуемый атрибут символа должен вычисляться действием, находящимся перед символом, а синтезируемый атрибут нетерминала в левой части — после вычисления всех атрибутов, от которых он зависит.

Чтобы адаптировать другие леворекурсивные схемы трансляции для предиктивного синтаксического анализа, выразим использование атрибутов  $R.i$  и  $R.s$  на рис. 5.25 более абстрактно. Предположим, у нас имеется следующая схема трансляции

$$\begin{aligned} A \rightarrow A_1 Y & \quad \{ A.a := g(A_1.a, Y.y) \} \\ A \rightarrow X & \quad \{ A.a := f(X.x) \} \end{aligned} \quad (5.2)$$

Каждый грамматический символ имеет синтезируемые атрибуты, записанные с использованием соответствующих строчных букв,  $a$ ,  $f$  и  $g$  — произвольные функции. Обобщение на случай дополнительных  $A$ -продукций и продукции со строками на месте символов  $X$  и  $Y$  может быть выполнено так, как в приведенном ниже примере 5.15.

Алгоритм устранения левой рекурсии в разделе 2.4 строит по (5.2) следующую грамматику

$$\begin{aligned} A \rightarrow X R \\ R \rightarrow Y R \mid \epsilon \end{aligned} \quad (5.3)$$

С учетом семантических действий запишем преобразованную схему в виде

$$\begin{aligned} A \rightarrow X & \quad \{ R.i := f(X.x) \} \\ & \quad R \\ R \rightarrow Y & \quad \{ R.i := g(R.i, Y.y) \} \\ & \quad R_1 \\ R \rightarrow \epsilon & \quad \{ R.s := R.i \} \end{aligned} \quad (5.4)$$

Эта схема использует атрибуты  $R.i$  и  $R.s$ , как и на рис. 5.25. Для того чтобы увидеть, что результаты (5.2) и (5.4) одни и те же, рассмотрим два аннотированных дерева разбора на рис. 5.27. Значение  $A.a$  на рис. 5.27а вычисляется в соответствии с (5.2). На рис. 5.27б приведено вычисление  $R.i$  вниз по дереву в соответствии с (5.4). Значение  $R.i$  внизу передается затем наверх неизмененным в виде  $R.s$  и становится корректным значением  $A.a$  в корне дерева (атрибут  $R.s$  на рисунке не показан).

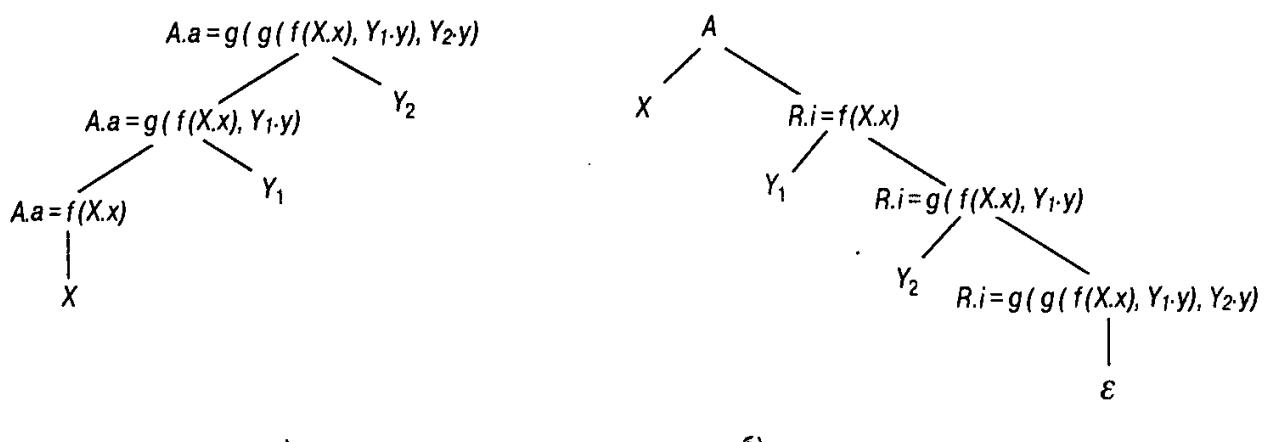


Рис. 5.27. Два способа вычисления значения атрибута

### Пример 5.15

Если синтаксически управляемое определение для построения синтаксических деревьев (рис. 5.9) преобразовать в схему трансляции, то продукция и семантические действия для  $E$  примут следующий вид

$$\begin{aligned} E \rightarrow E_1 + T & \quad \{ E.nptr := \text{mknode}('+', E_1.nptr, T.nptr) \} \\ E \rightarrow E_1 - T & \quad \{ E.nptr := \text{mknode}('-', E_1.nptr, T.nptr) \} \\ E \rightarrow T & \quad \{ E.nptr := T.nptr \} \end{aligned}$$

При устранении левой рекурсии из этой схемы трансляции нетерминал  $E$  соответствует  $A$  из (5.2), а строки  $+T$  и  $-T$  в первых двух продукциях —  $Y$ ; нетерминал  $T$  в третьей продукции соответствует  $X$ . Преобразованная схема трансляции показана на рис. 5.28. Продукции и семантические действия для  $T$  аналогичны таковым в исходном определении на рис. 5.9.

$$\begin{aligned} E \rightarrow T & \quad \{ R.i := T.nptr \} \\ R & \quad \{ E.nptr := R.s \} \\ R \rightarrow + & \\ & \quad T \quad \{ R_1.i := \text{mknode}('+', R.i, T.nptr) \} \\ & \quad R_1 \quad \{ R.s := R_1.s \} \\ R \rightarrow - & \\ & \quad T \quad \{ R_1.i := \text{mknode}('-', R.i, T.nptr) \} \\ & \quad R_1 \quad \{ R.s := R_1.s \} \\ R \rightarrow \epsilon & \quad \{ R.s := R.i \} \\ T \rightarrow ( & \\ & \quad E \\ & \quad ) \quad \{ T.nptr := E.nptr \} \\ T \rightarrow id & \quad \{ T.nptr := \text{mkleaf}(id, id.entry) \} \\ T \rightarrow num & \quad \{ T.nptr := \text{mkleaf}(num, num.val) \} \end{aligned}$$

Рис. 5.28. Преобразованная схема трансляции для построения синтаксических деревьев

На рис. 5.29 показано, каким образом действия на рис. 5.28 строят синтаксическое дерево для выражения  $a - 4 + c$ . Синтезируемые атрибуты представлены справа от узлов грамматических символов, а наследуемые — слева. Листья синтаксического дерева создаются с помощью действий, связанных с продукциями  $T \rightarrow id$  и  $T \rightarrow num$ , как в примере 5.8. У крайнего слева  $T$  атрибут  $T.nptr$  указывает на лист для  $a$ . Указатель на узел для  $a$  наследуется как атрибут  $R.i$  в правой части  $E \rightarrow TR$ .

При применении продукции  $R \rightarrow -TR_1$  к правому потомку корня,  $R.i$  указывает на узел для  $a$ , а  $T.nptr$  — на узел для  $4$ . Узел для  $a - 4$  строится путем вызова функции  $\text{mknode}$  для оператора “ $-$ ” и этих указателей.

И наконец, при использовании продукции  $R \rightarrow \epsilon$  атрибут  $R.i$  указывает на корень синтаксического дерева в целом. Все дерево целиком возвращается как значение  $E.nptr$ , вычисляемое с помощью (не показанных на рис. 5.29) атрибутов  $s$  узлов для  $R$ .  $\square$

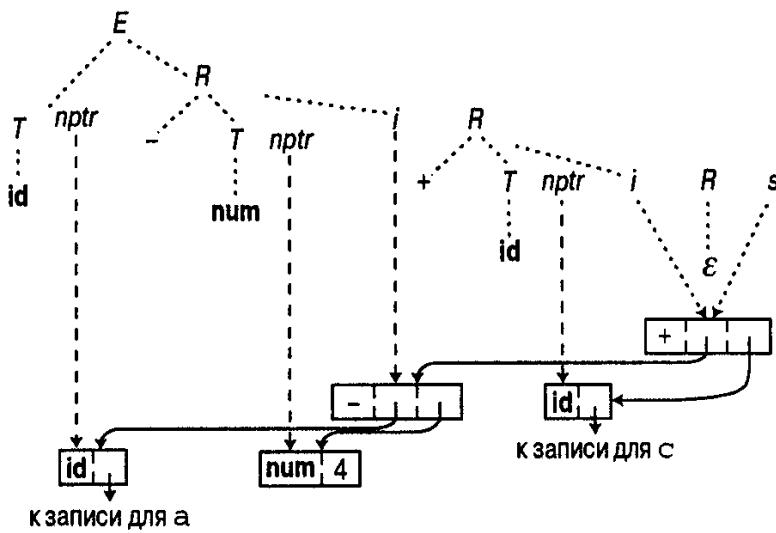


Рис. 5.29. Использование наследуемых атрибутов для построения синтаксических деревьев

## Разработка предиктивного транслятора

Приведенный далее алгоритм обобщает построение предиктивных синтаксических анализаторов для реализации схемы трансляции, основанной на пригодной для нисходящего синтаксического анализа грамматике.

### Алгоритм 5.2. Построение предиктивного синтаксически управляемого транслятора

**Вход.** Схема синтаксически управляемой трансляции, основанная на грамматике, пригодной для предиктивного синтаксического анализа.

**Выход.** Код синтаксически управляемого транслятора.

**Метод.** Данная технология представляет собой модификацию метода построения предиктивного синтаксического анализатора из раздела 2.4.

1. Для каждого нетерминала  $A$  создается функция, которая имеет формальный параметр для каждого наследуемого атрибута  $A$  и возвращает значения синтезируемых атрибутов  $A$  (возможно, в виде записи, указателя на запись с полями для каждого атрибута или с использованием механизма передачи параметров по ссылке, рассматриваемого в разделе 7.5). Для простоты полагаем, что каждый нетерминал имеет только один синтезируемый атрибут. Функция для  $A$  имеет локальную переменную для всех атрибутов каждого грамматического символа в продукции для  $A$ .
2. Как и в разделе 2.4, код для нетерминала  $A$  по текущему входному символу принимает решение о том, какая продукция должна использоваться.
3. Код, связанный с каждой продукцией, делает следующее (токены, нетерминалы и действия в правой части продукции рассматриваются слева направо).
  - i) Для токена  $X$  с синтезируемым атрибутом  $x$  его значение сохраняется в переменной, объявленной для  $X.x$ . Затем производится вызов для проверки соответствия токена  $X$  и текущего символа входного потока.
  - ii) Для нетерминала  $B$  генерируется присвоение  $c := B(b_1, b_2, \dots, b_k)$  с вызовом функции в правой части, где  $b_1, b_2, \dots, b_k$  — переменные для наследуемых атрибутов  $B$ , а  $c$  — переменная для синтезируемого атрибута  $B$ .
  - iii) Для действия код копируется в синтаксический анализатор с заменой каждой ссылки на атрибут переменной для этого атрибута.  $\square$

В разделе 5.7 алгоритм 5.2 расширен для реализации любого L-атрибутного определения; в разделе 5.8 будут рассмотрены пути усовершенствования трансляторов, построенных по алгоритму 5.2. Например, может оказаться возможным удаление операторов копирования вида  $x := y$  или использование одиночных переменных для хранения значений нескольких атрибутов. Некоторые усовершенствования могут быть выполнены автоматически с использованием методов из главы 10, “Оптимизация кода”.

### Пример 5.16

Грамматика на рис. 5.28 является LL(1)-грамматикой, а следовательно, пригодной для нисходящего синтаксического анализа. По атрибутам нетерминалов грамматики мы получаем следующие типы аргументов и возвращаемых результатов функций для  $E$ ,  $R$  и  $T$  (поскольку  $E$  и  $T$  не имеют наследуемых атрибутов, соответствующие функции не имеют аргументов).

```
function E : ↑узел_синтаксического_дерева;
function R (↑узел_синтаксического_дерева): ↑узел_синтаксического_дерева;
function T : ↑узел_синтаксического_дерева;
```

Мы объединяем две из  $R$ -продукций на рис. 5.28 для уменьшения транслятора. Новая продукция использует токен **addop** для представления + и -.

|                 |              |   |                                                  |   |  |       |
|-----------------|--------------|---|--------------------------------------------------|---|--|-------|
| $R \rightarrow$ | <b>addop</b> |   |                                                  |   |  |       |
|                 | $T$          | { | $R_1.i := mknodenode(addop.lexeme, R.i, T.nptr)$ | } |  | (5.5) |
|                 | $R_1$        | { | $R.s := R_1.s$                                   | } |  |       |
| $R \rightarrow$ | $\epsilon$   | { | $R.s := R.i$                                     | } |  |       |

Код для  $R$  основан на процедуре синтаксического анализа (рис. 5.30). Если текущий сканируемый символ — **addop**, то применяется продукция  $R \rightarrow addop T R$  с использованием процедуры *match* для чтения следующего за **addop** входного токена, а затем вызываются процедуры для  $T$  и  $R$ . В противном случае процедура не выполняет никаких действий, имитируя продукцию  $R \rightarrow \epsilon$ .

```
procedure R;
begin
 if lookahead = addop then begin
 match(addop); T; R
 end
 else begin /* ничего не делать */
 end
end;
```

Рис. 5.30. Процедура синтаксического анализа для продукции  $R \rightarrow addop T R | \epsilon$

```
function R(i: ↑syntax_tree_node) : ↑syntax_tree_node;
 var nptr, i1, s1, s: ↑syntax_tree_node;
 addoplexeme: char;
begin
 if lookahead = addop then begin
 /* продукция $R \rightarrow addop T R$ */
 addoplexeme := lexval;
 match(addop);
 nptr := T;
```

```

i1 := mknod(addoplexeme, i, nptr);
s1 := R(i1);
s := s1
end
else s:=i; /* продукция $R \rightarrow \epsilon$ */
return s
end;

```

Рис. 5.31. Построение синтаксических деревьев методом рекурсивного спуска

Процедура для  $R$  на рис. 5.31 содержит код вычисления атрибутов. Лексическое значение *lexval* токена *addop* сохраняется в переменной *addoplexeme*; после этого вызывается функция *match* для *addop*, затем функция  $T$ , и ее результат сохраняется в переменной *nptr*. Переменная *i1* соответствует наследуемому атрибуту  $R_1.i$ , а *s1* — синтезируемому атрибуту  $R_1.s$ . Инструкция *return* возвращает значение *s* при выходе из функции. Аналогично строятся функции  $E$  и  $T$ .  $\square$

## 5.6. Восходящее вычисление наследуемых атрибутов

В этом разделе мы представим метод реализации L-атрибутных определений в рамках восходящего синтаксического анализа. Этот метод применим к обработке всех L-атрибутных определений из предыдущего раздела. Он также пригоден для реализации любого L-атрибутного определения, основанного на LL(1)-грамматике. Метод способен также реализовать многие (хотя и не все) L-атрибутные определения, основанные на LR(1)-грамматиках. Этот метод является обобщением технологии восходящей трансляции, рассмотренной в разделе 5.3.

### Удаление внедренных действий из схемы трансляции

Все действия при восходящей трансляции находятся в правом конце продукции, в то время как в методе предиктивного синтаксического анализа из раздела 5.5 нам требовалось вставлять действия в разные места правой части. Чтобы начать обсуждение, каким образом наследуемые атрибуты могут обрабатываться снизу вверх, мы рассмотрим преобразование, по которому все вставленные действия в схеме трансляции располагаются в правых концах их продукции.

Это преобразование вставляет в базовую грамматику новые нетерминалы-маркеры, порождающие  $\epsilon$ . Мы заменяем каждое вставленное действие отдельным маркером  $M$  и добавляем действие в конец продукции  $M \rightarrow \epsilon$ . Например, схема трансляции

$$\begin{aligned}
 E &\rightarrow TR \\
 R &\rightarrow + T \{ \text{print}('+') \} R \mid - T \{ \text{print}(' - ') \} R \mid \epsilon \\
 T &\rightarrow \text{num} \{ \text{print}(\text{num}.val) \}
 \end{aligned}$$

преобразуется с помощью нетерминалов  $M$  и  $N$  в

$$\begin{aligned}
 E &\rightarrow TR \\
 R &\rightarrow + TM R \mid - TN R \mid \epsilon \\
 T &\rightarrow \text{num} \{ \text{print}(\text{num}.val) \} \\
 M &\rightarrow \epsilon \{ \text{print}('+') \} \\
 N &\rightarrow \epsilon \{ \text{print}(' - ') \}
 \end{aligned}$$

Грамматики в этих двух схемах трансляции задают один и тот же язык, и, изобразив дерево разбора с дополнительными узлами для действий, мы можем показать, что действия выполняются в одном и том же порядке. Действия в преобразованной схеме трансляции завершают продукцию, так что они могут быть выполнены непосредственно перед сверткой правой части в процессе восходящего разбора.

## Наследование атрибутов в стеке синтаксического анализатора

Восходящий синтаксический анализатор сворачивает правую часть продукции  $A \rightarrow XY$  удалением  $X$  и  $Y$  с вершины стека синтаксического анализатора и заменой их на  $A$ . Предположим, что  $X$  имеет синтезируемый атрибут  $X.s$  (который в соответствии с реализацией из раздела 5.3 хранится в стеке вместе с  $X$ ).

Поскольку значение  $X.s$  уже находится в стеке перед выполнением любой свертки в поддереве ниже  $Y$ , это значение может быть наследовано  $Y$ . Таким образом, если наследуемый атрибут  $Y.i$  определен правилом копирования  $Y.i := X.s$ , то значение  $X.s$  может быть использовано там, где требуется  $Y.i$ . Как мы увидим, правила копирования играют важную роль в вычислении наследуемых атрибутов в процессе восходящего синтаксического анализа.

### Пример 5.17

Тип идентификатора может быть передан с помощью правил копирования и наследуемых атрибутов, как показано на рис. 5.32 (он представляет собой адаптированный рис. 5.7). Рассмотрим вначале действия восходящего синтаксического анализатора при входной строке

```
real p, q, r
```

Затем мы покажем, каким образом можно получить значение атрибута  $T.type$  при использовании продукции для  $L$ . Схема трансляции, которую мы хотим реализовать, имеет следующий вид

$$\begin{array}{ll}
 D \rightarrow T & \{ L.in := T.type \} \\
 & L \\
 T \rightarrow \text{int} & \{ T.type := \text{integer} \} \\
 T \rightarrow \text{real} & \{ T.type := \text{real} \} \\
 L \rightarrow & \{ L_1.in := L.in \} \\
 & L_1, \text{id} \{ \text{addtype(id.entry, } L.in) \} \\
 L \rightarrow \text{id} & \{ \text{addtype(id.entry, } L.in) \}
 \end{array}$$

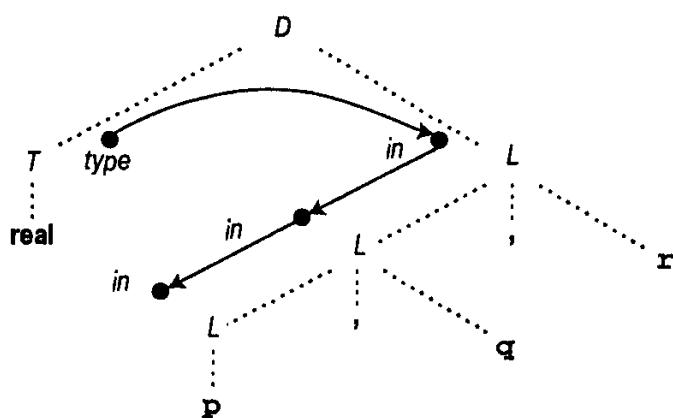


Рис. 5.32.  $L.in = T.type$  в каждом узле для  $L$

Если проигнорировать действия в приведенной выше схеме трансляции, то последовательность шагов синтаксического анализатора при входном потоке, показанном на рис. 5.32, будет такой, как на рис. 5.33. Для ясности вместо состояния стека здесь показан соответствующий символ грамматики, а вместо токенов **id** — соответствующие им идентификаторы.

| ВХОДНОЙ ПОТОК       | СОСТОЯНИЕ СТЕКА | ИСПОЛЬЗОВАННАЯ ПРОДУКЦИЯ     |
|---------------------|-----------------|------------------------------|
| <b>real</b> p, q, r | -               |                              |
| p, q, r             | <b>real</b>     |                              |
| p, q, r             | T               | $T \rightarrow \text{real}$  |
| ,                   | $T_p$           |                              |
| ,                   | $TL$            | $L \rightarrow \text{id}$    |
| q, r                | $TL,$           |                              |
| ,                   | $TL, q$         |                              |
| ,                   | $TL$            | $L \rightarrow L, \text{id}$ |
| r                   | $TL,$           |                              |
|                     | $TL, r$         |                              |
|                     | $TL$            | $L \rightarrow L, \text{id}$ |
| D                   |                 | $D \rightarrow TL$           |

Рис. 5.33. При свертке правой части продукции для  $L$ ,  $T$  находится непосредственно под правой частью

Предположим, что, как и в разделе 5.3, стек синтаксического анализатора реализован в виде пары массивов — *state* и *val*. Если *state*[*i*] — грамматический символ  $X$ , то в *val*[*i*] хранится синтезируемый атрибут  $X.s$ . Содержимое массива *state* показано на рис. 5.33. Заметьте, что всякий раз, когда сворачивается правая часть продукции  $L$ , в стеке непосредственно под правой частью располагается  $T$ . Этот факт можно использовать для доступа к значению атрибута  $T.type$ .

Реализация на рис. 5.34 использует информацию о местоположении  $T.type$  в стеке *val* относительно его вершины. Пусть *top* и *ntop* указывают верхний элемент стека непосредственно перед сверткой и после нее. Из правил копирования, определяющих  $L.in$ , мы знаем, что вместо  $L.in$  можно использовать  $T.type$ .

| ПРОДУКЦИЯ                    | ФРАГМЕНТ КОДА                          |
|------------------------------|----------------------------------------|
| $D \rightarrow TL;$          |                                        |
| $T \rightarrow \text{int}$   | $val[ntop] := \text{integer}$          |
| $T \rightarrow \text{real}$  | $val[ntop] := \text{real}$             |
| $L \rightarrow L, \text{id}$ | $\text{addtype}(val[top], val[top-3])$ |
| $L \rightarrow \text{id}$    | $\text{addtype}(val[top], val[top-1])$ |

Рис. 5.34. Значение  $T.type$  используется вместо  $L.in$

При применении продукции  $L \rightarrow \text{id}$  на вершине стека *val* находится  $\text{id}.entry$ , а непосредственно под ним —  $T.type$ . Следовательно,  $\text{addtype}(val[top], val[top-1])$  эквивалентно  $\text{addtype}(\text{id}.entry, T.type)$ . Аналогично, поскольку правая часть продукции

$L \rightarrow L$ ,  $\text{id}$  имеет три символа,  $T.type$  при ее свертке находится в  $val[top-3]$ . Правила копирования для атрибута  $L.in$  таким образом устраниются, поскольку вместо него используется значение  $T.type$  из стека.  $\square$

## Моделирование вычисления наследуемых атрибутов

Использование значения атрибута из стека применимо только в случае, когда грамматика позволяет предсказать позицию значения атрибута (для данной технологии синтаксического анализа — Прим. ред.).

### Пример 5.18

В качестве примера, когда мы не в состоянии предсказать позицию, рассмотрим следующую схему трансляции

| ПРОДУКЦИЯ            | СЕМАНТИЧЕСКИЕ ПРАВИЛА |       |
|----------------------|-----------------------|-------|
| $S \rightarrow aAC$  | $C.i := A.s$          |       |
| $S \rightarrow bABC$ | $C.i := A.s$          |       |
| $C \rightarrow c$    | $C.s := g(C.i)$       | (5.6) |

$C$  наследует синтезируемый атрибут  $A.s$  согласно правилу копирования. Заметим, что в стеке между  $A$  и  $C$  может располагаться  $B$  (а может и отсутствовать). При свертке  $C \rightarrow c$  значение  $C.i$  находится либо в  $val[top-1]$ , либо в  $val[top-2]$ , но неясно, какой именно случай реализуется в данный момент.

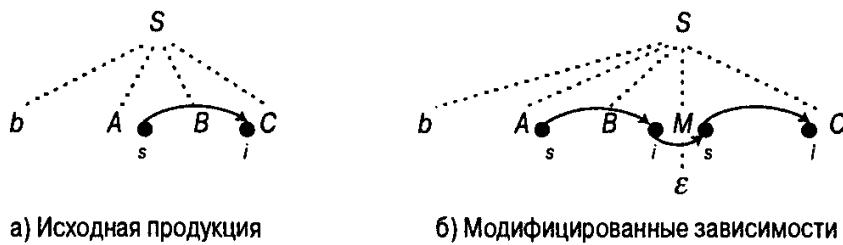


Рис. 5.35. Копирование значения атрибута с помощью маркера  $M$

На рис. 5.35 непосредственно перед  $C$  в правой части второй продукции (5.6) вставлен дополнительный маркер  $M$ . Атрибут  $C.i$  при разборе в соответствии с продукцией  $S \rightarrow bABMC$  наследует значение  $A.s$  опосредованно, через  $M.i$  и  $M.s$ . При применении продукции  $M \rightarrow \epsilon$  правило копирования  $M.s := M.i$  гарантирует, что значение  $M.s = M.i = A.s$  располагается непосредственно перед частью стека, используемой для разбора поддерева для  $C$ . Таким образом, значение  $C.i$  может быть найдено в  $val[top-1]$  при использовании продукции  $C \rightarrow c$ , независимо от того, какая из продукции приведенной ниже модифицированной грамматики — первая или вторая — использовалась до этого.

| ПРОДУКЦИЯ                | СЕМАНТИЧЕСКИЕ ПРАВИЛА    |
|--------------------------|--------------------------|
| $S \rightarrow aAC$      | $C.i := A.s$             |
| $S \rightarrow bABC$     | $M.i := A.s; C.i := M.s$ |
| $C \rightarrow c$        | $C.s := g(C.i)$          |
| $M \rightarrow \epsilon$ | $M.s := M.i$             |

Нетерминалы-маркеры могут также использоваться для моделирования семантических правил, не являющихся правилами копирования. Рассмотрим, например

| ПРОДУКЦИЯ           | СЕМАНТИЧЕСКИЕ ПРАВИЛА |       |
|---------------------|-----------------------|-------|
| $S \rightarrow aAC$ | $C.i := f(A.s)$       | (5.7) |

В этот раз правило, определяющее  $C.i$ , не является правилом копирования, так что значение  $C.i$  уже не находится в стеке  $val$ . Эта проблема также решается с помощью маркера.

| ПРОДУКЦИЯ                | СЕМАНТИЧЕСКИЕ ПРАВИЛА    |       |
|--------------------------|--------------------------|-------|
| $S \rightarrow aANC$     | $N.i := A.s; C.i := N.s$ | (5.8) |
| $N \rightarrow \epsilon$ | $N.s := f(N.i)$          |       |

Отдельный нетерминал  $N$  наследует  $A.s$  согласно правилу копирования. Его синтезируемый атрибут  $N.s$  устанавливается равным  $f(A.s)$ ; после этого  $C.i$  наследует это значение с использованием правила копирования. При свертке в соответствии с продукцией  $N \rightarrow \epsilon$  мы находим значение  $N.i$  на месте  $A.s$ , т.е. в  $val[top-1]$ . При свертке в соответствии с  $S \rightarrow aANC$  значение  $C.i$  также находится в  $val[top-1]$ , поскольку это —  $N.s$ . В действительности в этот момент нам не требуется значение  $C.i$ ; оно требовалось только в процессе свертки строки терминалов в  $C$ .

### Пример 5.19

На рис. 5.36 для гарантии того, что значение наследуемого атрибута  $B.ps$  находится при свертке поддерева для  $B$  в определенной позиции, используются три маркера —  $L$ ,  $M$  и  $N$ . Исходная грамматика была представлена на рис. 5.22, а ее связь с форматированием текста раскрыта в примере 5.13.

| ПРОДУКЦИЯ                              | СЕМАНТИЧЕСКИЕ ПРАВИЛА                 |
|----------------------------------------|---------------------------------------|
| $S \rightarrow LB$                     | $B.ps := L.s$                         |
| $L \rightarrow \epsilon$               | $S.ht := B.ht$                        |
| $B \rightarrow B_1 MB_2$               | $L.s := 10$                           |
|                                        | $B_1.ps := B.ps$                      |
|                                        | $M.i := B.ps$                         |
|                                        | $B_2.ps := M.s$                       |
|                                        | $B.ht := \max(B_1.ht, B_2.ht)$        |
| $B \rightarrow B_1 \text{ sub } N B_2$ | $B_1.ps := B.ps$                      |
|                                        | $N.i := B.ps$                         |
|                                        | $B_2.ps := N.s$                       |
|                                        | $B.ht := \text{disp}(B_1.ht, B_2.ht)$ |
| $B \rightarrow \text{text}$            | $B.ht := \text{text}.h \times B.ps$   |
| $M \rightarrow \epsilon$               | $M.s := M.i$                          |
| $N \rightarrow \epsilon$               | $N.s := \text{shrink}(N.i)$           |

Рис. 5.36. Наследуемые атрибуты определяются правилами копирования

Инициализация выполняется с использованием  $L$ . Продукция для  $S$  —  $S \rightarrow LB$  на рис. 5.36, так что  $L$  остается в стеке в процессе свертки поддерева ниже  $B$ . Значение 10 наследуемого атрибута  $B.ps = L.s$  вводится в стек в соответствии с правилом  $L.s := 10$ , связанным с  $L \rightarrow \epsilon$ .

Маркер  $M$  в  $B \rightarrow B_1 M B_2$  играет роль, аналогичную роли  $M$  на рис. 5.35; он гарантирует, что значение  $B.ps$  в стеке синтаксического анализатора находится непосредственно под  $B_2$ . В продукции  $B \rightarrow B_1 \text{ sub } N B_2$  нетерминал  $N$  используется так же, как и в (5.8). Согласно правилу копирования  $N.i := B.ps$ ,  $N$  наследует значение атрибута, от которого зависит значение  $B_2.ps$ , и синтезирует значение  $B_2.ps$  согласно правилу  $N.s := \text{shrink}(N.i)$ . Вывод, который мы оставляем в качестве упражнения, гласит, что значение  $B.ps$  всегда располагается непосредственно под правой частью при ее свертке к  $B$ .

Фрагменты кода, реализующие синтаксически управляемое определение на рис. 5.36, показаны на рис. 5.37. Все наследуемые атрибуты устанавливаются посредством правил копирования, представленных на рис. 5.36, так что в реализации соответствующие значения получаются отслеживанием позиций атрибутов в стеке  $val$ . Как и в предыдущем примере,  $top$  и  $ntop$  указывают вершину стека соответственно до и после свертки.  $\square$

| ПРОДУКЦИЯ                              | ФРАГМЕНТ КОДА                                    |
|----------------------------------------|--------------------------------------------------|
| $S \rightarrow L B$                    | $val[ntop] := val[top]$                          |
| $L \rightarrow \epsilon$               | $val[ntop] := 10$                                |
| $B \rightarrow B_1 M B_2$              | $val[ntop] := \max(val[top-2], val[top])$        |
| $B \rightarrow B_1 \text{ sub } N B_2$ | $val[ntop] := \text{disp}(val[top-3], val[top])$ |
| $B \rightarrow \text{text}$            | $val[ntop] := val[top] \times val[top-1]$        |
| $M \rightarrow \epsilon$               | $val[ntop] := val[top-1]$                        |
| $N \rightarrow \epsilon$               | $val[ntop] := \text{shrink}(val[top-2])$         |

Рис. 5.37. Реализация синтаксически управляемого определения на рис. 5.36

Систематическое введение маркеров (как в случае модификации (5.6) и (5.7)) может сделать возможным вычисление L-атрибутных определений в процессе LR-разбора. Поскольку для каждого маркера имеется ровно одна продукция, грамматика остается LL(1)-грамматикой при добавлении маркеров. Любая LL(1)-грамматика является также LR(1)-грамматикой, так что при добавлении маркеров к LL(1)-грамматике не возникает никаких конфликтов синтаксического анализа. К сожалению, этого нельзя сказать обо всех LR(1)-грамматиках; так что при введении маркеров в некоторые LR(1)-грамматики могут возникать конфликты синтаксического анализа.

Идеи, изложенные в предшествующих примерах, могут быть формализованы в виде следующего алгоритма.

### Алгоритм 5.3. Восходящий синтаксический анализ и трансляция с наследуемыми атрибутами

**Вход.** L-атрибутное определение и лежащая в его основе LL(1)-грамматика.

**Выход.** Синтаксический анализатор, который вычисляет значения всех атрибутов в стеке.

**Метод.** Для простоты предположим, что каждый нетерминал  $A$  имеет один наследуемый атрибут  $A.i$ , а каждый грамматический символ  $X$  — синтезируемый атрибут  $X.s$ . Если  $X$  — терминал, то его синтезируемый атрибут в действительности представляет собой лексическое значение, возвращаемое лексическим анализатором вместе с  $X$ ; это лексическое значение размещается в стеке, в массиве  $val$ , как и в предыдущих примерах.

Для каждой продукции  $A \rightarrow X_1 \dots X_n$  вводится  $n$  новых маркеров-нетерминалов  $M_1, \dots, M_n$ , и продукция заменяется следующей:  $A \rightarrow M_1 X_1 \dots M_n X_n$ <sup>4</sup>. Синтезируемый атрибут  $X_j.s$  вносится в стек синтаксического анализатора, в запись массива  $val$ , связанную с  $X_j$ . Наследуемый атрибут  $X_j.i$ , если таковой существует, располагается в том же массиве, но связан с  $M_j$ .

Важным инвариантом в процессе анализа оказывается то, что наследуемый атрибут  $A.i$ , если таковой существует, находится в массиве  $val$  непосредственно под  $M_1$ . Поскольку мы предположили, что у стартового символа нет наследуемого атрибута, нет и проблем в случае, когда стартовым символом оказывается  $A$ , но даже если такой наследуемый атрибут и есть, он может быть размещен ниже дна стека. Указанную инвариантность можно легко доказать применением индукции по числу шагов восходящего разбора, обратив внимание на то, что наследуемые атрибуты связаны с маркерами  $M_j$  и атрибут  $X_j.i$  вычисляется в  $M_j$  перед началом свертки к  $X_j$ .

Чтобы убедиться, что атрибуты могут быть вычислены, как и предполагалось, в процессе восходящего разбора, рассмотрим два случая. Во-первых, если мы выполняем свертку к маркеру  $M_j$ , то знаем, какой продукции  $A \rightarrow M_1 X_1 \dots M_n X_n$  принадлежит этот маркер. Следовательно, мы знаем позицию любого атрибута, необходимого для вычисления наследуемого атрибута  $X_j.i$ .  $A.i$  располагается в  $val[top-2j+2]$ ,  $X_1.i$  — в  $val[top-2j+3]$ ,  $X_1.s$  — в  $val[top-2j+4]$ ,  $X_2.i$  — в  $val[top-2j+5]$  и т.д. Значит, мы можем вычислить  $X_j.i$  и сохранить его в  $val[top+1]$ , элементе, который становится новой вершиной стека после свертки. Заметьте, насколько важен тот факт, что грамматика — LL(1); иначе мы бы не могли быть уверены в том, что  $\epsilon$  свертывается в один определенный маркер, а следовательно, не могли бы определить местонахождение нужных атрибутов или даже знать, какую формулу следует применить. Мы предлагаем читателю либо принять это на веру, либо самостоятельно вывести доказательство из того факта, что любая LL(1)-грамматика с маркерами остается LR(1)-грамматикой.

Во-вторых, при свертке к символу, не являющемуся маркером (например, в соответствии с продукцией  $A \rightarrow M_1 X_1 \dots M_n X_n$ ), мы должны вычислить только синтезируемый атрибут  $A.s$ . Заметьте, что наследуемый атрибут  $A.i$  уже вычислен и находится в стеке непосредственно под позицией, в которую мы вставляем сам  $A$ . Ясно, что необходимые для вычисления  $A.s$  атрибуты доступны и расположены в стеке в позициях  $X_j$  во время свертки.

Следующие упрощения сокращают количество маркеров; второе устраниет конфликты синтаксического анализа в леворекурсивных грамматиках.

- Если  $X_j$  не имеет наследуемого атрибута, нам не требуется маркер  $M_j$ . Конечно, если опустить  $M_j$ , позиция атрибута в стеке изменится, но это изменение может быть легко встроено в синтаксический анализатор.
- Если  $X_1.i$  существует, но вычисляется согласно правилу копирования  $X_1.i = A.i$ , то мы можем опустить  $M_1$ , поскольку в соответствии с нашим инвариантом мы знаем, что  $A.i$  находится в стеке непосредственно под  $X_1$  и может служить в качестве  $X_1.i$ .  $\square$

---

<sup>4</sup> Хотя вставка  $M_1$  перед  $X_1$  и упрощает обсуждение маркеров-нетерминалов, она имеет и нежелательный побочный эффект внесения конфликтов в леворекурсивную грамматику (см. упр. 5.21). Как указано ниже, маркер  $M_1$  может быть удален.

## Замена наследуемых атрибутов синтезируемыми

Иногда оказывается возможным избежать использования наследуемых атрибутов путем изменения грамматики. Например, объявление в Pascal может состоять из списка идентификаторов, за которым следует тип, например `m, n: integer`. Грамматика для такого объявления может включать продукцию вида

$$\begin{aligned} D &\rightarrow L : T \\ T &\rightarrow \text{integer} \mid \text{char} \\ L &\rightarrow L, \text{id} \mid \text{id} \end{aligned}$$

Поскольку идентификаторы порождаются  $L$ , но тип в поддереве  $L$  не содержится, мы не можем связать тип с идентификатором с помощью только синтезируемых атрибутов. Если нетерминал  $L$  наследует тип от  $T$ , мы получаем синтаксически управляемое определение, не являющееся  $L$ -атрибутным, так что трансляция на его основе не может быть выполнена в процессе синтаксического анализа.

Решение этой проблемы состоит в реструктуризации грамматики для включения типа в качестве последнего элемента в список идентификаторов.

$$\begin{aligned} D &\rightarrow \text{id } L \\ L &\rightarrow , \text{id } L \mid : T \\ T &\rightarrow \text{integer} \mid \text{char} \end{aligned}$$

Теперь тип может рассматриваться как синтезируемый атрибут  $L.type$ , и в таблицу символов можно внести тип каждого идентификатора, порожденного  $L$ .

## Сложное синтаксически управляемое определение

Алгоритм 5.3 реализации наследуемых атрибутов в процессе восходящего синтаксического анализа может быть расширен для некоторых (но не всех) LR-грамматик.  $L$ -атрибутное определение на рис. 5.38 основано на простой LR(1)-грамматике, но не может быть реализовано в процессе LR-разбора. Нетерминал  $L$  в  $L \rightarrow \epsilon$  наследует количество единиц, порожденных из  $S$ . Поскольку продукция  $L \rightarrow \epsilon$  свертывается восходящим синтаксическим анализатором первой, в этот момент транслятор еще не знает количество единиц во входном потоке.

| ПРОДУКЦИЯ                | СЕМАНТИЧЕСКИЕ ПРАВИЛА      |
|--------------------------|----------------------------|
| $S \rightarrow L$        | $L.count := 0$             |
| $L \rightarrow L_1 1$    | $L_1.count := L.count + 1$ |
| $L \rightarrow \epsilon$ | $print(L.count)$           |

Рис. 5.38. Сложное синтаксически управляемое определение

## 5.7. Рекурсивные вычислители

Рекурсивные функции, вычисляющие атрибуты при обходе дерева разбора, могут быть построены на основе синтаксически управляемых определений с использованием обобщения технологий предиктивной трансляции из раздела 5.5. Такие функции позволяют нам реализовать синтаксически управляемые определения, которые не могут быть реализованы одновременно с синтаксическим анализом. В этом разделе мы свяжем с ка-

ждым нетерминалом отдельную функцию трансляции. Эта функция “посещает” всех потомков узла для нетерминала в порядке, определяемом продукцией в узле (не обязательно в порядке слева направо). В разделе 5.10 мы увидим, как можно добиться эффекта многопроходной трансляции путем назначения нетерминалам нескольких функций.

## Обход слева направо

В алгоритме 5.2 мы показали, каким образом можно реализовать L-атрибутное определение на базе LL(1)-грамматики путем построения рекурсивной функции, которая анализирует и транслирует каждый терминал. Все L-атрибутные синтаксически управляемые определения могут быть реализованы, если подобная рекурсивная функция вызывается в узле для нетерминала в предварительно построенном дереве разбора. Такая функция может определить потомков узла, просматривая продукцию, связанную с ним. Функция для нетерминала  $A$  получает в качестве аргументов узел и значения наследуемых атрибутов  $A$  и возвращает в качестве результата значения синтезируемых атрибутов  $A$ .

Построение функции аналогично построению в алгоритме 5.2, кроме шага 2, где функция для нетерминала определяет используемую продукцию, основываясь на текущем входном символе. Здесь для определения используемой в узле продукции функция обычно применяет инструкцию выбора `case`. Вот пример, иллюстрирующий этот метод.

### Пример 5.20

Рассмотрим синтаксически управляемое определение для вычисления размера и высоты формул на рис. 5.22. Нетерминал  $B$  имеет наследуемый атрибут  $ps$  и синтезируемый атрибут  $ht$ . Используя модифицированный в соответствии со сказанным выше алгоритмом 5.2, построим функцию для  $B$ , приведенную на рис. 5.39.

```
function B(n, ps);
 var ps1, ps2, ht1, ht2;
begin
 case Продукция в узле n of
 'B → B1B2':
 ps1 := ps;
 ht1 := B(child(n,1), ps1);
 ps2 := ps;
 ht2 := B(child(n,2), ps2);
 return max(ht1, ht2);
 'B → B1 sub B2':
 ps1 := ps;
 ht1 := B(child(n,1), ps1);
 ps2 := shrink(ps);
 ht2 := B(child(n,3), ps2);
 return disp(ht1, ht2);
 'B → text':
 return ps × text.h;
 default:
 error
 end
end
```

Рис. 5.39. Функция для нетерминала  $B$  на рис. 5.22

Функция  $B$  получает в качестве аргументов узел  $n$  и значение, соответствующее  $B.ps$  в этом узле, и возвращает значение, соответствующее  $B.ht$  в узле  $n$ . Функция использует инструкцию `case` для каждой продукции с  $B$  в левой части. Код, соответствующий каждой продукции, моделирует семантические правила, связанные с ней. Порядок применения правил должен быть таким, чтобы наследуемые атрибуты нетерминала вычислялись до вызова функции для этого нетерминала.

В коде, соответствующем продукции  $B \rightarrow B \text{ sub } B$ , переменные  $ps$ ,  $ps1$  и  $ps2$  хранят значения наследуемых атрибутов  $B.ps$ ,  $B_1.ps$  и  $B_2.ps$ . Аналогично  $ht$ ,  $ht1$  и  $ht2$  хранят значения  $B.ht$ ,  $B_1.ht$  и  $B_2.ht$ . Функция  $child(m,i)$  используется для обращения к  $i$ -му потомку узла  $m$ . Поскольку  $B_2$  является меткой третьего дочернего по отношению к  $n$  узла, значение  $B_2.ht$  определяется вызовом функции  $B(child(n,3), ps2)$ .  $\square$

## Другие обходы

После того, как становится доступным построенное явно дерево разбора, мы можем посещать потомков узла в любом порядке. Рассмотрим не-L-атрибутное определение из примера 5.21. В трансляции, задаваемой этим определением, потомки узла для одной продукции должны посещаться слева направо, в то время как потомки узла для другой — справа налево.

Этот абстрактный пример иллюстрирует мощь использования взаимно рекурсивных функций для вычисления атрибутов в узлах дерева разбора. Эти функции не зависят от порядка, в котором создаются узлы дерева разбора. Принципиальным при вычислении в процессе обхода является то, чтобы наследуемые атрибуты в узле были вычислены до первого посещения узла, а синтезируемые атрибуты — до того, как мы в последний раз покинем узел.

### Пример 5.21

Каждый из нетерминалов на рис. 5.40 имеет наследуемый атрибут  $i$  и синтезируемый атрибут  $s$ . Показаны также графы зависимости для двух продукции. Правила, связанные с  $A \rightarrow LM$ , определяют зависимости слева направо, а правила, связанные с  $A \rightarrow QR$ , — справа налево.

| ПРОДУКЦИЯ          | СЕМАНТИЧЕСКИЕ ПРАВИЛА                                 |
|--------------------|-------------------------------------------------------|
| $A \rightarrow LM$ | $L.i := l(A.i)$<br>$M.i := m(L.s)$<br>$A.s := f(M.s)$ |
| $A \rightarrow QR$ | $R.i := r(a.i)$<br>$Q.i := q(A.i)$<br>$A.s := f(Q.s)$ |

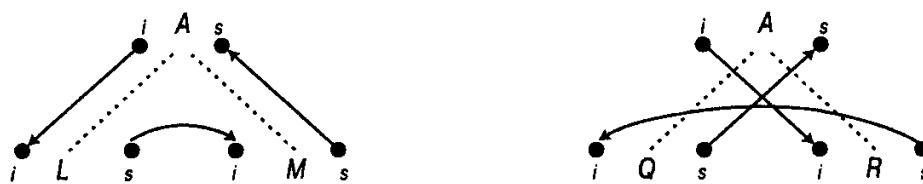


Рис. 5.40. Продукции и семантические правила для нетерминала  $A$

Функция для нетерминала  $A$  показана на рис. 5.41 (мы считаем построенными функции для  $L$ ,  $M$ ,  $Q$  и  $R$ ). Переменные в этом листинге именуются в соответствии с нетерминалом и его атрибутами, т.е.  $li$  означает переменную для  $L.i$ , а  $ls$  — для  $L.s$ .

```

function A(n, ai);
begin
 case Продукция в узле n of
 'A → LM': /* Порядок слева направо */
 li := l(ai);
 ls := L(child(n,1), li);
 mi := m(ls);
 ms := M(child(n,2), mi);
 return f(ms);
 'A → QR': /* Порядок справа налево */
 ri := r(ai);
 rs := R(child(n,2), ri);
 qi := q(rs);
 qs := Q(child(n,1), qi);
 return f(qs);
 default:
 error
 end
end;

```

Рис. 5.41. Зависимости, показанные на рис. 5.40, определяют порядок посещения дочерних узлов

Код, соответствующий продукции  $A \rightarrow LM$ , строится так же, как и в примере 5.20 — мы определяем наследуемый атрибут символа  $L$ , вызываем функцию  $L$  для определения синтезируемого атрибута  $L$  и повторяем процесс для  $M$ . Код, соответствующий  $A \rightarrow QR$ , проходит поддерево для  $R$  перед тем, как посетить поддерево для  $Q$ . В остальном код для этих двух продукции очень похож.  $\square$

## 5.8. Память для значений атрибутов во время компиляции

В этом разделе рассматривается назначение памяти для хранения значений атрибутов в процессе компиляции. Мы будем использовать информацию из графа зависимости для дерева разбора, так что подход из этого раздела применим в методах, использующих деревья разбора и определяющих порядок вычислений по графу зависимости. В следующем разделе будет описан случай, когда порядок вычислений может быть определен заранее, так что вопрос о выделении памяти для атрибутов при построении компилятора можно решить один раз.

При заданном порядке вычисления атрибутов (не обязательно в глубину) *время жизни* (lifetime) атрибута начинается, когда атрибут впервые вычисляется, и заканчивается, когда вычислены все атрибуты, зависящие от него. Мы можем сэкономить память, сохраняя значения атрибутов только на протяжении их времени жизни.

Для того чтобы подчеркнуть, что технологии из этого раздела применимы к любому порядку вычислений, рассмотрим не являющееся L-атрибутным синтаксически

управляемое определение, предназначенное для передачи информации о типе идентификаторам в объявлении.

### Пример 5.22

Синтаксически управляемое определение на рис. 5.42 представляет собой расширение определения (рис. 5.4), допускающее объявления вида

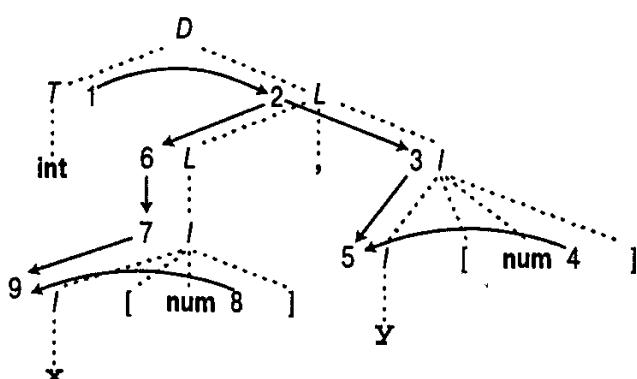
`real c[12][31];` (5.9)

`int x[3], y[5];` (5.10)

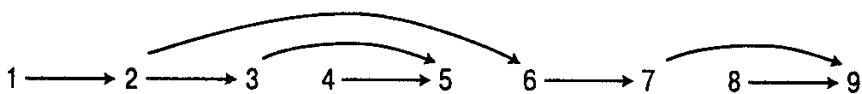
Дерево разбора для (5.10) показано на рис. 5.43 $a$  пунктирными линиями. Числа в узлах рассматриваются в следующем примере. Как и в примере 5.3, тип, полученный из  $T$ , наследуется  $L$  и передается вниз идентификаторам в объявлении. Дуга от  $T.type$  к  $L.in$  показывает, что  $L.in$  зависит от  $T.type$ . Синтаксически управляемое определение на рис. 5.42 не является L-атрибутным, поскольку  $I_1.in$  зависит от `num.val`, а `num` располагается справа от  $I_1$  в  $I \rightarrow I_1 [num]$ .  $\square$

| ПРОДУКЦИЯ                 | ПРАВИЛА                          |
|---------------------------|----------------------------------|
| $D \rightarrow TL$        | $L.in := T.type$                 |
| $T \rightarrow int$       | $T.type := integer$              |
| $T \rightarrow real$      | $T.type := real$                 |
| $L \rightarrow L_1, I$    | $L_1.in := L.in$                 |
|                           | $I.in := L.in$                   |
| $L \rightarrow I$         | $I.in := L.in$                   |
| $I \rightarrow I_1 [num]$ | $I_1.in := array(num.val, I.in)$ |
| $I \rightarrow id$        | $addtype(id.entry, I.in)$        |

Рис. 5.42. Передача типа идентификаторам в объявлении



а) Граф зависимости для дерева разбора



б) Узлы в порядке вычисления (а)

Рис. 5.43. Определение времени жизни значений атрибутов

## Назначение памяти атрибутам во время компиляции

Предположим, дана последовательность регистров для хранения значений атрибутов. Для удобства допустим, что каждый регистр может хранить любое значение атрибута (если атрибуты представляют различные типы, мы можем сформировать группы атрибутов, требующих одного и того же количества памяти, и рассматривать каждую группу в отдельности). Для определения регистров, в которых будут вычисляться атрибуты, мы используем информацию о времени жизни атрибутов.

### Пример 5.23

Предположим, что атрибуты вычисляются в порядке, заданном номерами узлов в графе зависимости на рис. 5.43<sup>5</sup>, построенном в последнем примере. Время жизни каждого узла начинается в момент вычисления его атрибута и заканчивается, когда этот атрибут используется в последний раз. Например, время жизни узла 1 заканчивается, когда вычислен атрибут узла 2, поскольку 2 — единственный узел, зависящий от 1. Аналогично время жизни узла 2 заканчивается при вычислении атрибута узла 6. □

На рис. 5.44 представлен метод вычисления атрибутов, использующий минимально возможное количество регистров. Мы рассматриваем узлы графа зависимости  $D$  дерева разбора в порядке их вычисления. Изначально у нас имеется пул регистров  $r_1, r_2, \dots$ . Если атрибут  $b$  определен семантическим правилом  $b := f(c_1, c_2, \dots, c_k)$ , то при вычислении  $b$  может закончиться время жизни одного или нескольких  $c_i$ . Соответствующие им регистры освобождаются после вычисления  $b$ . По возможности  $b$  вычисляется в регистре, хранившем один из  $c_1, c_2, \dots, c_k$ .

```
for каждый узел m из m_1, m_2, \dots, m_n do begin
 for каждый узел n , время жизни которого
 заканчивается при вычислении узла m do
 Пометить n -й регистр;
 if Некоторый регистр r помечен then begin
 Снять пометку r ;
 Вычислить m и разместить в регистре r ;
 Вернуть помеченные регистры в пул
 end
 else /* Помеченных регистров нет */
 Вычислить m и разместить в регистре из пула;
 /* Сюда можно вставить действия,
 использующие значение m */
 if Время жизни m завершилось then
 Вернуть регистр m в пул
end
```

Рис. 5.44. Назначение регистров значениям атрибутов

<sup>5</sup> В графе зависимости на рис. 5.43 не показаны узлы, соответствующие семантическому правилу  $addtype(id.entry, I.in)$ , поскольку для фиктивных атрибутов память не выделяется. Заметим, однако, что семантическое правило не должно выполняться до тех пор, пока не станет доступным значение  $I.in$ . Алгоритм для определения доступности должен работать с графиком зависимости, содержащим узлы для этого семантического правила.

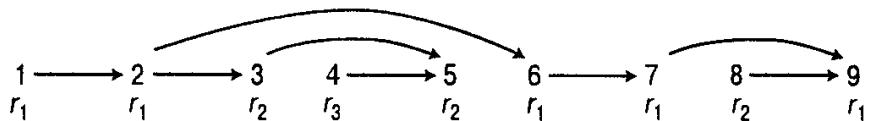


Рис. 5.45. Регистры, используемые для значений атрибутов на рис. 5.43б

Регистры, используемые в процессе вычисления графа зависимостей на рис. 5.43б, показаны на рис. 5.45. Мы начинаем с вычисления узла 1 в регистре  $r_1$ . Время жизни узла 1 заканчивается при вычислении узла 2, поэтому узел 2 использует тот же регистр  $r_1$ . Узел 3 получает новый регистр  $r_2$ , поскольку значение узла 2 потребуется при вычислении узла 6.

## Устранение копий

Мы можем усовершенствовать метод, приведенный на рис. 5.44, рассматривая правила копирования как специальный случай. Правило копирования имеет вид  $b := c$ , так что если значение  $c$  находится в регистре  $r$ , то там же автоматически размещается и значение  $b$ . Количество атрибутов, определяемых правилами копирования, может оказаться значительным, и явного копирования желательно избегать.

Множество узлов, имеющих одно и то же значение, образует класс эквивалентности. Метод на рис. 5.44 может быть модифицирован для хранения значения класса эквивалентности в регистре следующим образом. При рассмотрении узла  $m$  мы вначале проверяем, не определяется ли его значение правилом копирования. Если это так, следовательно, его значение уже находится в регистре и  $m$  присоединяется к классу эквивалентности со значением в этом регистре. Кроме того, такой регистр возвращается в пул только в конце времени жизни всех узлов со значениями в этом регистре.

### Пример 5.24

Граф зависимости, представленный на рис. 5.43б, перерисован на рис. 5.46. На рисунке использован знак равенства для указания узлов, определяемых правилом копирования. Исходя из синтаксически управляемого определения на рис. 5.42, мы находим, что тип, определенный в узле 1, копируется в каждый элемент списка идентификаторов; в результате узлы 2, 3, 6 и 7 на рис. 5.43 являются копиями 1.

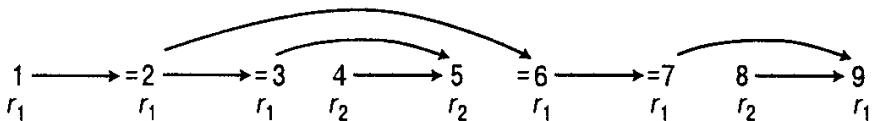


Рис. 5.46. Использование регистров с учетом правил копирования

Поскольку 2 и 3 являются копиями 1, их значения могут быть получены из регистра  $r_1$  на рис. 5.46. Обратите внимание, что время жизни 3 заканчивается при вычислении 5, но регистр  $r_1$ , хранящий значение 3, не возвращается в пул, поскольку еще не завершено время жизни 2 из класса эквивалентности.

Следующий код показывает, каким образом может быть обработано компилятором объявление (5.10) из примера 5.22.

```

r1 := integer; /* Вычисление узлов 1, 2, 3, 6, 7 */
r2 := 5; /* Вычисление узла 4 */
r2 := array(r2, r1); /* Тип у */
addtype(y, r2);

```

```
r2 := 3; /* Вычисление узла 8 */
r2 := array(r2, r1); /* Тип x */
addtype(x, r2);
```

Здесь  $x$  и  $y$  указывают на записи в таблице символов для  $x$  и  $y$ , а процедура *addtype* должна быть вызвана в соответствующий момент для добавления типов  $x$  и  $y$  в соответствующие записи таблицы символов.  $\square$

## 5.9. Назначение памяти в процессе создания компилятора

Хотя в процессе обхода можно хранить все значения атрибутов в одном стеке, иногда удается избежать копирования, если воспользоваться несколькими стеками. В целом, если зависимости между атрибутами делают размещение некоторых из них в стеке неудобным, мы можем хранить их в узлах явно построенного синтаксического дерева.

Мы уже видели в разделах 5.3 и 5.6, каким образом для хранения атрибутов в процессе восходящего синтаксического анализа применяется стек. Стек также неявно используется синтаксическим анализатором, работающим методом рекурсивного спуска, для слежения за вызовами процедур; эта тема будет рассмотрена в главе 7, “Среды времени исполнения”.

Для экономии памяти использование стека может сочетаться с другими технологиями. Один из методов экономии памяти — передача при вызове функций не больших объектов, а указателей на них (как это делалось, например, в разделе 5.2, когда в процессе построения синтаксических деревьев мы передавали указатели на узлы вместо целых поддеревьев). Эти технологии будут использованы в примерах 5.27 и 5.28.

### Вычисление времени жизни по грамматике

В случае, когда порядок вычисления атрибутов задается определенным обходом дерева разбора, мы можем предсказать время жизни атрибутов еще в процессе построения компилятора. Пусть, например, дочерние узлы посещаются слева направо в процессе обхода в глубину, как это было в разделе 5.4. Начиная обход с узла для продукции  $A \rightarrow BC$ , мы вначале посещаем поддерево для  $B$ , затем поддерево для  $C$  и после этого возвращаемся в узел для  $A$ . Родительский по отношению к  $A$  узел не может обращаться к атрибутам  $B$  и  $C$ , поэтому их время жизни завершается, когда мы возвращаемся в  $A$ . Заметим, что эти наблюдения основаны на продукции  $A \rightarrow BC$  и на порядке, в котором посещаются узлы для этих нетерминалов. Мы можем ничего не знать о поддеревьях  $B$  и  $C$ .

При любом порядке вычислений, если время жизни атрибута  $c$  является частью времени для  $b$ , то значение  $c$  может храниться в стеке над значением  $b$  (здесь  $b$  и  $c$  не обязаны быть атрибутами одного и того же нетерминала). Для продукции  $A \rightarrow BC$  в процессе обхода в глубину мы можем использовать стек следующим образом.

Начинаем в узле для  $A$  с наследуемыми атрибутами  $A$ , находящимися в стеке. Затем вычисляем и вносим в стек значения наследуемых атрибутов  $B$ . Эти атрибуты остаются в стеке при обходе поддерева  $B$ ; после обхода в стеке над наследуемыми атрибутами оказываются синтезируемые атрибуты. Затем тот же процесс повторяется с поддеревом  $C$ , т.е. в стек вносятся наследуемые атрибуты  $C$  и выполняется обход поддерева для  $C$ , после которого на вершине стека оказываются синтезируемые атрибуты  $C$ . Обозначая на-

следуемые и синтезируемые атрибуты  $X$  как  $I(X)$  и  $S(X)$  соответственно, можно записать содержимое стека как

$$I(A), I(B), S(B), I(C), S(C) \quad (5.11)$$

Все атрибуты, необходимые для вычисления синтезируемых атрибутов  $A$ , в настоящее время находятся в стеке, поэтому к  $A$  мы возвращаемся со стеком, содержащим  $I(A), S(A)$

Заметим, что количество (и, предположительно, размер) наследуемых и синтезируемых атрибутов символов грамматики зафиксировано. Следовательно, на каждом шаге описанного процесса мы знаем, в какой позиции стека находится интересующий нас атрибут.

### Пример 5.25

Предположим, что значения атрибутов в случае трансляции на рис. 5.22 хранятся в стеке так, как описано выше. Начнем обход в узле для продукции  $B \rightarrow B_1B_2$  с  $B.ps$ , находящемся на вершине стека. На рис. 5.47 показано содержимое стека до и после посещения узлов (соответственно слева и справа от узла). Как обычно, стек растет вниз.

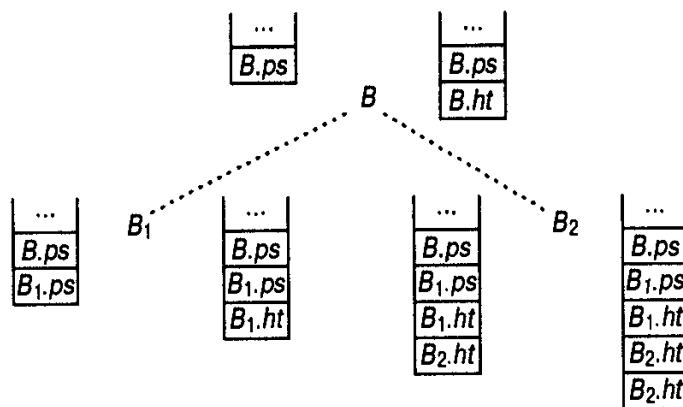


Рис. 5.47. Содержимое стека до и после посещения узлов

Обратите внимание на то, что непосредственно перед первым посещением узла для нетерминала  $B$  на вершине стека находится его атрибут  $ps$ , а сразу после завершения обхода поддерева, т.е. последнего посещения узла для  $B$ , верхние позиции стека заняты его атрибутами  $ht$  и  $ps$ .  $\square$

Когда атрибут  $b$  определяется правилом копирования  $b := c$ , а значение  $c$  находится на вершине стека значений атрибутов, возможно, удастся обойтись без размещения в стеке копии  $c$ . Возможностей для исключения правил копирования может оказаться больше, если для хранения значений атрибутов используется несколько стеков. В следующем примере мы используем отдельные стеки для наследуемых и синтезируемых атрибутов. Сравнение с примером 5.25 показывает, что при использовании отдельных стеков можно исключить большее количество правил копирования.

### Пример 5.26

Предположим, что при реализации синтаксически управляемого определения на рис. 5.22 мы используем отдельные стеки для наследуемого атрибута  $ps$  и синтезируемого  $ht$ . Мы работаем со стеками таким образом, чтобы значение  $B.ps$  находилось на вершине стека  $ps$  непосредственно перед первым и вслед за последним посещением узла для  $B$  (после чего  $B.ht$  оказывается на вершине стека  $ht$ ).

При использовании раздельных стеков мы можем обойтись без внесения в стек наследуемых атрибутов, определяемых правилами копирования  $B_1.ps := B.ps$  и  $B_2.ps := B.ps$ , поскольку, как видно из рис. 5.48, эти значения уже располагаются на вершине стека  $ps$ .

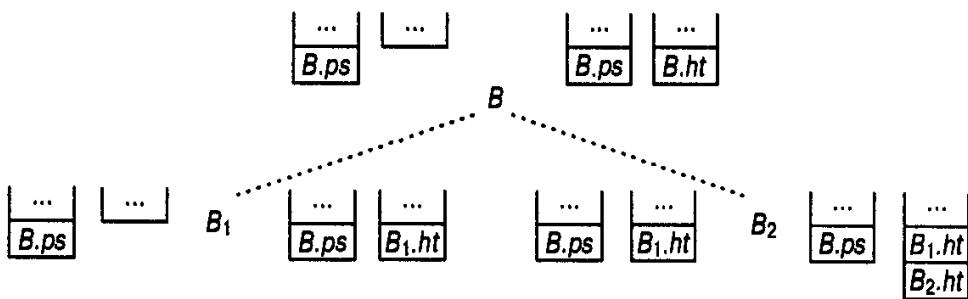


Рис. 5.48. Использование отдельных стеков для атрибутов  $ps$  и  $ht$

Схема трансляции, основанная на синтаксически управляемом определении (рис. 5.22), показана на рис. 5.49. Операция  $push(v, s)$  размещает значение  $v$  в стеке  $s$ , а  $pop(s)$  — снимает значение с вершины стека  $s$ . Для обращения к верхнему элементу стека  $s$  используется функция  $top(s)$ .  $\square$

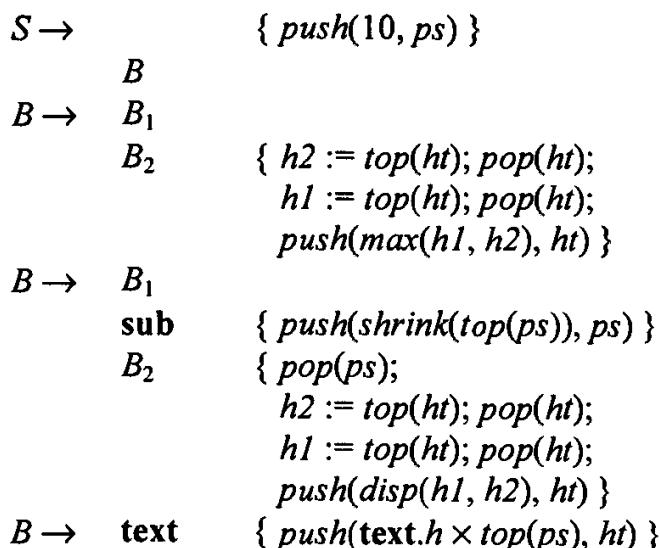


Рис. 5.49. Схема трансляции с двумя стеками —  $ps$  и  $ht$

Следующий пример сочетает использование стека для значений атрибутов с действиями по созданию кода.

### Пример 5.27

Здесь мы рассмотрим технологию реализации синтаксически управляемого определения, задающего генерацию промежуточного кода. Значение логического выражения  $E$  and  $F$  ложно, если ложно  $E$ . В языке программирования C подвыражение  $F$  в такой ситуации не должно вычисляться (такие логические выражения рассматриваются в разделе 8.4).

Логические выражения в синтаксически управляемом определении на рис. 5.50 строятся из идентификаторов и оператора `and`. Каждое выражение  $E$  наследует метки  $E.true$  и  $E.false$ , указывающие точки передачи управления в случаях, если  $E$  истинно или, соответственно, ложно.

Рассмотрим  $E \rightarrow E_1 \text{ and } E_2$ . Если при вычислении  $E_1$  ложно, то управление переходит к наследуемой метке  $E.false$ ; в противном случае оно передается коду вычисления  $E_2$ . Новая метка, созданная функцией `newlabel`, помечает начало кода вычисления  $E_2$ . От-

дельные инструкции кода генерируются вызовами функции *gen*. Более подробно о значимости рис. 5.50 для генерации промежуточного кода будет сказано в разделе 8.4.

| ПРОДУКЦИЯ                            | СЕМАНТИЧЕСКИЕ ПРАВИЛА                                                                                                                                                                                                                                                                                                                                                                       |
|--------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| $E \rightarrow E_1 \text{ and } E_2$ | $E_1.\text{true} := \text{newlabel}$<br>$E_1.\text{false} := E.\text{false}$<br>$E_2.\text{true} := E.\text{true}$<br>$E_2.\text{false} := E.\text{false}$<br>$E.\text{code} := E_1.\text{code} \parallel \text{gen('label' } E_1.\text{true}) \parallel E_2.\text{code}$<br>$E.\text{code} := \text{gen('if id.place 'goto' } E.\text{true}) \parallel \text{gen('goto' } E.\text{false})$ |
| $E \rightarrow \text{id}$            |                                                                                                                                                                                                                                                                                                                                                                                             |

Рис. 5.50. Сокращенные вычисления логических выражений

Синтаксически управляемое определение на рис. 5.50 является L-атрибутным, так что мы можем построить для него схему трансляции. Эта схема, приведенная на рис. 5.51, для генерации и инкрементного вывода инструкций в код использует процедуру *emit*. Кроме того, в схеме приведены действия для установки значений наследуемых атрибутов, вставленные, как обсуждалось в разделе 5.4, непосредственно перед соответствующими грамматическими символами.

$$\begin{array}{ll}
 E \rightarrow & \{ E_1.\text{true} := \text{newlabel}; \\
 & \quad E_1.\text{false} := E.\text{false} \} \\
 & \\
 E_1 & \\
 \text{and} & \{ \text{emit('label' } E_1.\text{true}); \\
 & \quad E_2.\text{true} := E.\text{true}; \\
 & \quad E_2.\text{false} := E.\text{false} \} \\
 & \\
 E_2 & \\
 E \rightarrow \text{id} & \{ \text{emit('if id.place 'goto' } E.\text{true}); \\
 & \quad \text{emit('goto' } E.\text{false}) \}
 \end{array}$$

Рис. 5.51. Генерация кода для логических выражений

Схема трансляции, приведённая на рис. 5.52, идет дальше; она использует отдельные стеки для хранения значений наследуемых атрибутов  $E.\text{true}$  и  $E.\text{false}$ . Как и в примере 5.26, правила копирования не воздействуют на стек. Для реализации правила  $E_1.\text{true} := \text{newlabel}$  новая метка вносится в стек *true* перед посещением  $E_1$ . Время жизни этой метки заканчивается с выполнением *emit('label' top(true))*, что соответствует *emit('label' } E\_1.\text{true})*, так что со стека *true* после выполнения действия снимается верхний элемент. Стек *false* в этом примере не изменяется, но он необходим при использовании сокращенных вычислений с оператором *or*.  $\square$

$$\begin{array}{ll}
 E \rightarrow & \{ \text{push(newlabel, true)} \} \\
 & \\
 E_1 & \\
 \text{and} & \{ \text{emit('label' top(true));} \\
 & \quad \text{pop(true)} \} \\
 & \\
 E_2 & \\
 E \rightarrow \text{id} & \{ \text{emit('if id.place 'goto' top(true));} \\
 & \quad \text{emit('goto' top(false))} \}
 \end{array}$$

Рис. 5.52. Генерация кода для логических выражений

## Неперекрывающиеся времена жизни

Отдельный регистр, по сути, является частным случаем стека. Если за каждой операцией внесения в стек *push* следует операция снятия со стека *pop*, то в стеке одновременно может находиться только один элемент. В этом случае вместо стека мы можем воспользоваться регистром. Таким образом, если времена жизни двух атрибутов не перекрываются, то их значения могут храниться в одном и том же регистре.

### Пример 5.28

Синтаксически управляемое определение на рис. 5.53 строит синтаксические деревья для выражений наподобие списков с операторами одного уровня приоритета. Эта схема получена из схемы трансляции на рис. 5.28.

| ПРОДУКЦИЯ                  | СЕМАНТИЧЕСКИЕ ПРАВИЛА                       |
|----------------------------|---------------------------------------------|
| $E \rightarrow TR$         | $R.i := T.nptr$                             |
|                            | $E.nptr := R.s$                             |
| $R \rightarrow addop TR_1$ | $R_1.i := mknoden(addop.lexeme, r, T.nptr)$ |
|                            | $R.s := R_1.s$                              |
| $R \rightarrow \epsilon$   | $R.s := R.i$                                |
| $T \rightarrow num$        | $T.nptr := mkleaf(num, num.val)$            |

Рис. 5.53. Синтаксически управляемое определение, полученное из рис. 5.28

Мы утверждаем, что время жизни каждого атрибута символа  $R$  заканчивается, когда вычисляется зависящий от него атрибут. Можно показать, что для любого дерева разбора атрибуты  $R$  могут быть вычислены в одном и том же регистре  $r$ . Следующее обоснование типично для анализа грамматик. Индукция ведется по размеру поддерева, присоединенного к  $R$  во фрагменте дерева разбора на рис. 5.54.

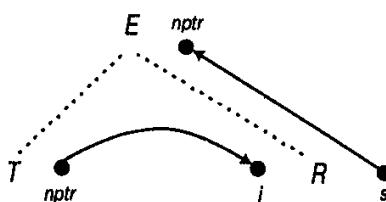


Рис. 5.54. Граф зависимости для  $E \rightarrow TR$

Наименьшее поддерево получается при использовании  $R \rightarrow \epsilon$ , когда  $R.s$  представляет собой копию  $R.i$ , так что оба их значения располагаются в регистре  $r$ . Для большего поддерева продукция в его корне должна быть  $R \rightarrow addop TR_1$ . Время жизни  $R.i$  заканчивается при вычислении  $R_1.i$ , поэтому значение  $R_1.i$  может быть помещено в регистр  $r$ . Согласно гипотезе индукции, все атрибуты экземпляров нетерминала  $R$  в поддереве для  $R_1$  могут быть назначены одному и тому же регистру. И наконец,  $R.s$  является копией  $R_1.s$ , так что его значение уже находится в регистре  $r$ .

Схема трансляции на рис. 5.55 вычисляет атрибуты в грамматике на рис. 5.53 с использованием единственного регистра  $r$  для хранения значений атрибутов  $R.i$  и  $R.s$  для всех экземпляров нетерминала  $R$ .

|                              |                                                            |
|------------------------------|------------------------------------------------------------|
| $E \rightarrow T$            | { $r := T.nptr$ /* В $r$ хранится $R.i$ */ }               |
| $R$                          | { $E.nptr := r$ /* $r$ возвратился со значением $R.s$ */ } |
| $R \rightarrow \text{addop}$ |                                                            |
| $T$                          | { $r := \text{mknode}(\text{addop}.lexeme, r, T.nptr)$ }   |
| $R$                          |                                                            |
| $R \rightarrow \epsilon$     |                                                            |
| $T \rightarrow \text{num}$   | { $T.nptr := \text{mkleaf}(\text{num}, \text{num}.val)$ }  |

Рис. 5.55. Преобразованная схема трансляции для построения синтаксических деревьев

Для полноты изложения на рис. 5.56 приведен код, реализующий данную схему трансляции; он построен в соответствии с алгоритмом 5.2. Нетерминал  $R$  больше не имеет атрибутов, так что  $R$  становится процедурой, а не функцией. Переменная  $r$  объявлена как локальная переменная функции  $E$ , поэтому  $E$  может вызываться рекурсивно, хотя схема на рис. 5.55 и не требует этого. Приведенный код можно усовершенствовать, устранив окончную рекурсию и затем заменив оставшийся вызов  $R$  телом полученной процедуры, как в разделе 2.5.  $\square$

```

function E: ↑syntax_tree_node;
 var r: ↑syntax_tree_node;
 addoplexeme: char;

procedure R;
begin
 if lookahead = addop then begin
 addoplexeme := lexval;
 match(addop);
 r := mknode(addoplexeme, r, T);
 end
end;
begin
 r := T; R;
 return r
end;

```

Рис. 5.56. Сравните процедуру  $R$  с кодом на рис. 5.31

## 5.10. Анализ синтаксически управляемых определений

В разделе 5.7 атрибуты вычислялись в процессе обхода дерева с использованием множества взаимно рекурсивных функций. Функция для нетерминала отображала значения наследуемых атрибутов узла в значения синтезируемых атрибутов.

Подход из раздела 5.7 расширяется для трансляций, которые не могут быть выполнены в процессе однократного обхода в глубину. Здесь мы будем использовать отдельные функции для каждого синтезируемого атрибута каждого нетерминала, хотя одной функцией могут вычисляться группы синтезируемых атрибутов. Группирование атрибутов определяется из зависимостей, установленных семантическими правилами в синтаксически управляемом определении. Следующий абстрактный пример иллюстрирует построение рекурсивного вычислителя.

### Пример 5.29

Синтаксически управляемое определение на рис. 5.57 вызвано проблемой, которую мы рассмотрим в главе 6, “Проверка типов”. Вкратце эта проблема состоит в следующем. “Перегруженный” идентификатор может иметь множество возможных типов. В результате выражение тоже может иметь множество возможных типов. Чтобы выбрать один из возможных типов для каждого подвыражения, используется контекстная информация. Проблема может быть решена выполнением восходящего прохода для создания множества всех возможных типов с последующим нисходящим проходом для сокращения множества до одного типа.

Семантические правила на рис. 5.57 абстрагируют данную проблему. Синтезируемый атрибут  $s$  представляет множество возможных типов, а наследуемый атрибут  $i$  — контекстную информацию. Дополнительный синтезируемый атрибут  $t$ , который не может быть вычислен в том же проходе, что и  $s$ , мог бы представлять генерируемый код или тип, выбранный для подвыражения. Графы зависимости для продукции на рис. 5.57 показаны на рис. 5.58.  $\square$

| ПРОДУКЦИЯ               | СЕМАНТИЧЕСКИЕ ПРАВИЛА                                                                                |
|-------------------------|------------------------------------------------------------------------------------------------------|
| $S \rightarrow E$       | $E.i := g(E.s)$<br>$S.r := E.t$                                                                      |
| $E \rightarrow E_1 E_2$ | $E.s := fs(E_1.s, E_2.s)$<br>$E_1.i := fi1(E.i)$<br>$E_2.i := fi2(E.i)$<br>$E.t := ft(E_1.t, E_2.t)$ |
| $E \rightarrow id$      | $E.s := id.s$<br>$E.t := h(E.i)$                                                                     |

Рис. 5.57. Синтезируемые атрибуты  $s$  и  $t$  не могут вычисляться вместе

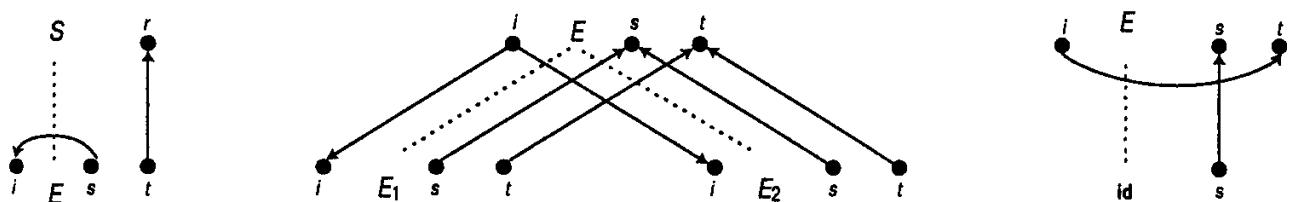


Рис. 5.58. Графы зависимости для продукции на рис. 5.57

### Рекурсивное вычисление атрибутов

Граф зависимости для дерева разбора формируется путем объединения меньших графов, соответствующих семантическим правилам продукции. Граф зависимости  $D_p$  для продукции  $p$  основывается только на семантических правилах для одной продукции, т.е. на семантических правилах для синтезируемых атрибутов левой части и наследуемых атрибутов грамматических символов правой части продукции. Таким образом, график  $D_p$  показывает только локальные зависимости. Например, на рис. 5.58 все дуги в графике зависимости для  $E \rightarrow E_1 E_2$  являются дугами между экземплярами одного и того же атрибута. Исходя из этого графа зависимости, мы не можем сказать, что атрибуты  $s$  должны быть вычислены раньше всех остальных атрибутов.

При внимательном рассмотрении графа зависимости для дерева разбора, приведенного на рис. 5.59, можно увидеть, что атрибуты каждого экземпляра нетерминала  $E$  должны вычисляться в следующем порядке:  $E.s$ ,  $E.i$ ,  $E.t$ . Заметим, что все атрибуты на рис. 5.59 могут быть вычислены в три прохода: восходящий проход для вычисления атрибутов  $s$ , нисходящий — для вычисления атрибутов  $i$  и заключительный восходящий — для вычисления атрибутов  $t$ .

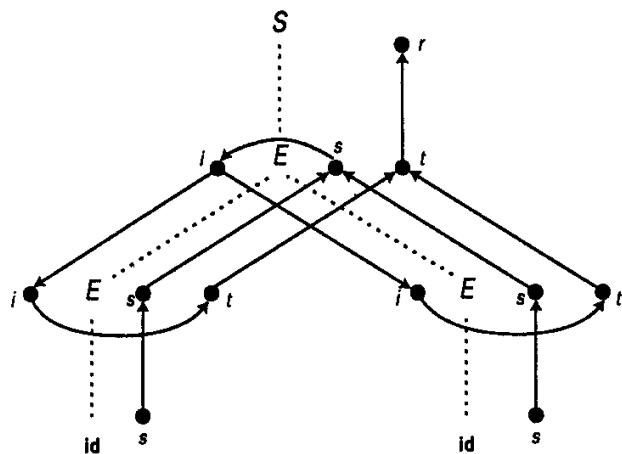


Рис. 5.59. Граф зависимости для дерева разбора

В рекурсивном вычислителе функция для синтезируемого атрибута получает в качестве параметров значения некоторых наследуемых атрибутов. В целом, если синтезируемый атрибут  $A.a$  может зависеть от наследуемого атрибута  $A.b$ , то функция для  $A.a$  получает  $A.b$  в качестве параметра. Перед анализом зависимостей мы рассмотрим пример, показывающий их использование.

### Пример 5.30

Функции  $Es$  и  $Et$  на рис. 5.60 возвращают значения синтезируемых атрибутов  $s$  и  $t$  в узле  $n$ , помеченном как  $E$ . Как и в разделе 5.7, функция для нетерминала содержит вариант для каждой продукции. Код, выполняемый в каждом случае, моделирует семантические правила, связанные с продукциями на рис. 5.57.

Из предыдущего обсуждения графа зависимости на рис. 5.59 нам известно, что атрибут  $E.t$  в узле дерева разбора может зависеть от  $E.i$ . Таким образом, мы передаем наследуемый атрибут  $i$  в качестве параметра в функцию  $Et$  для атрибута  $t$ . Поскольку атрибут  $E.s$  не зависит ни от одного наследуемого атрибута, функция  $Es$  не имеет параметров, соответствующих значениям атрибутов.  $\square$

```

function Es(n);
begin
 case Продукция в узле n of
 'E → E1 E2':
 s1 := Es(child(n, 1));
 s2 := Es(child(n, 2));
 return fs(s1, s2);
 'E → id':
 return id.s;
 default:
 error
 endcase
endfunction

```

```

 end
end;
function Et(n, i);
begin
 case Продукция в узле n of
 'E → E1 E2':
 i1 := f1(i);
 t1 := Et(child(n, 1), i1);
 i2 := f2(i);
 t2 := Et(child(n, 2), i2);
 return ft(t1, t2);
 'E → id':
 return h(i);
 default:
 error
 end
end;
function Sr(n);
begin
 s := Es(child(n, 1));
 i := g(s);
 t := Et(child(n, 1), i);
 return t
end;

```

• Рис. 5.60. Функции для синтезируемых атрибутов на рис. 5.57

## Строго нециклически управляемые определения

Рекурсивные вычислители могут быть построены для класса синтаксически управляемых определений, называемых “строго нециклическими” (strongly noncircular). У определения из такого класса атрибуты в каждом узле для нетерминала могут быть вычислены в некотором (частичном) порядке. При построении функции для синтезируемых атрибутов нетерминала этот порядок используется для выбора наследуемых атрибутов, которые становятся параметрами функции.

Мы дадим определение такого класса и покажем, что синтаксически управляемое определение на рис. 5.57 попадает в него. Затем мы приведем алгоритм для проверки цикличности и строгой нецикличности и покажем, как реализация примера 5.30 распространяется на все строго нециклические определения.

Рассмотрим нетерминал  $A$  в узле  $n$  дерева разбора. Граф зависимости для дерева разбора, вообще говоря, может иметь пути, которые начинаются с атрибута в узле  $n$ , проходят через атрибуты в других узлах дерева разбора и заканчиваются в другом атрибуте  $n$ . Для наших целей достаточно взглянуть на пути, которые располагаются в части дерева разбора ниже  $A$ . Небольшое размыщление приводит к выводу, что такие пути проходят из некоторого наследуемого атрибута  $A$  в некоторый синтезируемый атрибут  $A$ . Мы дадим оценку (возможно, слишком пессимистичную) множеству таких путей, рассматривая частичные порядки атрибутов  $A$ .

Пусть продукция  $p$  имеет нетерминалы  $A_1, A_2, \dots, A_n$ , встречающиеся в правой части. Пусть  $RA_j$  — частичный порядок атрибутов  $A_j$ ,  $1 \leq j \leq n$ . Обозначим через  $D_p[RA_1, RA_2, \dots, RA_n]$  граф, полученный добавлением дуг к  $D_p$  следующим образом: если  $RA_j$  располагает атрибут  $A_j.b$  перед  $A_j.c$ , то добавим дугу от  $A_j.b$  к  $A_j.c$ .

Синтаксически управляемое определение называется *строго нециклическим*, если для каждого нетерминала  $A$  мы можем найти частичный порядок  $RA$  на атрибутах  $A$ , такой, что для каждой продукции  $p$  с левой частью  $A$  и нетерминалами  $A_1, A_2, \dots, A_n$  в правой части

1.  $D_p[RA_1, RA_2, \dots, RA_n]$  ацикличен;
2. если в  $D_p[RA_1, RA_2, \dots, RA_n]$  существует дуга от атрибута  $A.b$  к  $A.c$ , то  $RA$  располагает  $A.b$  перед  $A.c$ .

### Пример 5.31

Пусть  $p$  — продукция  $E \rightarrow E_1 E_2$  (рис. 5.57), график зависимости  $D_p$  которой показан в центре рис. 5.58. Пусть  $RE$  — частичный порядок (в данном случае — полный порядок)  $s \rightarrow i \rightarrow t$ . В правой части  $p$  имеется два нетерминала —  $E_1$  и  $E_2$ . Следовательно,  $RE_1$  и  $RE_2$  аналогичны  $RE$ , и график  $D_p[RE_1, RE_2]$  выглядит так, как показано на рис. 5.61.

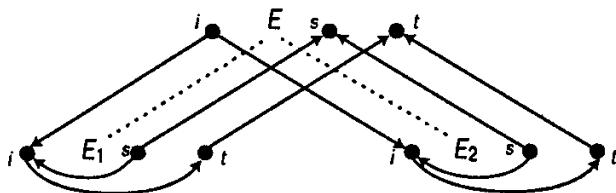


Рис. 5.61. Расширенный график зависимости для продукции

Среди атрибутов, связанных с корнем  $E$  (рис. 5.61), имеются только пути от  $i$  до  $t$ . Поскольку  $RE$  делает  $i$  предшествующим  $t$ , нарушений условия (2) нет.  $\square$

Если заданы строго нециклическое определение и частичный порядок  $RA$  для каждого нетерминала  $A$ , функция для синтезируемого атрибута  $s$  нетерминала  $A$  получает аргументы следующим образом. Если  $RA$  располагает наследуемый атрибут  $i$  перед  $s$ , то  $i$  является аргументом функции; в противном случае — нет.

### Проверка цикличности

Синтаксически управляемое определение называется циклическим, если график зависимости для некоторого дерева разбора имеет цикл; циклические определения являются некорректными и бессмысленными. Невозможно начать вычисление любого из значений атрибутов в цикле. Вычисление частных порядков, гарантирующее, что определение является строго нециклическим, тесно связано с проверкой цикличности определения. Поэтому вначале мы рассмотрим проверку цикличности.

### Пример 5.32

В следующем синтаксически управляемом определении пути между атрибутами  $A$  зависят от применяемой продукции. При применении  $A \rightarrow 1$  атрибут  $A.s$  зависит от  $A.i$ ; в противном случае этой зависимости нет. Для получения полной информации о возможных зависимостях мы, таким образом, должны отслеживать множества частичных порядков атрибутов нетерминала.  $\square$

| ПРОДУКЦИЯ         | СЕМАНТИЧЕСКИЕ ПРАВИЛА |
|-------------------|-----------------------|
| $S \rightarrow A$ | $A.i := c$            |
| $A \rightarrow 1$ | $A.s := f(A.i)$       |
| $A \rightarrow 2$ | $A.s := d$            |

Идея, стоящая за алгоритмом на рис. 5.62, следующая. Мы представляем частичные порядки в виде направленных ациклических графов (дагов). По дагам на атрибутах символов в правой части продукции мы можем определить даг для атрибутов левой части следующим образом.

```

for Грамматический символ X do
 $\mathfrak{F}(X)$ имеет единственный граф с атрибутами X и без дуг;
repeat
 $change := \text{false};$
 for Продукция p , заданная $A \rightarrow X_1X_2\dots X_k$ do begin
 for Даги $G_1 \in \mathfrak{F}(X_1), \dots, G_k \in \mathfrak{F}(X_k)$ do begin
 $D := D_p;$
 for Дуга $b \rightarrow c$ в $G_j, 1 \leq j \leq k$ do
 Добавить в D дугу между
 атрибутами b и c символа X_j ;
 if D имеет цикл then
 Тест на цикличность не пройден
 else begin
 $G :=$ новый граф с узлами для атрибутов
 A и без дуг;
 for Каждая пара атрибутов b и c
 нетерминала A do
 if Существует путь в D от b к c then
 Добавить $b \rightarrow c$ в G ;
 if G еще не находится в $\mathfrak{F}(A)$ then begin
 Добавить G к $\mathfrak{F}(A)$;
 $change := \text{true}$
 end
 end
 end
 end
 until $change = \text{false}$

```

Рис. 5.62. Проверка цикличности

Пусть продукция  $p — A \rightarrow X_1X_2\dots X_k$  с графом зависимости  $D_p$ . Пусть  $D_j$  — даг для  $X_j, 1 \leq j \leq k$ . Каждая дуга  $b \rightarrow a$  в  $D_j$  временно добавляется в граф зависимости  $D_p$  продукции. Если полученный граф имеет цикл, то синтаксически управляемое определение циклическо. В противном случае пути в результирующем графе определяют новый даг на атрибутах из левой части продукции, и полученный даг добавляется к  $\mathfrak{F}(A)$ .

Проверка циклическо на рис. 5.62 обладает экспоненциальной зависимостью времени работы от числа графов в множестве  $\mathfrak{F}(X)$  для любого грамматического символа  $X$ .

Существуют синтаксически управляемые определения, которые не могут быть проверены на цикличность за полиномиальное время.

Если синтаксически управляемое определение строго нециклическо, приведенный на рис. 5.62 алгоритм можно преобразовать в более эффективную проверку следующим образом. Вместо семейства графов  $\mathfrak{F}(X)$  для каждого  $X$  мы собираем всю информацию с помощью одного графа  $F(X)$ . Заметьте, что все графы в  $\mathfrak{F}(X)$  имеют одни и те же узлы для атрибутов  $X$ , но могут иметь различные дуги.  $F(X)$  представляет собой граф из узлов для атрибутов  $X$ , имеющий дугу между  $X.b$  и  $X.c$ , если ее имеет какой-нибудь граф из  $\mathfrak{F}(X)$ .  $F(X)$  представляет “оценку худшего случая” зависимостей между атрибутами  $X$ . В частности, если  $F(X)$  ациклический, то синтаксически управляемое определение гарантированно нециклическо (однако обратное неверно: если  $F(X)$  имеет цикл, то это еще не значит, что синтаксически управляемое определение циклическо).

Модифицированная проверка циклическости строит ациклические графы  $F(X)$  для каждого  $X$ , если это удается. На основе этих графов мы можем построить вычислитель для синтаксически управляемого определения. Этот метод является прямым обобщением примера 5.30. Функция для синтезируемого атрибута  $X.s$  получает в качестве аргументов все наследуемые атрибуты, которые предшествуют  $s$  в  $F(X)$ , и только их. Функция, вызываемая в узле  $i$ , вызывает другие функции, которые вычисляют все необходимые синтезируемые атрибуты в дочерних по отношению к  $i$  узлах. Для вычисления этих атрибутов подпрограммам передаются значения необходимых наследуемых атрибутов. Успешная проверка строгой нециклическости гарантирует, что эти наследуемые атрибуты могут быть вычислены.

## Упражнения

- 5.1. Постройте аннотированное дерево разбора, соответствующее синтаксически управляемому определению на рис. 5.2, для входной строки  $(4 * 7 + 1) * 2$ .
- 5.2. Постройте дерево разбора и синтаксическое дерево для выражения  $((a) + (b))$  согласно
  - a) синтаксически управляемому определению на рис. 5.9;
  - b) схеме трансляции на рис. 5.28.
- 5.3. Постройте даг и укажите номера значений подвыражений следующего выражения, полагая операцию + левоассоциативной:  $a + a + (a + a + a + (a + a + a + a))$ .
- \*5.4. Приведите синтаксически управляемое определение для трансляции инфиксных выражений в инфиксные выражения без лишних скобок. Например, поскольку + и \* левоассоциативны,  $((a * (b + c)) * (d))$  можно переписать как  $a * (b + c) * d$ .
- 5.5. Приведите синтаксически управляемое определение для дифференцирования выражений, построенных с применением арифметических операторов + и \* к переменной  $x$  и константам, например  $x * (3 * x + x * x)$ . Считаем, что никакие упрощения не делаются, т.е.  $3 * x$  транслируется в  $3 * 1 + 0 * x$ .
- 5.6. Следующая грамматика порождает выражения, формируемые применением арифметического оператора + к целым и действительным константам. При сложении двух целых тип результата — целый, в противном случае — действительный.

$$\begin{array}{lcl} E & \rightarrow & E + T \mid T \\ T & \rightarrow & \text{num . num} \mid \text{num} \end{array}$$

- a) Дайте синтаксически управляемое определение для установления типа каждого подвыражения.
- b) Расширьте синтаксически управляемое определение (а) как для установления типов, так и трансляции выражений в постфиксную форму. Используйте унарный оператор `inttoreal` для преобразования целого значения в эквивалентное действительное так, чтобы оба операнда + в постфиксной форме имели один и тот же тип.
- 5.7. Расширьте синтаксически управляемое определение на рис. 5.22 для отслеживания ширины блоков в дополнение к отслеживанию их высоты. Полагаем, что терминал `text` имеет синтезируемый атрибут `w`, дающий нормализованную ширину текста.
- 5.8. Пусть синтезируемый атрибут `val` представляет значение двоичного числа, порождаемого `S` в следующей грамматике. Например, при входной строке `101.101`  $S.val = 5.625$ .
- $$\begin{array}{l} S \rightarrow L \cdot L | L \\ L \rightarrow L B | B \\ B \rightarrow 0 | 1 \end{array}$$
- a) Для определения `S.val` воспользуйтесь синтезируемыми атрибутами.
- b) Определите `S.val` с помощью синтаксически управляемого определения, в котором используется единственный синтезируемый атрибут `B.c`, представляющий вклад бита, порожденного `B`, в конечное значение. Например, вклады первого и последнего битов числа `101.101` в значение `5.625` составляют `4` и `0.125` соответственно.
- 5.9. Перепишите грамматику, лежащую в основе синтаксически управляемого определения из примера 5.3, так, чтобы информация о типе могла распространяться с использованием только синтезируемых атрибутов.
- \*5.10. Когда инструкции, порождаемые следующей грамматикой, транслируются в абстрактный машинный код, инструкция `break` транслируется в переход к инструкции, следующей за ближайшей окружающей инструкцией `while`. Для простоты выражения представлены терминалом `expr`, а прочие типы инструкций — терминалом `other`. Эти терминалы имеют синтезируемый атрибут `code`, представляющий их трансляцию.

$$\begin{array}{l} S \rightarrow \text{while } \text{expr} \text{ do begin } S \text{ end} \\ | S ; S \\ | \text{break} \\ | \text{other} \end{array}$$

Приведите синтаксически управляемое определение, транслирующее инструкции в код для стековой машины из раздела 2.8. Убедитесь, что инструкция `break` корректно транслируется во вложенных инструкциях `while`.

- 5.11. Устраните левую рекурсию из синтаксически управляемых определений в упр. 5.6.
- 5.12. Выражения, порождаемые следующей грамматикой, могут содержать операции присвоения.

$$\begin{array}{l} S \rightarrow E \\ E \rightarrow E := E | E + E | ( E ) | \text{id} \end{array}$$

Семантика выражений такая же, как и в языке программирования С, т.е.  $b := c$  — выражение, присваивающее значение с переменной  $b$ ;  $r$ -значение этого выражения то же, что и значение  $c$ . Выражение  $a := (b := c)$  присваивает значение  $c$  переменной  $b$ , а затем —  $a$ .

- a) Постройте синтаксически управляемое определение для проверки того, что левая часть выражения представляет собой  $l$ -значение. Воспользуйтесь наследуемым атрибутом *side* нетерминала  $E$  для указания того, находится ли выражение, порождаемое  $E$ , в левой или правой части присвоения.
  - b) Расширьте синтаксически ориентированное определение из (a), чтобы генерировать промежуточный код для стековой машины из раздела 2.8 при проверке входного потока.
- 5.13.** Перепишите лежащую в основе предыдущего упражнения грамматику так, чтобы она группировала подвыражения  $:=$  вправо, а подвыражения  $+$  — влево.
- a) Постройте схему трансляции, которая моделирует синтаксически управляемое определение из упр. 5.12b.
  - b) Измените схему трансляции из (a) для инкрементального вывода кода в выходной файл.
- 5.14.** Приведите схему трансляции для проверки того, что один и тот же идентификатор не появляется в списке дважды.
- 5.15.** Предположим, что объявления порождаются следующей грамматикой.
- $$\begin{array}{lcl} D & \rightarrow & \text{id } L \\ L & \rightarrow & , \text{id } L \mid : T \\ T & \rightarrow & \text{integer} \mid \text{real} \end{array}$$
- a) Постройте схему трансляции для занесения типа каждого идентификатора в таблицу символов, как в примере 5.3.
  - b) Постройте предиктивный транслятор на основе схемы трансляции из (a).
- 5.16.** Приведенная грамматика представляет собой однозначную версию грамматики, лежащей в основе синтаксически управляемого определения на рис. 5.22. Фигурные скобки  $\{ \}$  используются только для группирования блоков и удаляются при трансляции.
- $$\begin{array}{lcl} S & \rightarrow & L \\ L & \rightarrow & LB \mid B \\ B & \rightarrow & B \text{ sub } F \mid F \\ F & \rightarrow & \{ L \} \mid \text{text} \end{array}$$
- a) Адаптируйте синтаксически управляемое определение на рис. 5.22 для использования приведенной здесь грамматики.
  - b) Преобразуйте синтаксически управляемое определение из (a) в схему трансляции.
- \*5.17.** Расширьте преобразование для устранения левой рекурсии в разделе 5.5 так, чтобы для нетерминала  $A$  из (5.2) допустить
- a) наследуемые атрибуты, определяемые правилами копирования;
  - b) наследуемые атрибуты.
- 5.18.** Устраним левую рекурсию из схемы трансляции в упр. 5.16b.

- 5.19. Предположим, у нас есть L-атрибутное определение, в основе которого лежит либо LL(1)-грамматика, либо грамматика, у которой мы можем разрешить неоднозначности и построить предиктивный синтаксический анализатор. Покажите, что мы можем содержать наследуемые и синтезируемые атрибуты в стеке нисходящего синтаксического анализатора, управляемого посредством таблицы предиктивного синтаксического анализа.
- 5.20. Докажите, что добавление различных маркеров-нетерминалов в произвольных местах продукции LL(1)-грамматики приводит к LR(1)-грамматике.
- 5.21. Рассмотрим следующую модификацию LR(1)-грамматики  $L \rightarrow L b | a$ .
- $$\begin{array}{l} L \rightarrow M L b | a \\ M \rightarrow \epsilon \end{array}$$
- a) В каком порядке нисходящий синтаксический анализатор будет применять продукции в дереве разбора для входной строки  $abbb$ ?
- \*b) Покажите, что модифицированная грамматика не является LR(1)-грамматикой.
- \*5.22. Покажите, что в схеме трансляции, основанной на рис. 5.36, значение наследуемого атрибута  $B.ps$  всегда располагается непосредственно под правой частью при свертке последней к  $B$ .
- 5.23. Алгоритм 5.3 для восходящего синтаксического анализа и трансляции с наследуемыми атрибутами использует маркеры-нетерминалы для хранения значений наследуемых атрибутов в “предсказуемых” позициях стека. Может потребоваться меньшее количество маркеров, если значения размещаются в отдельном стеке.
- a) Преобразуйте синтаксически управляемое определение на рис. 5.36 в схему трансляции.
- b) Модифицируйте схему трансляции, построенную в (a), так, чтобы значения наследуемого атрибута  $ps$  размещались в отдельном стеке. Удалите при этом из схемы маркер-нетерминал  $M$ .
- \*5.24. Рассмотрим трансляцию в процессе синтаксического анализа, как в упр. 5.23. С. Джонсон предложил метод моделирования отдельного стека для наследуемых атрибутов с использованием маркеров и глобальной переменной для каждого наследуемого атрибута. В следующей продукции значение  $v$  вносится в стек  $i$  посредством первого действия, а снимается вторым:  $A \rightarrow \alpha \{push(v, i)\} \beta \{pop(i)\}$ . Стек  $i$  может быть смоделирован следующими продукциями с использованием глобальной переменной  $g$  и маркера-нетерминала  $M$  с синтезируемым атрибутом  $s$ :
- $$\begin{array}{l} A \rightarrow \alpha M \beta \{ g := M.s \} \\ M \rightarrow \epsilon \{ M.s := g; g := v \} \end{array}$$
- a) Примените это преобразование к схеме трансляции из упр. 5.23b. Замените все ссылки на вершину отдельного стека ссылками на глобальную переменную.
- b) Покажите, что схема трансляции, построенная в (a), вычисляет те же значения синтезируемого атрибута стартового символа, что и в упр. 5.23b.
- 5.25. Используйте подход из раздела 5.8 для реализации всех атрибутов  $E.side$  схемы трансляции из упр. 5.12b единственной логической переменной.
- 5.26. Модифицируйте использование стека в процессе обхода в глубину в примере 5.26 таким образом, чтобы значения в стеке соответствовали значениям, содержащимся в стеке синтаксического анализатора в примере 5.19.

## Библиографические примечания

Использование синтезируемых атрибутов для определения трансляции языка впервые рассматривалось в работе [206]. Идея вызова синтаксического анализатора для семантических действий обсуждалась в работах [388] и [64]. Наследуемые атрибуты, графы за висимости и проверка строгой нецикличности появились в работе Кнута [254] (причем проверка нецикличности приведена в исправлениях к статье). В расширенном примере статьи использовались контролируемые побочные эффекты глобальных переменных связанных с корнем дерева разбора. Если атрибуты могут быть функциями, наследуемые атрибуты можно избежать [303].

Одно из приложений, где побочные эффекты в семантических правилах нежелательны, — синтаксически управляемое редактирование. Предположим, что редактор генерируется по атрибутной грамматике для исходного языка, как в [367], и рассмотрим редактирование, изменяющее исходную программу так, что в результате удаляется часть дерева разбора программы. До тех пор, пока нет побочных эффектов, значения атрибутов для измененной программы могут быть инкрементально перевычислены.

Для отслеживания общих подвыражений Ершов [123] использовал хеширование.

Определение L-атрибутных грамматик в работе [287] вызвано трансляцией в процессе синтаксического анализа. Аналогичные ограничения на зависимости атрибутов применяются в [58] к каждому из обходов в глубину слева направо. Аффиксные грамматики [268] тесно связаны с L-атрибутными. Ограничения L-атрибутных грамматик для управления доступом к глобальным атрибутам предложены в [267].

Механическое построение предиктивного транслятора, подобного построенному в соответствии с алгоритмом 5.2, описано в [59]. Ошибочность мнения, что нисходящий синтаксический анализ обеспечивает большую гибкость трансляции, была показана в [66], где доказано, что схема трансляции, основанная на LL(1)-грамматике, может быть моделирована в процессе LR(1)-анализа. Независимо в работе [445] использовались маркеры-нетерминалы для того, чтобы обеспечить размещение в стеке значений наследуемых атрибутов во время восходящего разбора. Позиции в правых частях продукции, в которых можно использовать маркеры-нетерминалы без потери свойства LR(1), рассмотрены в работе [356] (см. упр. 5.21). Само по себе требование, чтобы наследуемые атрибуты определялись правилами копирования, не гарантирует, что атрибуты могут быть вычислены в процессе восходящего синтаксического анализа; достаточные условия для семантических правил даны в [423]. В [225] в терминах состояний синтаксического анализатора дана характеристика атрибутов, которые могут быть вычислены в процессе LR(1)-анализа. В качестве примера трансляции, которая не может быть выполнена в процессе синтаксического анализа, в [162] рассмотрена генерация кода для логических выражений. В разделе 8.6 мы увидим, что для решения этой проблемы можно использовать технологию обратных поправок, так что полный второй проход не является необходимым.

Разработано множество инструментов для реализации синтаксически управляемых определений, начиная с FOLDS [127], но лишь немногие из них получили широкое распространение. DELTA [290] строит граф зависимости во время компиляции, а также экономит память, отслеживая времена жизни атрибутов и устранивая правила копирования. Методы вычисления атрибутов, основанные на дереве разбора, обсуждаются в работах [92, 242].

Обзор методов вычисления атрибутов дан в [122], а в работе [98] приведен обзор соответствующих теоретических основ. НLP, описанный в [362], выполняет чередующиеся обходы в глубину, предложенные в [213]. LINGUIST [128] также выполняет чередующиеся обходы. MUG [156] определяет порядок обхода потомков узла на основе продукции в узле. GAG [235] допускает повторные посещения потомков узла, реализуя класс упорядоченных атрибутных грамматик, определенный в [234]. Идея повторных посещений появилась ранее в [243], где был построен вычислитель для большого класса строго нециклических грамматик. В [387] описана модификация этого метода, позволяющая экономить память путем содержания значений атрибутов в стеке, если они не нужны при последующих посещениях. В [227] описано построение рекурсивных вычислителей для этого класса. Рекурсивные вычислители также создавались в [237]. Совершенно иной подход был применен в NEATS [299], где даг строился для выражений, представляющих значения атрибутов.

Анализ зависимостей при построении компилятора может сэкономить время и память в процессе компиляции. Проверка на цикличность является типичной проблемой анализа. В [211] доказано, что проверка цикличности требует времени, выраженного экспоненциальной зависимостью от размера грамматики. Некоторые технологии усовершенствования проверки цикличности рассмотрены в [111, 292, 361].

Размер памяти, используемой простыми вычислителями, ведет к разработке технологий ее экономии. Алгоритм назначения атрибутам регистров из раздела 5.8 был описан (в совершенно ином контексте) в [300]. В [398] показано, что проблема нахождения топологической сортировки графа зависимости, минимизирующей количество используемых регистров, является NP-полной. Анализ времени жизни во время компиляции в многопроходном вычислителе был рассмотрен в [212, 360]. В [62] упоминалось об использовании отдельных стеков для хранения синтезируемых и наследуемых атрибутов в процессе обхода. GAG выполняет анализ времени жизни и при необходимости размещает значения атрибутов в глобальных переменных, стеках и узлах дерева разбора. Сравнение технологий экономии памяти, используемых GAG и LINGUIST, проведено в [129].

# ГЛАВА 6

## Проверка типов

Компилятор должен убедиться, что исходная программа следует как синтаксическим, так и семантическим соглашениям исходного языка. Такая проверка, именуемая *статической* (static checking), — в отличие от *динамической*, выполняемой в процессе работы целевой программы, — гарантирует, что будут выявлены определенные типы программных ошибок. Ниже представлены примеры статических проверок.

1. *Проверки типов.* Компилятор должен сообщать об ошибке, если оператор применяется к несовместимому с ним операнду, например при сложении переменных, представляющих собой массив и функцию.
2. *Проверки управления.* Передача управления за пределы языковых конструкций должна производиться в определенное место. Например, в языке программирования С оператор `break` передает управление за пределы наиболее вложенной инструкции `while`, `for` или `switch`; если же таковые отсутствуют, то выводится сообщение об ошибке.
3. *Проверки единственности.* Существуют ситуации, когда объект может быть определен только один раз. Например, в языке программирования Pascal идентификатор должен объявляться только один раз, все метки в конструкции `case` должны быть различны, а элементы в скалярном типе не должны повторяться.
4. *Проверки, связанные с именами.* Иногда одно и то же имя должно использоваться дважды (или большее число раз). Например, в языке программирования Ada цикл или блок может иметь имя, которое должно находиться как в начале, так и в конце конструкции. Компилятор должен проверить, что в обоих местах используется одно и то же имя.

В этой главе нас, в первую очередь, интересует проверка типов. Как видно из приведенных примеров, большинство прочих статических проверок являются рутинными и могут быть реализованы с использованием технологий, описанных в предыдущей главе. Некоторые из них могут использоваться и для выполнения других действий. Например, при внесении информации об имени в таблицу символов мы можем убедиться в единственности объявления данного имени. Многие компиляторы Pascal объединяют статические проверки и генерацию промежуточного кода с синтаксическим анализом. При наличии более сложных конструкций, наподобие используемых в языке программирования Ada, может оказаться более удобным выполнить отдельный проход для проведения проверок типов между синтаксическим анализом и генерацией промежуточного кода, как показано на рис. 6.1.

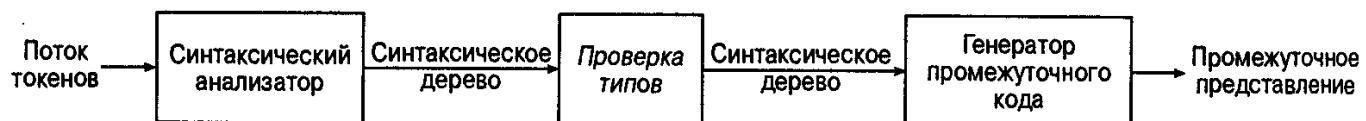


Рис. 6.1. Положение программы проверки типов

Программа проверки типов проверяет, чтобы тип конструкции соответствовал ожидаемому в данном контексте. Например, встроенный арифметический оператор `mod` в Pascal требует целых operandов, поэтому программа проверки типов должна проверить, что operandы `mod` в исходной программе — целого типа. Точно так же программа проверки типов должна убедиться, что операция разыменования применяется к указателю, индексирование выполняется с массивом, что определенная пользователем функция вызывается с корректным числом аргументов верного типа и т.д. Спецификацию простой программы проверки типов вы найдете в разделе 6.2. Представление типов и вопрос их соответствия рассматриваются в разделе 6.3.

Информация о типах, собираемая программой проверки типов, может потребоваться при генерации кода. Например, обычно арифметические операторы типа `+` применяются к целым и действительным числам, а возможно, и к другим типам данных, так что для определения смысла оператора `+` требуется рассмотрение контекста его применения. Символ, который может представлять различные операции в разных контекстах, называется “перегруженным” (overloaded). Перегрузка может сопровождаться принудительным преобразованием типов operandов в ожидаемые в данном контексте, которое выполняется компилятором.

Другое понятие, отличное от понятия перегрузки, — полиморфизм. Тело полиморфной функции может выполняться с аргументами различных типов. Алгоритм унификации, обеспечивающий определение типов полиморфных функций, завершает эту главу.

## 6.1. Системы типов

Построение программы проверки типов базируется на информации о синтаксических конструкциях языка, представлении типов и правилах присвоения типов конструкциям языка. Следующие цитаты из сообщения о Pascal и справочного руководства по С представляют собой примеры информации, с которой может начать работу создатель компилятора.

- “Если оба operandы арифметического оператора сложения, вычитания и умножения имеют целый тип, то результат также имеет целый тип”.
- “Результат применения унарного оператора `&` является указателем на объект, к которому обращается данный оператор. Если тип operand — ‘...’, то тип результата — ‘указатель на ...’”.

В приведенных выше отрывках неявно говорится, что каждое выражение имеет связанный с ним тип. Кроме того, тип имеет структуру; ‘тип указатель на ...’ строится из типа ‘...’.

И в Pascal, и в С типы делятся на базовые (фундаментальные) и создаваемые. Базовые типы — это атомарные типы без внутренней структуры, к которым в Pascal относятся, например, `boolean`, `char`, `integer` и `real`. Типы поддиапазонов, наподобие `1..10`, и перечисления вроде

```
(violet, indigo, blue, green, yellow, orange, red)
```

также могут рассматриваться как базовые. Pascal позволяет программисту создавать типы из базовых и других создаваемых типов, примерами чего могут служить массивы, записи и множества. Кроме того, в качестве создаваемых типов могут рассматриваться указатели и функции.

## Выражения типов

Тип языковой конструкции будет описываться “выражением типа”. Говоря неформально, выражение типа либо представляет собой базовый тип, либо построено с помощью применения оператора, называемого *конструктором типа*, к другим выражениям типа. Множества базовых типов и конструкторов зависят от проверяемого языка.

В этой главе используется следующее определение *выражений типа*.

1. Базовый тип является выражением типа. В базовые типы входят *boolean*, *char*, *integer* и *real*. Специальный базовый тип *type\_error* сигнализирует о возникшей в процессе проверки типа ошибке. И наконец, базовый тип *void* означает “отсутствие значения” и позволяет проверять не только выражения, но и инструкции.
2. Поскольку выражения типа можно именовать, имя типа является выражением типа. Пример использования имен типов можно найти в п. 3(с) ниже; выражения типа, содержащие имена, рассматриваются в разделе 6.3.
3. Конструктор типа, применяемый к выражениям типа, является выражением типа. Конструкторы включают следующее.
  - a) *Массивы*. Если  $T$  — выражение типа, то  $\text{array}(I, T)$  является выражением типа, указывающим тип массива с элементами типа  $T$  и множеством индексов  $I$ . Чаще всего  $I$  представляет собой диапазон целых чисел. Например, объявление в языке программирования Pascal

```
var A: array[1..10] of integer;
```

связывает выражение типа  $\text{array}[1..10]$  of *integer* с  $A$ .
  - b) *Произведения*. Если  $T_1$  и  $T_2$  являются выражениями типа, то их декартово произведение  $T_1 \times T_2$  — выражение типа. Мы считаем декартово произведение  $\times$  левоассоциативным.
  - c) *Записи*. Отличие записи от произведения состоит в том, что поля записи имеют имена. Конструктор типа *record* применяется к кортежу, образованному именами и типами полей. (Технически имена полей должны быть частью конструктора типа, однако удобнее хранить имена полей вместе со связанными с ними типами. В главе 8, “Генерация промежуточного кода”, конструктор типа *record* применяется к указателю на таблицу символов, содержащую записи для имен полей.) Например, программный фрагмент на языке Pascal

```
type row = record
 address: integer;
 lexeme : array[1..15] of char
 end;
var table: array[1..101] of row;
```

объявляет имя типа *row*, представляющее выражение типа *record((address×integer)×(lexeme×array(1..15, char)))*, и переменную *table* — массив записей этого типа.
  - d) *Указатели*. Если  $T$  является выражением типа, то *pointer( $T$ )* представляет собой выражение типа, описывающее тип “указатель на объект типа  $T$ ”. Так, в Pascal описание `var p: ^row` объявляет переменную *p* как имеющую тип *pointer(row)*.

- e) *Функции.* С математической точки зрения функция отображает элементы одного множества, *области определения* (domain), на другое множество, *область значений* (range). В языках программирования мы можем рассматривать функции как отображения *типа области определения D* на *тип области значений R*. Тип такой функции записывается с помощью выражения типа  $D \rightarrow R$ . Например, встроенная функция Pascal mod имеет тип области определения  $\text{int} \times \text{int}$  (пары целых значений) и тип области значений  $\text{int}$ , т.е. мы можем сказать, что mod имеет тип<sup>1</sup>  $\text{int} \times \text{int} \rightarrow \text{int}$ .

В качестве другого примера объявление Pascal

```
function f(a, b: char) : ^integer; ...
```

говорит, что тип области определения  $f$  —  $\text{char} \times \text{char}$ , а тип области значений —  $\text{pointer}(\text{integer})$ . Тип  $f$  можно записать с помощью выражения типа  $\text{char} \times \text{char} \rightarrow \text{pointer}(\text{integer})$ .

Зачастую по причинам, связанным с реализацией и обсуждаемым в следующей главе, вводятся ограничения на тип области значений функции, например функции не могут возвращать массивы. Однако имеются языки, среди которых наиболее ярким примером служит Lisp, позволяющие возвращать объекты произвольного типа; так что, например, можно определить функцию  $g$  типа  $(\text{integer} \rightarrow \text{integer}) \rightarrow (\text{integer} \rightarrow \text{integer})$ . Таким образом, функция  $g$  принимает в качестве аргумента функцию, отображающую целые числа на целые числа, и возвращает в качестве результата функцию того же типа.

#### 4. Выражения типа могут содержать переменные, значения которых являются выражениями типа. Переменные типа будут рассмотрены в разделе 6.6.

Удобным способом представления выражений типа является применение графов. Используя синтаксически управляемый подход из раздела 5.2, мы можем построить дерево или даг для выражения типа, с внутренними узлами для конструкторов типа и листьями для базовых типов, имен и переменных типов (рис. 6.2). Примеры представлений выражений типа, используемые в компиляторах, приведены в разделе 6.3.



Рис. 6.2. Дерево и даг для типа  $\text{char} \times \text{char} \rightarrow \text{pointer}(\text{integer})$

## Системы типов

*Система типов* представляет собой набор правил для назначения выражений типов различным частям программы. Программа проверки типов реализует систему типа. В этой главе системы типов определяются синтаксически управляемым способом, так что они могут быть легко реализованы с использованием технологий из предыдущей главы.

<sup>1</sup> Мы полагаем, что  $\times$  имеет более высокий приоритет, чем  $\rightarrow$ , так что  $\text{int} \times \text{int} \rightarrow \text{int}$  представляет собой то же, что и  $(\text{int} \times \text{int}) \rightarrow \text{int}$ . Кроме того, оператор  $\rightarrow$  правоассоциативен.

Разные компиляторы и процессоры одного и того же языка могут использовать различные системы типов. Например, в Pascal тип массива включает множество его индексов; так что функция, получающая массив в качестве аргумента, может быть применима только к массивам с предопределенным множеством индексов. Однако многие компиляторы языка Pascal позволяют при передаче массивов оставлять множество индексов не определенным. Соответственно, такие компиляторы используют системы типов, отличающиеся от системы типов в определении языка Pascal<sup>2</sup>. Можно упомянуть также команду UNIX `lint` для проверки программ на языке C, которая использует более подробную систему типов, чем компилятор C.

## Статическая и динамическая проверка типов

Проверка, выполняемая компилятором, называется статической, а проверка, выполняемая целевой программой, — динамической. В принципе, любая проверка может быть выполнена динамически, если целевой код содержит вместе со значением элемента его тип.

Надежная система типов (sound type system) устраняет необходимость динамической проверки типов во избежание связанных с ними ошибок, так как позволяет нам статически определить, что такие ошибки не могут возникнуть в процессе работы программы. Таким образом, если система типов присвоила части программы тип, отличный от `type_error`, то ошибки, связанные с типами, при работе этой части программы возникнуть не могут. Язык называется *строго типизированным*, если его компилятор может гарантировать, что скомпилированная программа будет выполняться без ошибок, связанных с типами.

На практике некоторые проверки могут быть выполнены только динамически. Например, если мы объявим

```
table: array[0..255] of char;
i : integer;;
```

а затем вычислим `table[i]`, то компилятор не в состоянии гарантировать, что в процессе выполнения программы значение `i` будет находиться в диапазоне 0...255<sup>3</sup>.

## Восстановление после ошибки

Поскольку проверка типов в состоянии “выловить” определенные ошибки в программе, важно, чтобы программа проверки типов выполняла некоторые разумные действия при обнаружении ошибок. Наименьшее, что должен сделать компилятор, — это сообщить о природе и местонахождении ошибки. Желательно восстановление программы проверки типов после выявленной ошибки, с тем чтобы она могла продолжить проверку оставшейся части входного потока. Поскольку обработка ошибок влияет на правила проверки типов, она должна быть встроена в систему типов изначально; правила проверки типов должны быть записаны так, чтобы справляться с ошибками.

<sup>2</sup> Различные расширения языка, обеспечивая компиляцию программ, написанных с учетом требований стандарта, в то же время позволяют программисту создавать более мощные и эффективные программы. Однако побочный эффект, заключающийся в непереносимости таких программ между различными компиляторами, часто сводит на нет все достоинства языковых расширений. — Прим. ред.

<sup>3</sup> Для выяснения, находится ли `i` в определенных границах, можно использовать технологии анализа потоков данных, подобные изложенным в главе 10, “Оптимизация кода”. Однако универсальной технологии, обеспечивающей корректное решение во всех случаях, не существует.

Включение обработки ошибок может привести к системе типов, выходящей за пределы, необходимые только для описания корректных программ. Например, при выявлении ошибки мы можем не знать тип ошибочного фрагмента программы. При отсутствии необходимой информации используются технологии, подобные применяемым к языкам, которые не требуют объявления идентификаторов до их использования. Переменные типа, рассматриваемые в разделе 6.6, могут использоваться для того, чтобы гарантировать непротиворечивое применение необъявленных или некорректно объявленных идентификаторов.

## 6.2. Спецификация простой программы проверки типов

В этом разделе мы определим программу проверки типов для простого языка, в котором тип каждого идентификатора должен быть объявлен до использования этого идентификатора. Программа проверки типов представляет собой схему трансляции, синтезирующую тип каждого выражения из типов их подвыражений, и может обрабатывать массивы, указатели, инструкции и функции.

### Простой язык

Грамматика на рис. 6.3 порождает программы, представленные нетерминалом  $P$  и состоящие из последовательности объявлений  $D$ , за которой следует единственное выражение  $E$ .

$$\begin{array}{lcl} P & \rightarrow & D ; E \\ D & \rightarrow & D ; D \mid id : T \\ T & \rightarrow & char \mid integer \mid array [ num ] of T \mid \uparrow T \\ E & \rightarrow & literal \mid num \mid id \mid E \text{ mod } E \mid E [ E ] \mid E \uparrow \end{array}$$

Рис. 6.3. Грамматика исходного языка

Вот пример программы, порожденной грамматикой на рис. 6.3:

```
key: integer;
key mod 1999
```

Перед тем как приступить к рассмотрению выражений, рассмотрим типы данного языка. Сам по себе язык имеет два базовых типа — *char* и *integer*; третий базовый тип — *type\_error* — используется для указания об ошибке. Для простоты мы считаем, что все массивы начинаются с элемента 1, т.е. *array[256]* of *char* приводит к выражению типа *array(1..256, char)*, состоящему из конструктора *array*, примененного к поддиапазону 1..256 и типу *char*. Как и в Pascal, префиксный оператор  $\uparrow$  в объявлении создает тип указателя, так что  $\uparrow integer$  дает выражение типа *pointer(integer)*, состоящее из конструктора *pointer*, примененного к типу *integer*.

В схеме трансляции на рис. 6.4 действие, связанное с продукцией  $D \rightarrow id : T$ , сохраняет тип в записи таблицы символов для идентификатора. Действие *addtype(id.entry, T.type)* применяется к синтезируемому атрибуту *entry*, указывающему на запись в таблице символов для *id*, и выражению типа, представленному синтезируемым атрибутом *type* нетерминала *T*.

Если *T* порождает *char* или *integer*, то *T.type* соответственно определяется как *char* или *integer*. Верхняя граница массива определяется по атрибуту *val* токена *num* (этот ат-

рибут дает целое, представленное токеном `num`). Массивы, как мы уже говорили, начинаются с элемента 1, так что конструктор типа `array` применяется к поддиапазону `1..num.val` и типу элементов.

|                                      |                                                        |
|--------------------------------------|--------------------------------------------------------|
| $P \rightarrow D ; E$                |                                                        |
| $D \rightarrow D ; D$                |                                                        |
| $D \rightarrow id : T$               | { <code>addtype(id.entry, T.type)</code> }             |
| $T \rightarrow char$                 | { <code>T.type := char</code> }                        |
| $T \rightarrow integer$              | { <code>T.type := integer</code> }                     |
| $T \rightarrow \uparrow T_1$         | { <code>T.type := pointer(T_1.type)</code> }           |
| $T \rightarrow array [ num ] of T_1$ | { <code>T.type := array(1..num.val), T_1.type</code> } |

Рис. 6.4. Часть схемы трансляции, сохраняющая типы идентификаторов

Поскольку  $D$  в правой части продукции  $P \rightarrow D ; E$  располагается раньше  $E$ , мы можем быть уверены, что типы всех объявляемых идентификаторов будут сохранены до проверки выражения, порожденного  $E$  (см. главу 5, “Синтаксически управляемая трансляция”). При желании схемы трансляции в этом разделе могут быть реализованы как в процессе восходящего, так и нисходящего синтаксического анализа путем внесения соответствующих изменений в грамматику на рис. 6.3.

## Проверка типов выражений

В следующих правилах синтезируемый атрибут `type` символа  $E$  дает тип выражения, присвоенный системой типов выражению, порождаемому  $E$ . Следующие семантические правила гласят, что константы, представленные токенами `literal` и `num`, имеют типы `char` и `integer` соответственно.

|                         |                                    |
|-------------------------|------------------------------------|
| $E \rightarrow literal$ | { <code>E.type := char</code> }    |
| $E \rightarrow num$     | { <code>E.type := integer</code> } |

Для получения типа, сохраненного в записи таблицы символов, на которую указывает  $e$ , используем функцию `lookup(e)`. Когда в выражении встречается идентификатор, мы получаем тип из его объявления и присваиваем атрибуту `type`:

|                    |                                             |
|--------------------|---------------------------------------------|
| $E \rightarrow id$ | { <code>E.type := lookup(id.entry)</code> } |
|--------------------|---------------------------------------------|

Выражение, формируемое путем применения оператора `mod` к двум подвыражениям типа `integer`, имеет тип `integer`; в противном случае его тип — `type_error`. Соответствующее правило выглядит следующим образом:

|                                 |                                                                                                                                      |
|---------------------------------|--------------------------------------------------------------------------------------------------------------------------------------|
| $E \rightarrow E_1 \ mod \ E_2$ | { <code>E.type := if E_1.type = integer and</code><br><code>E_2.type = integer then integer</code><br><code>else type_error</code> } |
|---------------------------------|--------------------------------------------------------------------------------------------------------------------------------------|

В обращении к массиву  $E_1[E_2]$  индексное выражение должно иметь целый тип, и в этом случае тип результата определяется типом элемента  $t$ , полученным из типа `array(s, t)` выражения  $E_1$ ; множество индексов  $s$  не используется.

|                             |                                                                                                                                   |
|-----------------------------|-----------------------------------------------------------------------------------------------------------------------------------|
| $E \rightarrow E_1 [ E_2 ]$ | { <code>E.type := if E_2.type = integer and</code><br><code>E_1.type = array(s,t) then t</code><br><code>else type_error</code> } |
|-----------------------------|-----------------------------------------------------------------------------------------------------------------------------------|

В выражении постфиксный оператор  $\uparrow$  порождает объект, на который указывает операнд. Тип  $E \uparrow$  является типом  $t$  объекта, на который указывает указатель  $E$ :

$$E \rightarrow E_1 \uparrow \quad \{ E.type := \text{if } E_1.type = \text{pointer}(t) \text{ then } t \\ \text{else type\_error} \}$$

Мы оставляем читателю рассмотрение других продукции и семантических правил, обеспечивающих использование дополнительных типов и операций в выражениях. Например, чтобы разрешить применение идентификаторов типа *boolean*, мы должны ввести в грамматику на рис. 6.3 продукцию  $T \rightarrow \text{boolean}$ . Введение в продукцию грамматики для  $E$  операторов сравнения (наподобие  $<$ ) и логических операторов (*and* и других) обеспечит возможность построения выражений с типом *boolean*.

## Проверка типов инструкций

Поскольку обычно языковые конструкции типа инструкций не имеют значений, им присваивается специальный базовый тип *void*. Инструкция с ошибкой получает тип *type\_error*.

Рассматриваемые нами инструкции — присвоение, условные инструкции и цикл *while*. Последовательности инструкций разделяются точкой с запятой. Продукции на рис. 6.5 могут быть объединены с продукциями на рис. 6.3, если заменить продукцию для всей программы продукцией  $P \rightarrow D ; S$ . Теперь программа состоит из объявлений, за которыми следуют инструкции. Приведенные выше правила для проверки выражений остаются необходимыми, поскольку инструкции могут содержать выражения.

$$\begin{array}{ll} S \rightarrow \text{id} := E & \{ S.type := \text{if id.type} = E.type \text{ then void else type\_error} \} \\ S \rightarrow \text{if } E \text{ then } S_1 & \{ S.type := \text{if } E.type = \text{boolean} \text{ then } S_1.type \text{ else type\_error} \} \\ S \rightarrow \text{while } E \text{ do } S_1 & \{ S.type := \text{if } E.type = \text{boolean} \text{ then } S_1.type \text{ else type\_error} \} \\ S \rightarrow S_1 ; S_2 & \{ S.type := \text{if } S_1.type = \text{void and } S_2.type = \text{void then void else type\_error} \} \end{array}$$

Рис. 6.5. Схема трансляции для проверки типа инструкций

Правила для проверки инструкций приведены на рис. 6.5. Первое правило проверяет, имеют ли левая и правая части присвоения одинаковые типы<sup>4</sup>. Второе и третье правила определяют, что выражения в условной инструкции и цикле *while* должны иметь тип *boolean*. Согласно последнему правилу, происходит распространение ошибки, поскольку последовательность инструкций имеет тип *void* только в том случае, когда каждая составляющая инструкция также имеет тип *void*. Во всех приведенных правилах несоответствие типов приводит к назначению типа *type\_error*; конечно, хорошая программа проверки типов должна при обнаружении ошибки сообщить, в чем она состоит и где находится.

## Проверка типов функций

Использование функций можно описать продукцией

$$E \rightarrow E(E),$$

в которой выражение представляет собой применение одного выражения к другому. Правила для связывания выражений типа с нетерминалом  $T$  могут быть расширены

<sup>4</sup> Кроме того, при присвоении мы должны различать *l*- и *r*-значения в левой части присвоения. Например, несмотря на идентичность типов присвоение  $1 := 2$  недопустимо, поскольку присвоить значение константе 1 невозможно.

следующей продукцией и действием, обеспечивающими работу с типами функций в объявлениях.

$$T \rightarrow T_1 \rightarrow' T_2 \quad \{ T.type := T_1.type \rightarrow T_2.type \}$$

Кавычки вокруг стрелки, используемой как конструктор функции, приведены для отличия ее от стрелки, используемой в качестве метасимвола в продукции.

Правило для проверки типа функции следующее:

$$E \rightarrow E_1 ( E_2 ) \quad \{ E.type := \text{if } E_2.type = s \text{ and } E_1.type = s \rightarrow t \text{ then } t \text{ else type\_error} \}$$

Это правило гласит, что в выражении, полученном путем применения  $E_1$  к  $E_2$ , тип  $E_1$  должен быть функцией  $s \rightarrow t$  из типа  $s$  выражения  $E_2$  в некоторый тип области значений  $t$ ; типом  $E_1(E_2)$  является  $t$ .

Многие вопросы, связанные с проверкой типов функций, могут рассматриваться в соответствии с приведенным выше простым синтаксисом. Обобщение на функции с более чем одним аргументом выполняется путем построения произведения типов аргументов. Заметим, что  $n$  аргументов типов  $T_1, \dots, T_n$  могут рассматриваться как единый аргумент типа  $T_1 \times \dots \times T_n$ . Например, мы можем записать

$$\text{root} : (\text{real} \rightarrow \text{real}) \times \text{real} \rightarrow \text{real} \quad (6.1)$$

для объявления функции `root`, которая принимает два аргумента — функцию из типа `real` в этот же тип и значение `real` — и возвращает значение `real`. Такое объявление на языке программирования Pascal выглядит как

```
function root (function f (real) : real; x : real) : real
```

Синтаксис (6.1) отделяет объявление типа функции от имен ее параметров.

### 6.3. Эквивалентность выражений типа

Правила проверки типов в предыдущем разделе имеют вид “если два выражения типов равны, то вернуть некоторый тип, иначе вернуть `type_error`”. Следовательно, очень важно иметь точное определение равенства двух типов. Потенциальная неоднозначность возникает с именами выражений типа, которые затем используются в последующих выражениях типа. Ключевой вопрос в том, является ли имя в выражении типа само по себе выражением или сокращением для другого выражения типа.

Поскольку между понятием эквивалентности типов и их представлением имеется взаимосвязь, мы будем говорить о них вместе. Для более высокой эффективности работы компиляторы используют представления, которые позволяют быстро определить, являются ли типы эквивалентными. Представление эквивалентности типов, реализованное конкретным компилятором, зачастую рассматривается с использованием концепций структурной и именной эквивалентностей, рассматриваемых в этом разделе. Обсуждение проводится с применением представления выражений типа в виде графов с листьями для базовых типов и имен типов и внутренними узлами для конструкторов типов, как было показано на рис. 6.2. Как мы увидим, рекурсивно определенные типы приводят к появлению циклов в графике типов, если рассматривать имена как сокращения для выражений типа.

#### Структурная эквивалентность выражений типа

Поскольку выражения типа строятся из базовых типов и конструкторов, естественным понятием эквивалентности двух выражений типа является *структурная эквивалентность*, т.е. два выражения либо представляют собой один и тот же базовый тип, ли-

бо сформированы путем применения одного и того же конструктора к структурно эквивалентным типам. Таким образом, два выражения типа структурно эквивалентны тогда и только тогда, когда они идентичны. Например, выражение типа *integer* эквивалентно только *integer*, потому что это один и тот же базовый тип. Аналогично тип *pointer(integer)* эквивалентен только типу *pointer(integer)*, поскольку оба построены посредством применения одного и того же конструктора *pointer* к эквивалентным типам. Если мы воспользуемся методом номера значения из алгоритма 5.1 для построения дага, представляющего выражение типа, то идентичные выражения типа будут представлены одним и тем же узлом.

На практике для отражения реальных правил проверки типов исходного языка часто требуется внесение изменений в представление о структурной эквивалентности. Например, при передаче массивов в качестве параметров мы можем не включать границы массива как составную часть типа.

Алгоритм проверки структурной эквивалентности, представленный на рис. 6.6, может быть адаптирован для проверки модифицированного представления эквивалентности. В нем предполагается, что используемые конструкторы типов ограничиваются массивами, произведениями, указателями и функциями. Алгоритм рекурсивно сравнивает структуры выражений типов без проверки цикличности, так что он может быть применен к представлению в виде дерева или дага. Идентичные выражения типов не обязательно будут представлены одним и тем же узлом дага. Структурная эквивалентность узлов в графах типа с циклами может быть проверена с использованием алгоритма из раздела 6.7.

```

(1) function sequiv(s, t): boolean;
begin
(2) if s и t представляют один и тот же
 базовый тип then
 return true
(3) else if s = array(s1, s2) and t = array(t1, t2) then
 return sequiv(s1, t1) and sequiv(s2, t2)
(4) else if s = s1 × s2 and t = t1 × t2 then
 return sequiv(s1, t1) and sequiv(s2, t2)
(5) else if s = pointer(s1) and t = pointer(t1) then
 return sequiv(s1, t1)
(6) else if s = s1 → s2 and t = t1 → t2 then
 return sequiv(s1, t1) and sequiv(s2, t2)
(7) else
 return false
(8) end

```

Рис. 6.6. Проверка структурной эквивалентности двух выражений типа *s* и *t*

Границы массивов *s<sub>1</sub>* и *t<sub>1</sub>* в

```

s = array(s1, s2)
t = array(t1, t2)

```

игнорируются, если переписать проверку эквивалентности массивов в строках (4) и (5) на рис. 6.6 следующим образом.

```

(4) else if s = array(s1, s2) and t = array(t1, t2) then
(5) return sequiv(s2, t2)

```

В некоторых ситуациях для выражений типов можно найти значительно более компактную по сравнению с графом запись. В следующем примере информация о выражении типа закодирована последовательностью битов, а значит, может быть интерпретирована как целое число. Кодирование осуществляется таким образом, что различные числа представляют структурно незэквивалентные выражения типов. Проверка структурной эквивалентности при этом ускоряется за счет того, что сначала проверяется незэквивалентность путем сравнения целых представлений типов, и только в случае их равенства применяется алгоритм на рис. 6.6.

### Пример 6.1

Кодирование выражений типов в этом примере взято из компилятора C, созданного Д. Ритчи. Оно также используется в компиляторе C, описанном в работе [217].

Рассмотрим выражения типа со следующими конструкторами типа для указателей, функций и массивов: *pointer(t)* означает указатель на тип *t*, *freturns(t)* — функцию с некоторыми аргументами, возвращающую объект типа *t*, *array(t)* — массив (неопределенной длины) элементов типа *t*. Заметим, что мы упростили конструкторы типов для функций и массивов. Мы будем отслеживать количество элементов в массиве, но оно будет храниться где-то в другом месте, как и количество, и типы аргументов функции. Таким образом, объекты, структурно эквивалентные с точки зрения целочисленного кодирования, могут быть незэквивалентны с точки зрения проверки на рис. 6.6, применяемой к более подробной системе типов, чем используемая в данном примере.

Поскольку каждый из этих конструкторов представляет собой унарный оператор, выражения типа, образованные применением этих конструкторов к базовым типам, имеют весьма однородную структуру. Вот примеры таких выражений типа.

```
char
freturns(char)
pointer(freturns(char))
array(pointer(freturns(char))))
```

Каждое из приведенных выражений может быть представлено последовательностью битов с использованием простой схемы кодирования. Поскольку имеется только три конструктора типов, для кодирования конструктора нам достаточно двух битов.

| КОНСТРУКТОР ТИПА | КОДИРОВАНИЕ |
|------------------|-------------|
| <i>pointer</i>   | 01          |
| <i>array</i>     | 10          |
| <i>freturns</i>  | 11          |

Базовые типы C в [217] кодируются с использованием четырех битов; наши четыре базовых типа могут быть закодированы следующим образом.

| БАЗОВЫЙ ТИП    | КОДИРОВАНИЕ |
|----------------|-------------|
| <i>boolean</i> | 0000        |
| <i>char</i>    | 0001        |
| <i>integer</i> | 0010        |
| <i>real</i>    | 0011        |

Ограниченные выражения типа теперь могут быть представлены последовательностью битов. Правые четыре бита представляют базовый тип в выражении типа.

При перемещении справа налево каждые два бита указывают конструктор типа, применяемый к типу, закодированному битами справа от этой пары битов, как показано в следующем примере.

| ВЫРАЖЕНИЕ ТИПА                        | КОД         |
|---------------------------------------|-------------|
| <i>char</i>                           | 000000 0001 |
| <i>freturns(char)</i>                 | 000011 0001 |
| <i>pointer(freturns(char))</i>        | 000111 0001 |
| <i>array(pointer(freturns(char)))</i> | 100111 0001 |

(Упр. 6.12 также посвящено этому вопросу.)

Кроме экономии памяти, такое представление отслеживает конструкторы, появляющиеся в любом выражении типа. Две различные последовательности битов не могут представлять один и тот же тип, поскольку при этом различны либо базовые типы, либо примененная к ним последовательность конструкторов. Конечно, разные типы могут иметь одну и ту же последовательность битов — в силу того, что размеры массивов и аргументы функций не представлены в данной системе кодирования.

Кодирование из этого примера может быть расширено для включения типов записей. Идея заключается в рассмотрении каждой записи в качестве базового типа при кодировании; отдельная последовательность битов кодирует тип каждого поля записи. Эквивалентность типов в С рассматривается в примере 6.4. □

## Имена выражений типа

В некоторых языках программирования типам могут даваться имена. Например, во фрагменте программы на языке Pascal

```
type link = ↑cell;
var next : link;
 last : link;
 p : ↑cell;
 q, r : ↑cell;
```

(6.2)

идентификатор *link* объявлен в качестве имени типа  $\uparrow$ *cell*. Возникает вопрос, идентичны ли типы переменных *next*, *last*, *p*, *q* и *r*? Интересно, что ответ на этот вопрос зависит от реализации. Проблема в том, что в официальном сообщении о Pascal не определяется идентичность типов.

Для моделирования этой ситуации мы разрешим именовать выражения типа и позволим этим именам присутствовать в выражениях типа, где до сих пор использовались только базовые типы. Например, если *cell* является именем выражения типа, то и *pointer(cell)* также представляет собой выражение типа. Пока предположим, что циклические определения выражения типа отсутствуют (например, не может быть определения *cell* как имени выражения типа, содержащего *cell*).

Если в выражениях типа разрешено применение имен, то в зависимости от их трактовки возникает два варианта эквивалентности выражений типа. С точки зрения *эквивалентности имен (именной эквивалентности)* имя каждого типа представляет собственный тип, отличный от типа с другим именем. При рассмотрении *структурной эквивалентности* имена замещаются выражениями типа, определяемыми этими именами; так что два выражения типа структурно эквивалентны, если представляют два структурно эквивалентных выражения типа после замены всех имен соответствующими выражениями.

## Пример 6.2

Выражения типа, связанные с переменными в объявлениях (6.2), приведены в следующей таблице.

| ПЕРЕМЕННАЯ | ВЫРАЖЕНИЕ ТИПА       |
|------------|----------------------|
| next       | link                 |
| last       | link                 |
| p          | <i>pointer(cell)</i> |
| q          | <i>pointer(cell)</i> |
| r          | <i>pointer(cell)</i> |

С точки зрения эквивалентности имен переменные *next* и *last* имеют один и тот же тип, поскольку с ними связаны одинаковые выражения типа. Переменные *p*, *q* и *r* также одного типа, но *p* и *next* разнотипны, поскольку связанные с ними выражения типа различны. Однако с точки зрения структурной эквивалентности все пять переменных имеют один и тот же тип, поскольку *link* представляет собой не что иное, как имя для выражения типа *pointer(cell)*.  $\square$

Концепции именной и структурной эквивалентности полезны при объяснении правил, используемых различными языками для назначения типов идентификаторам посредством объявлений.

## Пример 6.3

В Pascal возникает определенная путаница вследствие того, что многие реализации связывают с каждым объявляемым идентификатором неявное имя типа. Если в объявлении используется выражение типа, не являющееся именем, создается неявное имя типа. Такое имя создается заново всякий раз, когда в объявлении переменной встречается выражение типа.

Таким образом, неявные имена в (6.2) создаются для объявлений *p*, *q* и *r*. Фактически объявления (6.2) трактуются следующим образом.

```
type link = ↑cell;
 np = ↑cell;
 nqr = ↑cell;
var next : link;
 last : link;
 p : np;
 q : nqr;
 r : nqr;
```

Здесь введены новые имена типов *np* и *nqr*. С точки зрения эквивалентности имен переменные *last* и *next*, а также *q* и *r* рассматриваются как однотипные, поскольку имена их типов (явные или неявные) совпадают. Однако переменные *next*, *p* и *q* — разных типов, поскольку имена их типов различны.

Типичная реализация состоит в построении для представления типов соответствующего графа. Всякий раз, когда в объявлении встречается конструктор типа или базовый тип, создается новый узел, а при появлении нового имени типа создается новый лист (однако при этом мы отслеживаем, на какое выражение типа ссылается это имя). При использовании такого представления два выражения типа эквивалентны, если они пред-

ставлены одним и тем же узлом в графе типов. На рис. 6.7 показан график типов для объявлений (6.2). Пунктирные линии показывают связь между переменными и узлами в графике типов. Заметим, что имя типа *cell* имеет три родительских узла, помеченных как *pointer*. Знак равенства поставлен между именем типа *link* и узлом в графике типов, на который ссылается это имя.  $\square$

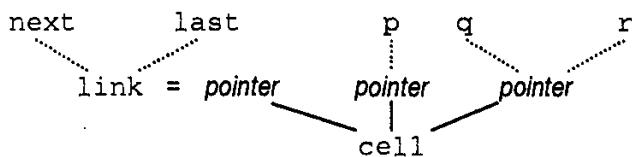


Рис. 6.7. Связь переменных и узлов в графике типов

## Циклы в представлениях типов

Базовые структуры данных, такие как связанные списки и деревья, зачастую определяются рекурсивно — например, связанный список либо пуст, либо состоит из ячейки с указателем на связанный список. Такие структуры данных обычно реализуются с использованием записей, содержащих указатели на такие же записи; имена типов играют существенную роль в определении типов таких записей.

Рассмотрим связанный список ячеек, каждая из которых содержит некоторую целочисленную информацию и указатель на следующую ячейку в списке. Объявление имен типов в Pascal, соответствующее связям и ячейкам, имеет следующий вид.

```

type link = ^cell;
 cell = record
 info : integer;
 next : link;
 end;

```

Обратите внимание, что имя типа *link* определено с помощью имени *cell*, а *cell* — с помощью *link*; так что определения рекурсивны.

Рекурсивно определенные имена типов могут быть удалены, если мы готовы допустить наличие циклов в графике типов. Если заменить *link* на *pointer(cell)*, для *cell* будет получено выражение типа, показанное на рис. 6.8а. Используя цикл, как показано на рис. 6.8б, мы можем удалить упоминание о *cell* из части графа, расположенной ниже узла, помеченного как *record*.

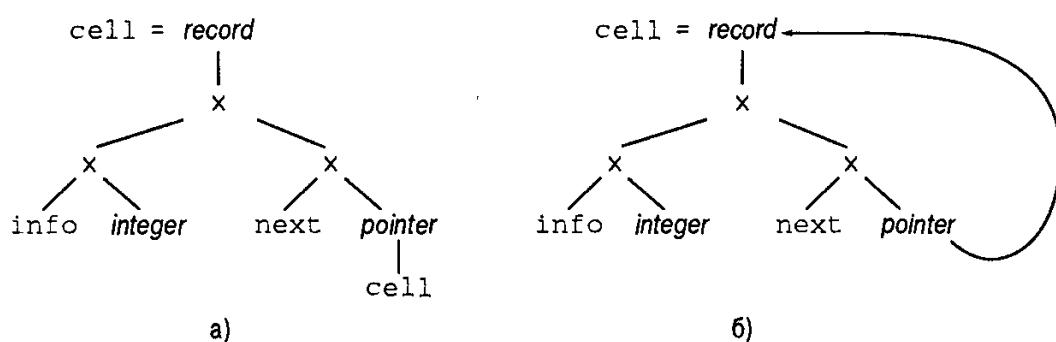


Рис. 6.8. Рекурсивно определенное имя типа *cell*

#### Пример 6.4

В языке С можно избежать циклов в графе типа, используя структурную эквивалентность для всех типов, за исключением записей. В С объявление `cell` будет иметь следующий вид.

```
struct cell {
 int info;
 struct cell * next;
};
```

Вместо ключевого слова `record`, С использует `struct`, и имя `cell` становится частью типа записи. По сути, С использует нециклическое представление на рис. 6.8а.

Язык С требует объявления имен типов до их использования, за исключением указателей на необъявленные типы записей. Все потенциальные циклы, таким образом, возникают из-за указателей на записи. Поскольку имя записи является частью ее типа, проверка структурной эквивалентности прекращается по достижении конструктора записи — сравниваемые типы эквивалентны только в том случае, когда имеют одинаково именованный тип записи. □

## 6.4. Преобразования типов

Рассмотрим выражения наподобие `x+i`, где `x` имеет действительный, а `i` — целочисленный тип. Поскольку представление действительных и целых чисел в компьютере различно и для операций над целыми и действительными числами используются различные машинные инструкции, компилятор, возможно, должен первоначально привести один из операндов к типу другого, чтобы они имели один и тот же тип в процессе сложения.

Определение языка указывает необходимые преобразования. При присвоении целого числа действительной переменной (или наоборот) присваиваемое число преобразуется в тип переменной в левой части присвоения. В выражениях целочисленные значения обычно преобразуются в действительные числа, после чего над парой действительных операндов выполняется требуемая действительная операция. Подпрограмма проверки типов компилятора может использоваться для добавления в промежуточное представление исходной программы операций по необходимому преобразованию типов. Например, постфиксная запись для `x+i` может выглядеть следующим образом.

```
x i inttoreal real+
```

Здесь оператор `inttoreal` преобразует `i` из целого в действительное, после чего оператор `real+` выполняет сложение двух действительных операндов.

Преобразование типов часто возникает и в другом контексте. Символ, имеющий разные значения в зависимости от контекста, называется перегруженным (overloaded). Перегрузка будет рассмотрена в следующем разделе, а здесь она упоминается в связи с тем, что ей часто сопутствует преобразование типов.

### Неявное преобразование типов

Преобразование одного типа в другой называется *неявным*, если оно должно выполняться компилятором автоматически. Во многих языках неявное преобразование типов ограничено ситуациями, где в принципе не происходит потери информации, например целое число может быть преобразовано в действительное, но не наоборот (на практике,

однако, при размещении действительного числа в том же количестве бит, что и целое, потери вполне возможны).

Преобразование называется *явным*, если для его задания программист должен специально что-то написать. Например, на самом деле все преобразования типов в Ada явные. Явные преобразования с точки зрения программы проверки типов выглядят как применение функции, так что их наличие не вызывает новых проблем.

Например, в Pascal встроенная функция `ord` отображает символ в целое число, а `chr` выполняет обратное преобразование целого числа в символ, так что эти преобразования являются явными. Язык же С в арифметических выражениях неявно преобразует ASCII-символы в целые числа от 0 до 127.

### Пример 6.5

Рассмотрим выражения, образованные путем применения арифметического оператора `op` к константам и идентификаторам, как показано в грамматике на рис. 6.9. Предположим, что имеется два типа — действительный и целочисленный, причем при необходимости целочисленный тип преобразуется в действительный. Атрибут *type* нетерминала *E* может принимать значения *real* и *integer*, а используемые правила проверки типов вы найдете в таблице на рис. 6.9. Как и в разделе 6.2, функция *lookup(e)* возвращает тип, сохраненный в записи таблицы символов, на которую указывает *e*.  $\square$

| ПРОДУКЦИЯ                               | СЕМАНТИЧЕСКОЕ ПРАВИЛО                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                   |
|-----------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| $E \rightarrow \text{num}$              | $E.type := \text{integer}$                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                              |
| $E \rightarrow \text{num} . \text{num}$ | $E.type := \text{real}$                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                 |
| $E \rightarrow \text{id}$               | $E.type := \text{lookup(id.entry)}$                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                     |
| $E \rightarrow E_1 \text{ op } E_2$     | $E.type :=$<br>$\quad \text{if } E_1.type = \text{integer} \text{ and } E_2.type = \text{integer}$<br>$\quad \quad \text{then integer}$<br>$\quad \text{else if } E_1.type = \text{integer} \text{ and } E_2.type = \text{real}$<br>$\quad \quad \text{then real}$<br>$\quad \text{else if } E_1.type = \text{real} \text{ and } E_2.type = \text{integer}$<br>$\quad \quad \text{then real}$<br>$\quad \text{else if } E_1.type = \text{real} \text{ and } E_2.type = \text{real}$<br>$\quad \quad \text{then real}$<br>$\quad \text{else type error}$ |

Рис. 6.9. Правила проверки типов для неявного преобразования *integer* в *real*

Обычно неявное преобразование констант выполняется в процессе компиляции, что зачастую существенно повышает производительность компилируемой программы. В приведенном ниже фрагменте кода X — массив действительных чисел, инициализированных единицами. В работе [55] указывается, что при использовании одного компилятора Pascal фрагмент кода

```
for I := 1 to N do X[I] := 1
```

выполнялся за время  $48.4N \mu\text{s}$ , в то время как фрагмент

```
for I := 1 to N do X[I] := 1.0
```

— за время  $5.4N \mu\text{s}$ . Оба фрагмента присваивают элементам массива действительных чисел одно и то же значение — 1. Однако код, сгенерированный для первого фрагмента,

содержал вызов программы времени исполнения для преобразования целого представления 1 в действительное. Поскольку тип X известен в процессе компиляции, было бы разумнее выполнить это преобразование во время компиляции.

## 6.5. Перегрузка функций и операторов

*Перегруженным* (overloaded) называется символ, имеющий различные значения в зависимости от его контекста. В математике таким символом является, например, +, поскольку в выражении  $A+B$  он имеет разные значения в зависимости от того, являются ли  $A$  и  $B$  целыми, действительными или комплексными числами или, скажем, матрицами. В языке программирования Ada перегруженными являются скобки () — выражение  $A(I)$  может быть I-м элементом массива A, вызовом функции A с аргументом I или явным преобразованием выражения I к типу A.

Перегрузка *разрешаема* (resolved), если значение перегруженного символа определяется однозначно. Например, если + может означать либо целое, либо действительное сложение, то в записи  $x + (i + j)$  формы сложения однозначно определяются по типам x, i и j. Разрешение перегрузки иногда называется *идентификацией операторов*, поскольку при этом определяется, какую операцию означает операторный символ.

В большинстве языков арифметические операторы перегружены. Однако перегрузка арифметических операторов типа + может быть разрешена рассмотрением только операндов данного оператора. Анализ подобен проведенному в семантическом правиле для продукции  $E \rightarrow E_1 \text{ or } E_2$  на рис. 6.9, где тип  $E$  определяется рассмотрением возможных типов  $E_1$  и  $E_2$ .

### Множество возможных типов подвыражения

Не всегда можно разрешить перегрузку одним лишь рассмотрением аргументов функции, как показывает приведенный ниже пример. Подвыражение вместо единственного типа может иметь целое множество допустимых типов. Информация, достаточная для сужения выбора до единственного типа, в языке Ada должна обеспечиваться контекстом.

#### Пример 6.6

В языке программирования Ada одной из стандартных (т.е. встроенных) интерпретаций оператора \* является функция пары целых чисел, дающая целое число. Оператор может быть перегружен путем добавления объявлений наподобие следующих:

```
function "*" (i, j: integer) return complex;
function "*" (x, y: complex) return complex;
```

После этих объявлений возможные типы для \* включают

|         |   |         |   |         |
|---------|---|---------|---|---------|
| integer | × | integer | → | integer |
| integer | × | integer | → | complex |
| complex | × | complex | → | complex |

Предположим, что единственный допустимый тип для 2, 3 и 5 — *integer*. Тогда при приведенных выше объявлениях подвыражение  $3 * 5$  может иметь тип *integer* или *complex* — в зависимости от контекста. Если полное выражение —  $2 * (3 * 5)$ , то типом  $3 * 5$

должен быть *integer*, в силу того что  $*$  может принимать только пару значений *integer* или пару значений *complex*. Если же полное выражение —  $(3*5)*z$  и  $z$  объявлено как *complex*, то  $3*5$  должно иметь тип *complex*.  $\square$

В разделе 6.2 мы полагали, что каждое выражение имеет единственный тип, так что правило проверки типов для функции выглядело следующим образом.

$$E \rightarrow E_1(E_2) \quad \{ E.type := \text{if } E_2.type = s \text{ and } E_1.type = s \rightarrow t \text{ then } t \text{ else type\_error} \}$$

Естественное обобщение этого правила для множества типов показано на рис. 6.10. Единственная операция на рис. 6.10 — применение функции; правила для проверки других операторов в выражениях аналогичны. Возможно несколько объявлений перегруженного идентификатора, поэтому мы полагаем, что запись таблицы символов в состоянии содержать множество возможных типов, которое возвращается при вызове функции *lookup*. Стартовый нетерминал  $E'$  порождает полное выражение (его роль станет ясной ниже).

| ПРОДУКЦИЯ                | СЕМАНТИЧЕСКОЕ ПРАВИЛО                                                                                       |
|--------------------------|-------------------------------------------------------------------------------------------------------------|
| $E' \rightarrow E$       | $E'.types := E.types$                                                                                       |
| $E \rightarrow id$       | $E.types := \text{lookup}(id.entry)$                                                                        |
| $E \rightarrow E_1(E_2)$ | $E.types := \{ t   \text{существует } s \in E_2.types \text{ такое, что } s \rightarrow t \in E_1.types \}$ |

Рис. 6.10. Определение множества возможных типов выражения

Последнее правило на рис. 6.10 гласит, что если  $s$  является одним из типов  $E_2$  и один из типов  $E_1$  может отображать  $s$  на  $t$ , то  $t$  — один из типов  $E_1(E_2)$ . Несоответствие типов при использовании функции приводит к пустому множеству  $E.types$ , условию, которое мы временно используем для сообщения о выявленной ошибке типов.

### Пример 6.7

Кроме иллюстрирования спецификации, приведенной на рис. 6.10, этот пример показывает, как описанный выше подход переносится на другие конструкции. В частности, рассмотрим выражение  $3*5$ . Пусть объявление оператора  $*$  те же, что и в примере 6.6, т.е.  $*$  может отображать пару целых чисел либо в целое число, либо в комплексное — в зависимости от контекста. Множество возможных типов подвыражения  $3*5$  показано на рис. 6.11, где  $i$  и  $c$  — сокращения для *integer* и *complex* соответственно.

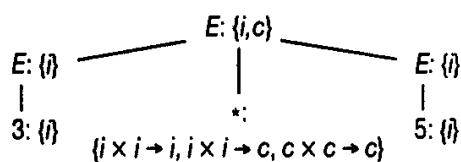


Рис. 6.11. Множество возможных типов выражения  $3*5$

Вновь предположим, что единственным возможным типом для 3 и 5 является *integer*. Оператор  $*$ , таким образом, применяется к паре целых чисел. Если мы рассмотрим эту пару как единую величину, то ее тип — *integer*  $\times$  *integer*. Во множестве типов для  $*$  есть две функции, применимые к парам целых чисел; одна из них возвращает *integer*, вторая — *complex*, так что корень может иметь как тип *integer*, так и *complex*.  $\square$

## Сужение множества возможных типов

Для получения единственного типа язык программирования Ada требует полного выражения. Имея единственный тип из контекста, мы можем сузить выбор типов для каждого подвыражения. Если этот процесс не приводит к единственному типу для каждого подвыражения, то выражение в целом содержит ошибку типов.

Перед тем как перейти от выражений к подвыражениям, мы поближе познакомимся с множествами  $E.types$ , построенными согласно правилам, приведенным на рис. 6.10. Мы покажем, что каждый тип  $t$  в  $E.types$  осуществим (feasible), т.е. из перегруженных типов идентификаторов в  $E$  можно выбрать такие, что  $E$  получит тип  $t$ . Для идентификаторов это свойство обеспечивается объявлением, поскольку каждый элемент  $id.types$  осуществим. В качестве шага индукции рассмотрим тип  $t$  из  $E.types$ , где  $E$  представляет собой  $E_1(E_2)$ . По правилу для функции, приведенному на рис. 6.10, некоторый тип  $s$  должен находиться в  $E.types$ , а тип  $s \rightarrow t$  — в  $E_1.types$ . По индукции  $s$  и  $s \rightarrow t$  являются осуществимыми типами для  $E_2$  и  $E_1$  соответственно. Отсюда следует, что  $t$  — осуществимый тип для  $E$ .

Возможны несколько путей достижения пригодного типа. Например, рассмотрим выражение  $f(x)$ , где  $f$  может иметь типы  $a \rightarrow c$  и  $b \rightarrow c$ , а  $x$  — типы  $a$  и  $b$ . Тогда  $f(x)$  в любом случае имеет тип  $c$ , хотя  $x$  может иметь либо тип  $a$ , либо  $b$ .

Синтаксически управляемое определение на рис. 6.12 получено из представленного на рис. 6.10 путем добавления семантических правил для определения наследуемого атрибута *unique* нетерминала  $E$ . Синтезируемый атрибут *code* нетерминала  $E$  рассматривается ниже.

| ПРОДУКЦИЯ                | СЕМАНТИЧЕСКИЕ ПРАВИЛА                                                                                                                                                                                                                                                                                                                                                                                                                                                                    |
|--------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| $E' \rightarrow E$       | $E'.types := E.types$<br>$E.unique := \text{if } E'.types = \{t\} \text{ then } t \text{ else type\_error}$<br>$E'.code := E.code$                                                                                                                                                                                                                                                                                                                                                       |
| $E \rightarrow id$       | $E.types := \text{lookup}(id.entry)$<br>$E.code := \text{gen}(id.lexeme ':\! E.unique)$                                                                                                                                                                                                                                                                                                                                                                                                  |
| $E \rightarrow E_1(E_2)$ | $E.types := \{s' \mid \text{существует } s \in E_2.types \text{ такое, что } s \rightarrow s' \in E_1.types\}$<br>$t := E.unique$<br>$S := \{s \mid s \in E_2.types \text{ and } s \rightarrow t \in E_1.types\}$<br>$E_2.unique := \text{if } S = \{s\} \text{ then } s \text{ else type\_error}$<br>$E_1.unique := \text{if } S = \{s\} \text{ then } s \rightarrow t \text{ else type\_error}$<br>$E.code := E_1.code \parallel E_2.code \parallel \text{gen}('apply' ':\! E.unique)$ |

Рис. 6.12. Сужение множества типов для выражения

Поскольку полностью выражение порождается  $E'$ , мы хотим, чтобы  $E'.types$  представляло собой множество, содержащее единственный тип  $t$ . Этот тип наследуется как значение  $E.unique$ . Базовый тип *type\_error* вновь сигнализирует о наличии ошибки типов.

Если функция  $E_1(E_2)$  возвращает тип  $t$ , то мы можем найти тип  $s$ , осуществимый для аргумента  $E_2$ ; в то же время тип  $s \rightarrow t$  осуществим для функции. Множество  $S$  в соответствующем семантическом правиле на рис. 6.12 используется для проверки наличия единственного типа  $s$  с таким свойством.

Синтаксически управляемое определение (рис. 6.12) можно реализовать двумя проходами в глубину по синтаксическому дереву для выражения. Во время первого прохода атрибут *types* синтезируется снизу вверх, а при втором — атрибут *unique* распространяется сверху вниз, и при возвращении из узла может быть синтезирован атрибут *code*. На практике программа проверки типов может просто назначить единственный тип каждому узлу синтаксического дерева. На рис. 6.12 мы создаем постфиксную запись в качестве возможного варианта генерации промежуточного кода. В постфиксной записи каждый идентификатор и экземпляр оператора *apply* имеют тип, назначаемый им функцией *gen*.

## 6.6. Полиморфные функции

Инструкции в теле обычной процедуры выполняются с аргументами фиксированных типов; при вызове полиморфной процедуры инструкции ее тела всякий раз могут выполняться с аргументами различных типов. Термин “полиморфный” можно также применить к любой части кода, которая может выполняться с аргументами разных типов, так что мы можем говорить о полиморфизме как функций, так и операторов.

Встроенные операторы для индексирования массивов, применения функций и работы с указателями обычно полиморфны, так как не ограничены определенными типами массивов, функций или указателей. Например, справочное руководство по языку программирования С говорит об операторе указателя & следующее: “Если тип операнда — ‘...’, то тип результата — ‘указатель на ...’”. Поскольку ‘...’ можно заменить любым типом, оператор & в С полиморфен.

В Ada “обобщенные” (*generic*) функции полиморфны, но полиморфизм в этом языке имеет ограниченный характер. Поскольку термин “обобщенный” используется для указания на перегруженные функции и для неявного приведения аргументов функций, мы будем избегать использования этого термина.

Этот раздел посвящен проблеме, возникающей при разработке программы проверки типов для языка с полиморфными функциями. Для работы с полиморфизмом мы расширим наше множество выражений типа для включения переменных типа. Включение переменных типа вызывает некоторые алгоритмические проблемы, связанные с эквивалентностью выражений типа.

### Почему используются полиморфные функции

Полиморфные функции очень привлекательны, поскольку облегчают реализацию алгоритмов, работающих со структурами данных независимо от типов элементов этих структур. Например, удобно иметь подпрограмму определения длины списка, работающую без информации о типах элементов списка.

```
type link = ^cell;
 cell = record
 info: integer;
 next: link;
 end;

function length(lptr: link) : integer;
var len : integer;
begin
 len := 0;
```

```

while lptr <> nil do begin
 len := len + 1;
 lptr := lptr^.next
end;
length := len
end;

```

*Рис. 6.13. Программа определения длины списка на языке Pascal*

Языки типа Pascal требуют полного определения типов параметров функций, так что функция для определения длины связанного списка целых чисел не может применяться к списку действительных чисел. Код на рис. 6.13 применим только к спискам целых чисел. Функция `length` проходит по всем связям `next` элементов списка, пока не будет достигнуто значение `nil`. Хотя функция никоим образом не зависит от типа информации, хранящейся в элементе списка, Pascal требует, чтобы тип поля `info` был объявлен при написании функции `length`.

```

fun length(lptr) =
 if null(lptr) then 0
 else length(tl(lptr)) + 1;

```

*Рис. 6.14. Программа определения длины списка на языке ML*

В языке с полиморфными функциями, наподобие ML ([317]), функция `length` может быть записана таким образом, чтобы быть применимой к любому виду списка (рис. 6.14). Функции `null` и `tl` предопределены; `null` проверяет, пуст ли список, а `tl` возвращает остаток списка, после того как из него удален первый элемент. При использовании определения, показанного на рис. 6.14, оба вызова функции, показанные ниже, дают значение 3.

```

length(["sun", "mon", "tue"]);
length([10, 9, 8]);

```

В первом случае `length` применяется к списку строк, а во втором — к списку целых чисел.

## Переменные типа

Переменные, представляющие выражения типа, позволяют нам говорить о неизвестных типах. В оставшейся части этого раздела греческие буквы  $\alpha$ ,  $\beta$ , ... представляют переменные типа в выражениях типа.

Важное применение переменных типа состоит в проверке согласованного использования идентификаторов в языке, который не требует, чтобы идентификаторы были описаны до их использования. Переменная представляет тип необъявленного идентификатора. Просматривая программу, мы можем увидеть, например, что один и тот же идентификатор используется как целое число в одной инструкции и как массив — в другой. Такая несогласованность может приводить к сообщению об ошибке. Однако если переменная всегда используется как целое число, то мы можем не только убедиться в последовательности ее использования, но и прийти к заключению о ее типе.

*Выведение типа* (type inference) представляет собой задачу определения типа языковой конструкции по способу ее использования. Этот термин часто применим к задаче определения типа функции по ее телу.

### Пример 6.8

Технологии выведения типа могут быть применены к программам на языках, подобных С или Pascal, для замещения недостающей информации о типах во время компиляции. Фрагмент кода на рис. 6.15 показывает процедуру `mlist` с параметром `p`, который сам является процедурой. Все, что мы знаем из первой строки процедуры `mlist` — то, что `p` является процедурой; в частности, мы не знаем количества и типов аргументов, получаемых этой процедурой. Такая неполная спецификация типа `p` допускается языками С и Pascal.

```
type link = ↑cell;

procedure mlist(lptr : link; procedure p);
begin
 while lptr <> nil do begin
 p(lptr);
 lptr := lptr↑.next
 end
end;
```

Рис. 6.15. Процедура `mlist` с параметром-процедурой `p`

Процедура `mlist` применяет параметр `p` к каждой ячейке связанного списка. Например, `p` может использоваться для инициализации или печати целых чисел, хранящихся в ячейках списка. Несмотря на то что типы аргументов `p` не определены, из использования `p` в выражении `p(lptr)` мы можем сделать вывод, что тип `p` должен быть `link → void`.

Любой вызов `mlist` с параметром-процедурой, тип которой не соответствует указанному, является ошибкой. Процедуру можно рассматривать как функцию, которая не возвращает значение, т.е. с типом результата `void`. □

Технологии выведения типа и проверки типов имеют много общего. В каждом случае мы имеем дело с выражениями типа, содержащими переменные. Рассуждения, подобные приведенным в следующем примере, используются позже в этом разделе программой проверки типов для вывода типов, представленных переменными.

### Пример 6.9

В приведенной далее псевдопрограмме тип полиморфной функции `deref` (которая выполняет те же действия, что и оператор разыменования `↑`) можно определить следующим образом.

```
function deref(p);
begin
 return p↑
end;
```

При рассмотрении первой строки `function deref(p);` мы ничего не знаем о типе `p`, так что представим его переменной типа  $\beta$ . По определению постфиксный оператор `↑` получает указатель на объект и возвращает объект. Из того, что оператор `↑` применяется к `p` в выражении `p↑`, следует, что `p` должен быть указателем на объект неизвестного типа  $\alpha$ . В результате получаем, что  $\beta = \text{pointer}(\alpha)$ , где  $\alpha$  — еще одна переменная типа. Более того, выражение `p↑` имеет тип  $\alpha$ , так что для типа функции `deref` мы можем записать выражение типа

для любого типа  $\alpha$   $\text{pointer}(\alpha) \rightarrow \alpha$ .

(6.3)

□

## Язык с полиморфными функциями

Все, что мы пока сказали о полиморфных функциях, — это то, что они могут быть выполнены с аргументами “различного типа”. Точные утверждения о множестве типов, к которым применима полиморфная функция, записываются с использованием символа  $\forall$ , который означает “для любого типа”. Следовательно,

$$\forall \alpha. \text{pointer}(\alpha) \rightarrow \alpha \quad (6.4)$$

представляет собой запись выражения типа (6.3) для типа функции `deref` в примере 6.9. Полиморфная функция `length` на рис. 6.14 получает список элементов некоторого типа и возвращает целое число, поэтому ее тип можно записать как

$$\forall \alpha. \text{list}(\alpha) \rightarrow \text{integer}. \quad (6.5)$$

Здесь `list` представляет конструктор типа. Без символа  $\forall$  мы можем привести только примеры возможных типов областей определения и областей значения функции `length`:

$$\begin{aligned} \text{list}(\text{integer}) &\rightarrow \text{integer} \\ \text{list}(\text{list}(\text{char})) &\rightarrow \text{integer}. \end{aligned}$$

Выражения типа наподобие (6.5) представляют собой наиболее общие формулировки, которые могут быть сделаны о типе полиморфной функции.

Символ  $\forall$  является квантором общности (universal quantifier), а о переменной типа, к которой он применяется, говорят, что она связана (bounded) этим квантором. Связанные переменные могут быть произвольным образом переименованы, лишь бы при этом были переименованы все вхождения данной переменной. Таким образом, выражение типа  $\forall \gamma. \text{pointer}(\gamma) \rightarrow \gamma$  эквивалентно (6.4). Выражение типа с символом  $\forall$  неформально можно назвать “полиморфным типом”.

Язык, который будет использоваться для проверки полиморфных функций, порождается грамматикой на рис. 6.16.

$$\begin{array}{lcl} P & \rightarrow & D ; E \\ D & \rightarrow & D ; D | \text{id} : Q \\ Q & \rightarrow & \forall \text{type\_variable}. Q | T \\ T & \rightarrow & T' \rightarrow' T \\ & | & T \times T \\ & | & \text{unary\_constructor}(T) \\ & | & \text{basic\_type} \\ & | & \text{type\_variable} \\ & | & (T) \\ E & \rightarrow & E(E) | E, E | \text{id} \end{array}$$

Рис. 6.16. Грамматика для языка с полиморфными функциями

Программы, порождаемые этой грамматикой, состоят из последовательности объявлений, за которыми следует проверяемое выражение `E`, например

$$\begin{aligned} \text{deref} : & \forall \alpha. \text{pointer}(\alpha) \rightarrow \alpha; \\ q : & \text{pointer}(\text{pointer}(\text{integer})); \\ \text{deref}(\text{deref}(q)) & \end{aligned} \quad (6.6)$$

Мы минимизируем обозначения тем, что нетерминал  $T$  порождает выражения типов непосредственно. Конструкторы ' $\rightarrow$ ' и  $\times$  образуют типы функции и произведения. Унарные конструкторы, представленные символом `unary_constructor`, позволяют записывать типы наподобие `pointer(integer)` и `list(integer)`. Скобки используются просто для группирования типов. Выражения, типы которых проверяются, имеют очень простой синтаксис: это могут быть идентификаторы, последовательности выражений, образующих кортежи, или функции, применяемые к своим аргументам.

Правила проверки типов для полиморфных функций имеют три отличия от правил для обычных функций из раздела 6.2. Перед представлением этих правил проиллюстрируем различия с помощью выражения `deref(deref(q))` из программы (6.6). Синтаксическое дерево для этого выражения показано на рис. 6.17. Каждому узлу дерева приписаны две метки. Первая указывает подвыражение, представленное этим узлом, а вторая — выражение типа, соответствующее данному подвыражению. Индексы  $o$  и  $i$  указывают внешнее (outer) и внутреннее (inner) вхождения `deref` в рассматриваемом выражении.

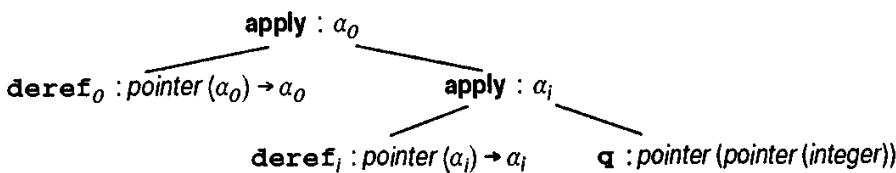


Рис. 6.17. Помеченное синтаксическое дерево для `deref(deref(q))`

Далее представлены отличия от правил для обычных функций.

1. Различные появления полиморфной функции в одном и том же выражении не обязаны иметь аргументы одного и того же типа. В выражении `deref0(derefi(q))` `deref0` снимает один уровень косвенности указателя, так что `deref0` применяется уже к другому типу. Реализация этого свойства основана на интерпретации  $\forall \alpha$  как “для любого типа  $\alpha$ ”. Каждое появление `deref` имеет собственный вид того, что представлено связанный переменной  $\alpha$  в (6.4). Таким образом, каждому появлению `deref` мы присваиваем выражение типа, образованное заменой  $\alpha$  в (6.4) новой переменной, и удаляем при этом квантор  $\forall$ . На рис. 6.17 в выражениях типа, присвоенных внешнему и внутреннему `deref`, такими новыми переменными являются соответственно  $\alpha_o$  и  $\alpha_i$ .
2. Поскольку в выражениях типа могут использоваться переменные, мы должны пересмотреть представление об эквивалентности типов. Предположим, что  $E_1$  типа  $s \rightarrow s'$  применяется к  $E_2$  типа  $t$ . Вместо простого определения эквивалентности  $s$  и  $t$  мы должны их “унифицировать”. Унификация (unification) определена ниже; неформально мы просто определяем, могут ли  $s$  и  $t$  быть сделаны структурно эквивалентными путем замещения переменных типа в выражениях типа для  $s$  и  $t$ . Например, во внутреннем узле, помеченном на рис. 6.17 как `apply`, равенство  

$$\text{pointer}(\alpha_i) = \text{pointer}(\text{pointer}(\text{integer}))$$
истинно, если заменить  $\alpha_i$  выражением `pointer(integer)`.
3. Нам необходим механизм для записи результата унификации двух выражений. Вообще говоря, переменная типа может появляться в нескольких выражениях типа. Если в результате унификации  $s$  и  $s'$  переменная  $\alpha$  представляет тип  $t$ , то  $\alpha$  должна продолжать представлять  $t$  в процессе проверки типов. Например, на рис. 6.17  $\alpha_i$  представляет собой тип области значений `derefi`; так что мы можем использовать

ее для типа  $\text{deref}_i(q)$ . Унификация типа области определения  $\text{deref}_i$  с типом  $q$ , следовательно, воздействует на выражение типа во внутреннем узле, помеченном как **apply**. Другая переменная типа на рис. 6.17 —  $\alpha$  — представляет тип *integer*.

## Подстановки, примеры и унификация

Информация о типах, представленная переменными, формализована определением отображения переменных типа на выражения типа, называемого *подстановкой* (*substitution*). Приведенная далее рекурсивная функция  $\text{subst}(t)$  дает точное представление о применении подстановки  $S$  для замены всех переменных типа в выражении  $t$ . Как обычно, конструктор типа функции рассматривается как “типичный” конструктор.

```
function subst(t : type_expression) : type_expression;
begin
 if t — базовый тип then return t
 else if t — переменная then return S(t)
 else if t является $t_1 \rightarrow t_2$ then
 return subst($t_1 \rightarrow subst(t_2)$)
end
```

Для удобства полученное в результате применения  $\text{subst}(t)$  выражение типа мы записываем как  $S(t)$  и называем *примером* (*instance*)  $t$ . Если подстановка  $S$  не определяет выражение для переменной  $\alpha$ , мы полагаем, что  $S(\alpha)$  просто равно  $\alpha$ ; таким образом,  $S$  представляет собой тождественное отображение для переменных типа.

### Пример 6.10

Здесь для указания того, что  $s$  является примером  $t$ , мы используем запись  $s < t$ .

|                                             |     |                             |
|---------------------------------------------|-----|-----------------------------|
| $\text{pointer}(\text{integer})$            | $<$ | $\text{pointer}(\alpha)$    |
| $\text{pointer}(\text{real})$               | $<$ | $\text{pointer}(\alpha)$    |
| $\text{integer} \rightarrow \text{integer}$ | $<$ | $\alpha \rightarrow \alpha$ |
| $\text{pointer}(\alpha)$                    | $<$ | $\beta$                     |
| $\alpha$                                    | $<$ | $\beta$                     |

Однако записанные далее выражения типа слева не являются примерами выражений справа (по указанной там же причине).

|                                          |                             |                                             |
|------------------------------------------|-----------------------------|---------------------------------------------|
| $\text{integer}$                         | $\text{real}$               | Подстановка неприменима к базовым типам     |
| $\text{integer} \rightarrow \text{real}$ | $\alpha \rightarrow \alpha$ | Несогласованная подстановка $\alpha$        |
| $\text{integer} \rightarrow \alpha$      | $\alpha \rightarrow \alpha$ | Должны быть заменены все появления $\alpha$ |

Два выражения типа  $t_1$  и  $t_2$  унифицируемы, если существует некоторая подстановка  $S$ , такая, что  $S(t_1) = S(t_2)$ . На практике нас интересует *наиболее общий унификатор*, представляющий собой подстановку, накладывающую наименьшее число ограничений на переменные в выражении. Говоря точнее, наиболее общий унификатор выражений  $t_1$  и  $t_2$  представляет собой подстановку  $S$  со следующими свойствами.

1.  $S(t_1) = S(t_2)$ .
2. Для любой другой подстановки  $S'$ , такой, что  $S'(t_1) = S'(t_2)$ , подстановка  $S'$  представляет собой пример  $S$ , т.е. для любого  $t$   $S'(t)$  является примером  $S(t)$ .

Далее, говоря об унификации, мы будем иметь в виду именно наиболее общий унификатор.

## Проверка полиморфных функций

Правила для проверки выражений, порожденных грамматикой на рис. 6.16, будут записаны с использованием следующих операций над представлением типов в виде графа.

1.  $\text{fresh}(t)$  заменяет связанные переменные в выражении типа  $t$  новыми и возвращает указатель на узел, представляющий полученное выражение типа. Все символы  $\forall$  в  $t$  при этом удаляются.
2.  $\text{unify}(m, n)$  унифицирует выражения типов, представленные узлами, на которые указывают  $m$  и  $n$ . Эта операция имеет побочное действие отслеживания подстановок, делающих эти выражения эквивалентными. Если выражения не унифицируются, считается неуспешным весь процесс проверки типов<sup>5</sup>.

Отдельные листья и внутренние узлы графа типов строятся с использованием операций  $\text{mkleaf}$  и  $\text{mknod}$ , подобных описанным в разделе 5.2. Для каждой переменной типа необходимо наличие отдельного листа, однако другие структурно-эквивалентные выражения не обязаны иметь отдельные узлы.

Операция  $\text{unify}$  основана на следующей формулировке унификации и подстановок, связанной с графиками. Предположим, что узлы  $m$  и  $n$  графа представляют соответственно выражения  $e$  и  $f$ . Мы говорим, что узлы  $m$  и  $n$  эквивалентны по отношению к подстановке  $S$ , если  $S(e) = S(f)$ . Задача поиска наиболее общего унификатора  $S$  может быть переформулирована как задача группирования по множествам узлов, которые должны быть эквивалентны по отношению к  $S$ . Для эквивалентности выражений требуется эквивалентность их корней. Кроме того, два узла  $m$  и  $n$  эквивалентны тогда и только тогда, когда они представляют один и тот же оператор и их соответствующие потомки эквивалентны.

Алгоритм унификации пары выражений отслеживает множества узлов, эквивалентных по отношению к возникающим подстановкам. Отложим его рассмотрение до следующего раздела.

```
 $E \rightarrow E_1 (E_2) \quad \{ p := \text{mkleaf}(\text{newtypevar});$
 $\text{unify}(E_1.\text{type}, \text{mknod}(\rightarrow, E_2.\text{type}, p));$
 $E.\text{type} := p \}$
 $E \rightarrow E_1 , E_2 \quad \{ E.\text{type} := \text{mknod}(\times, E_1.\text{type}, E_2.\text{type}) \}$
 $E \rightarrow \text{id} \quad \{ E.\text{type} := \text{fresh}(\text{id.type}) \}$
```

Рис. 6.18. Схема трансляции для проверки полиморфных функций

Правила проверки типов для выражений показаны на рис. 6.18. Мы не показываем обработку объявлений. В процессе проверки выражений типа, порожденных нетерминалами  $T$  и  $Q$ ,  $\text{mkleaf}$  и  $\text{mknod}$  добавляют узлы к графу типов, следуя построению дага в разделе 5.2. При объявлении идентификатора тип из объявления сохраняется в таблице символов в виде указателя на узел, представляющий тип. На рис. 6.18 этот указатель представлен в виде синтезируемого атрибута  $\text{id.type}$ . Как упоминалось выше, операция  $\text{fresh}$  удаляет символ  $\forall$  при замене связанных переменных новыми. Действие, связанное с продукцией  $E \rightarrow E_1 , E_2$ , устанавливает  $E.\text{type}$  равным произведению типов  $E_1$  и  $E_2$ .

<sup>5</sup> Причина прекращения процесса проверки типов заключается в том, что побочное действие некоторых унификаций может быть записано до обнаружения неудачи при проверке. Восстановление после ошибки может быть реализовано, если побочные действия операции  $\text{unify}$  откладываются до тех пор, пока выражения не будут успешно унифицированы.

Правило проверки типов для функции  $E \rightarrow E_1(E_2)$  объясняется рассмотрением случая, когда  $E_1.type$  и  $E_2.type$  являются переменными типа, например,  $E_1.type = \alpha$  и  $E_2.type = \beta$ . Здесь  $E_1.type$  должно быть функцией, такой, что для некоторого неизвестного типа  $\gamma$  мы имеем  $\alpha = \beta \rightarrow \gamma$ . На рис. 6.18 создается новая переменная типа, соответствующая  $\gamma$ , и  $E_1.type$  унифицируется с  $E_2.type \rightarrow \gamma$ . Каждый вызов `newtypevar` возвращает новую переменную типа, лист для которой строится вызовом `mkleaf`, а узел, представляющий функцию, унифицируемую с  $E_1.type$ , — вызовом `mknoden`. После успешной унификации результирующий тип представлен новым листом.

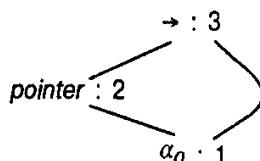
Правила на рис. 6.18 будут детально проиллюстрированы на небольшом примере. Мы представим работу алгоритма записью выражений типа для каждого подвыражения, как показано на рис. 6.19. При применении каждой функции операция `unify` может иметь побочное действие, которое выражается в записи выражения типа для некоторых переменных типа. Такие побочные действия представлены в столбце подстановок на рис. 6.19.

| ВЫРАЖЕНИЕ :           | ТИП                                      | ПОДСТАНОВКА                   |
|-----------------------|------------------------------------------|-------------------------------|
| $q$                   | $pointer(pointer(integer))$              |                               |
| $deref_i$             | $pointer(\alpha_i) \rightarrow \alpha_i$ |                               |
| $deref_i(q)$          | $pointer(integer)$                       | $\alpha_i = pointer(integer)$ |
| $deref_o$             | $pointer(\alpha_o) \rightarrow \alpha_o$ |                               |
| $deref_o(deref_i(q))$ | $integer$                                | $\alpha_o = integer$          |

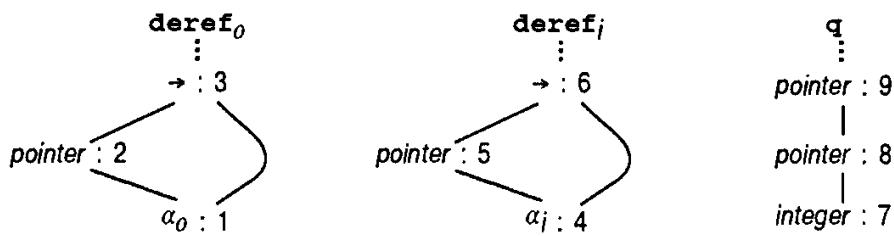
Рис. 6.19. Сводка восходящего определения типа

### Пример 6.11

Проверка типа выражения  $deref_o(deref_i(q))$  в программе (6.6) производится снизу вверх, от листвьев. Здесь, как и ранее, индексы  $o$  и  $i$  относятся к внешнему и внутреннему появлению `deref`. При рассмотрении подвыражения  $deref_o$  функция `fresh` строит следующие узлы с использованием новой переменной типа  $\alpha_o$ .

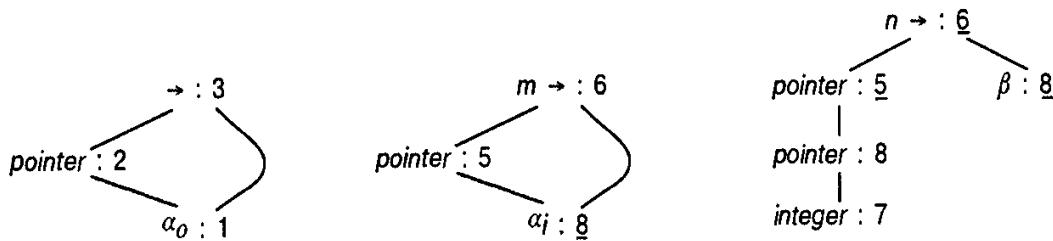


Число возле узла указывает класс эквивалентности, которому принадлежит данный узел. Ниже показана часть графа типов для трех идентификаторов. Пунктирные линии указывают, что узлы с номерами 3, 6 и 9 соответствуют  $deref_o$ ,  $deref_i$  и  $q$ .



Применение функции  $deref_i(q)$  проверяется с помощью построения узла  $n$  для функции, действующей из типа  $q$  в новую переменную типа  $\beta$ . Эта функция успешно унифицируется с типом  $deref_i$ , представленным узлом  $m$  на рисунке ниже. До унифика-

ции узлов  $m$  и  $n$  каждый узел имеет свой номер. После унификации эквивалентные узлы оказываются с одним и тем же номером; измененные номера на рисунке подчеркнуты.



Заметьте, что узлы для  $\alpha_i$  и  $pointer(integer)$  имеют один и тот же номер 8, т.е.  $\alpha_i$  унифицировано с этим выражением типа, как показано на рис. 6.19. Следовательно,  $\alpha_0$  унифицировано с  $integer$ .  $\square$

Следующий пример связывает выведение типа полиморфной функции в ML с правилами проверки типов на рис. 6.18. Синтаксис определения функции в ML задается следующим образом.

```
fun id0 (id1, ..., idk) = E;
```

Здесь  $id_0$  представляет имя функции, а  $id_1, \dots, id_k$  — ее параметры. Для простоты полагаем, что синтаксис выражения  $E$  такой же, как на рис. 6.16, и что идентификаторами в  $E$  могут быть только имя функции, ее параметры и встроенные функции.

Такой подход представляет собой формализацию подхода из примера 6.9, где выводился полиморфный тип для `deref`. Для имени функции и ее параметров создаются новые переменные типа. Встроенные функции, вообще говоря, имеют полиморфные типы; любые переменные типа, появляющиеся в этих типах, связаны квантором  $\forall$ . Затем мы проверяем соответствие типов выражений  $id_0(id_1, \dots, id_k)$  и  $E$ . При успешной проверке соответствия выводим тип для имени функции. И наконец, любые переменные в выведенном типе связаны кванторами  $\forall$  для придания функции полиморфного типа.

### Пример 6.12

Вспомним ML-функцию определения длины списка на рис. 6.14.

```
fun length(lptr) =
 if null(lptr) then 0
 else length(tl(lptr)) + 1;
```

Переменные типов  $\beta$  и  $\gamma$  введены для типов `length` и `lptr` соответственно. Мы обнаруживаем, что тип функции `length(lptr)` соответствует типу выражения, обраzuющего тело функции, и переменная `length` должна иметь тип  $\forall\alpha.list(\alpha) \rightarrow integer$ .

Более детально это показано в программе на рис. 6.20, к которой применяются правила проверки типов, приведенные на рис. 6.18. Объявления в программе связывают новые переменные типа  $\beta$  и  $\gamma$  с `length` и `lptr` и делают типы встроенных операций явными. Мы используем стиль записи из рис. 6.16 с применением полиморфного оператора `if` к трем operandам, представляющим проверяемое выражение, а также части `then` и `else`. Объявление гласит, что части `then` и `else` могут быть любого соответствующего типа, который также является типом результата.

```
length : β;
lptr : γ;
if : ∀α. boolean × α × α → α
```

```

· null : $\forall \alpha. \text{list}(\alpha) \rightarrow \text{boolean}$
 tl : $\forall \alpha. \text{list}(\alpha) \rightarrow \text{list}(\alpha)$
 0 : integer;
 1 : integer;
 + : integer \times integer \rightarrow integer;
match : $\forall \alpha. \alpha \times \alpha \rightarrow \alpha$;
match (
 length(lptr),
 if(null(lptr), 0, length(tl(lptr))+1)
)

```

Рис. 6.20. Объявления с проверяемыми выражениями

Ясно, что функция `length(lptr)` должна иметь тот же тип, что и тело функции; эта проверка кодируется с помощью оператора `match`. Использование `match` позволяет выполнять все проверки в стиле программы на рис. 6.16.

Результат применения правил проверки типов (рис. 6.18) к программе на рис. 6.20 представлен на рис. 6.21. Новые переменные, вводимые операцией `fresh`, которая применяется к полиморфным типам встроенных операций, отличаются нижними индексами у  $\alpha$ . Из строки (3) мы узнаем, что `length` должна быть функцией из  $\gamma$  в некоторый неизвестный тип  $\delta$ . Затем при проверке подвыражения `null(lptr)` в строке (6) мы находим, что  $\gamma$  унифицируется с  $\text{list}(\alpha_n)$ , где  $\alpha_n$  представляет собой неизвестный тип. В этот момент мы знаем, что тип `length` должен быть  $\forall \alpha_n. \text{list}(\alpha_n) \rightarrow \delta$ .

| СТРОКА | ВЫРАЖЕНИЕ          | : | ТИП                                                                   | ПОДСТАНОВКА                         |
|--------|--------------------|---|-----------------------------------------------------------------------|-------------------------------------|
| (1)    | lptr               | : | $\gamma$                                                              |                                     |
| (2)    | length             | : | $\beta$                                                               |                                     |
| (3)    | length(lptr)       | : | $\delta$                                                              | $\beta = \gamma \rightarrow \delta$ |
| (4)    | lptr               | : | $\gamma$                                                              |                                     |
| (5)    | null               | : | $\text{list}(\alpha_n) \rightarrow \text{boolean}$                    |                                     |
| (6)    | null(lptr)         | : | $\text{boolean}$                                                      | $\gamma = \text{list}(\alpha_n)$    |
| (7)    | 0                  | : | integer                                                               |                                     |
| (8)    | lptr               | : | $\text{list}(\alpha_n)$                                               |                                     |
| (9)    | tl                 | : | $\text{list}(\alpha_t) \rightarrow \text{list}(\alpha_t)$             |                                     |
| (10)   | tl(lptr)           | : | $\text{list}(\alpha_n)$                                               | $\alpha_t = \alpha_n$               |
| (11)   | length             | : | $\text{list}(\alpha_n) \rightarrow \delta$                            |                                     |
| (12)   | length(tl(lptr))   | : | $\delta$                                                              |                                     |
| (13)   | 1                  | : | integer                                                               |                                     |
| (14)   | +                  | : | integer $\times$ integer $\rightarrow$ integer                        |                                     |
| (15)   | length(tl(lptr)) + | : | integer                                                               | $\delta = \text{integer}$           |
|        | 1                  | : |                                                                       |                                     |
| (16)   | if                 | : | $\text{boolean} \times \alpha_i \times \alpha_i \rightarrow \alpha_i$ |                                     |
| (17)   | if(...)            | : | integer                                                               | $\alpha_i = \text{integer}$         |
| (18)   | match              | : | $\alpha_m \times \alpha_m \rightarrow \alpha_m$                       |                                     |
| (19)   | match(...)         | : | integer                                                               | $\alpha_m = \text{integer}$         |

Рис. 6.21. Выведение типа  $\text{list}(\alpha_n) \rightarrow \text{integer}$  для `length`

В конце концов, при проверке сложения в строке (15) — знак сложения для ясности записан между своими аргументами —  $\delta$  унифицируется с *integer*.

Когда проверка полностью выполнена, переменная типа  $\alpha_n$  остается в типе функции *length*. Поскольку никаких предположений относительно  $\alpha_n$  сделано не было, при использовании функции вместо  $\alpha_n$  может быть подставлен любой тип. Таким образом мы делаем ее связанный переменной, и для типа *length* записываем  $\forall \alpha_n. list(\alpha_n) \rightarrow integer$ .  $\square$

## 6.7. Алгоритм унификации

Говоря нестрого, унификация представляет собой задачу определения, можно ли два выражения  $e$  и  $f$  сделать идентичными путем подстановки выражений вместо переменных в  $e$  и  $f$ . Проверка равенства выражений — это специальный случай унификации; если  $e$  и  $f$  содержат только константы, но не переменные, то  $e$  и  $f$  унифицируются тогда и только тогда, когда они идентичны. Алгоритм унификации, предлагаемый в этом разделе, может применяться к графам с циклами; так что его можно использовать для проверки структурной эквивалентности циклических типов<sup>6</sup>.

Унификация была определена в предыдущем разделе с помощью функции  $S$ , называемой подстановкой, отображающей переменные в выражения. Мы записываем  $S(e)$  для выражения, полученного посредством замены каждой переменной  $\alpha$  на  $S(\alpha)$ .  $S$  является унификатором  $e$  и  $f$ , если  $S(e) = S(f)$ . Алгоритм в этом разделе определяет подстановку, представляющую собой наиболее общий унификатор пары выражений.

### Пример 6.13

Рассмотрим два выражения типа.

$$\begin{aligned} ((\alpha_1 \rightarrow \alpha_2) \times list(\alpha_3)) &\rightarrow list(\alpha_2) \\ ((\alpha_3 \rightarrow \alpha_4) \times list(\alpha_3)) &\rightarrow \alpha_5 \end{aligned}$$

Два унификатора —  $S$  и  $S'$  — для данных выражений представляют собой следующее.

| $x$        | $S(x)$           | $S'(x)$          |
|------------|------------------|------------------|
| $\alpha_1$ | $\alpha_3$       | $\alpha_1$       |
| $\alpha_2$ | $\alpha_2$       | $\alpha_1$       |
| $\alpha_3$ | $\alpha_3$       | $\alpha_1$       |
| $\alpha_4$ | $\alpha_2$       | $\alpha_1$       |
| $\alpha_5$ | $list(\alpha_2)$ | $list(\alpha_1)$ |

Эти подстановки отображают  $e$  и  $f$  следующим образом.

$$\begin{aligned} S(e) = S(f) &= ((\alpha_3 \rightarrow \alpha_2) \times list(\alpha_3)) \rightarrow list(\alpha_2) \\ S'(e) = S'(f) &= ((\alpha_1 \rightarrow \alpha_1) \times list(\alpha_1)) \rightarrow list(\alpha_1) \end{aligned}$$

Подстановка  $S$  представляет собой наиболее общий унификатор  $e$  и  $f$ . Заметим, что  $S'(e)$  — экземпляр  $S(e)$ , потому что мы можем подставить  $\alpha_1$  вместо обеих переменных в  $S(e)$ . Однако обратное неверно, поскольку каждое  $\alpha_1$  в  $S'(e)$  должно быть заменено на

<sup>6</sup> В некоторых приложениях является ошибкой унифицировать переменную с выражением, содержащим эту переменную. Алгоритм 6.1 разрешает использовать такие подстановки.

одно и то же выражение; так что мы не можем получить  $S(e)$  подстановкой для переменной  $\alpha_1$  в  $S'(e)$ .  $\square$

Если унифицируемые выражения представлены в виде деревьев, количество узлов дерева для выражения после подстановки  $S(e)$  может быть экспонентой от числа узлов в деревьях для  $e$  и  $f$ , даже если  $S$  — наиболее общий унификатор. Однако такое увеличение размера не является обязательным, если для представления выражений и подстановок используются графы, а не деревья.

Мы реализуем теоретико-графовую формулировку унификации, представленную в предыдущем разделе. Задача заключается в том, что узлы, которые должны быть эквивалентны по отношению к наиболее общему унификатору двух выражений, группируются в множества. В примере 6.13 эти выражения представлены двумя узлами, помеченными на рис. 6.22 как  $\rightarrow : 1$ . Целые числа в узлах указывают классы эквивалентности, которым принадлежат узлы после унификации узлов с номером 1. Эти классы эквивалентности имеют то свойство, что все внутренние узлы класса относятся к одному и тому же оператору. Соответствующие потомки внутренних узлов в классе эквивалентности также эквивалентны.

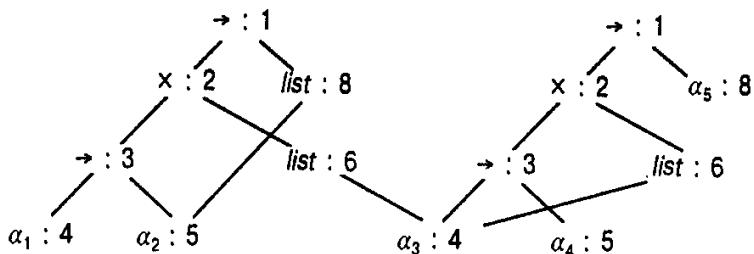


Рис. 6.22. Классы эквивалентности после унификации

### Алгоритм 6.1. Унификация пар узлов в графе

**Вход.** Граф и пара унифицируемых узлов  $t$  и  $n$ .

**Выход.** Логическое значение “истина”, если выражения, представленные узлами  $t$  и  $n$ , унифицируемы; в противном случае — “ложь”. Версия операции *unify*, необходимая для правил проверки типов на рис. 6.18, может быть получена, если функцию из этого алгоритма модифицировать так, чтобы вместо возврата значения “ложь” она завершалась неуспешно.

**Метод.** Узлы представляются записями формата, показанного на рис. 6.23, с полями для бинарного оператора и указателями на левого и правого потомков.

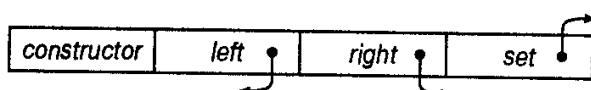


Рис. 6.23. Структура данных для представления узла

Множества эквивалентных узлов поддерживаются с помощью поля *set*. Один из узлов в каждом классе эквивалентности выбирается для представления класса путем установки его указателя *set* равным нулевому указателю. Поля *set* остальных узлов из класса эквивалентности указывают — непосредственно или опосредованно через другие узлы множества — на этот представительский узел. Изначально каждый узел  $n$  находится в собственном классе эквивалентности и является представительским узлом множества, состоящего из одного этого узла.

```

function unify(m, n: node) : boolean;
begin
 s := find(m);
 t := find(n);
 if s = t then
 return true
 else if s и t – узлы, представляющие один
 и тот же базовый тип then
 return true
 else if s представляет собой op-узел
 с потомками s1 и s2 and
 t представляет собой op-узел
 с потомками t1 и t2 then begin
 union(s, t);
 return unify(s1, t1) and unify(s2, t2)
 end
 else if s или t представляют переменную then begin
 union(s, t);
 return true
 end
 else return false
 /* внутренние узлы с разными операторами
 не могут быть унифицированы */
end

```

*Рис. 6.24. Алгоритм унификации*

Алгоритм унификации, приведенный на рис. 6.24, использует две следующие операции над узлами.

1. *find(n)* возвращает представительский узел класса эквивалентности, содержащего узел *n*.
2. *union(m, n)* объединяет классы эквивалентности, содержащие узлы *m* и *n*. Если один из представительских узлов классов эквивалентности *m* и *n* не соответствует переменной, *union* делает его представительским узлом объединенного класса эквивалентности; в противном случае представительским узлом становится один из представительских узлов объединяемых множеств. Эта асимметрия в определении *union* существенна, поскольку переменная не может использоваться в качестве представителя класса эквивалентности для выражения, содержащего конструктор типа или базовый тип. В противном случае посредством переменной могут оказаться унифицированы два неэквивалентных выражения.

Операция *union* над множествами реализуется простым изменением поля *set* представительского узла одного класса эквивалентности таким образом, чтобы он указывал на представительский узел другого класса. Чтобы найти класс эквивалентности, которому принадлежит данный узел, мы следуем по цепочке указателей *set* до тех пор, пока не достигнем представительского узла (у которого указатель *set* имеет нулевое значение).

Заметим, что алгоритм на рис. 6.24 использует *s = find(m)* и *t = find(n)*, а не *m* и *n*. Представительские узлы *s* и *t* эквивалентны, если *m* и *n* находятся в одном классе эквивалентности. Если *s* и *t* представляют один и тот же базовый тип, вызов

*unify(m, n)* возвращает **true**. Если *s* и *t* являются внутренними узлами бинарного конструктора типа, мы наудачу объединяем их классы эквивалентности и производим рекурсивную проверку эквивалентности их соответствующих потомков. Поскольку вначале выполняется объединение, мы уменьшаем количество классов эквивалентности перед рекурсивной проверкой потомков, так что алгоритм рано или поздно завершает свою работу.

Подстановка выражения для переменной реализуется путем добавления листа для переменной в класс эквивалентности, содержащий узел для выражения. Пусть либо *m*, либо *n* представляют собой лист для переменной, который был помещен в класс эквивалентности, содержащий узел, который представлял выражение с конструктором типа или базовым типом. Тогда *find* вернет представительский узел, отражающий этот конструктор типа или базовый тип, с тем, чтобы переменная не могла быть унифицирована с двумя различными выражениями.

### Пример 6.14

На рис. 6.25 показан начальный даг для двух выражений из примера 6.13. Все узлы пронумерованы, и каждый из них относится к собственному классу эквивалентности. При вычислении *unify(1, 9)* алгоритм обнаруживает, что узлы 1 и 9 представляют один и тот же оператор, и объединяет их в один класс эквивалентности, после чего вызывает *unify(2, 10)* и *unify(8, 14)*. Результат вычисления *unify(1, 9)* представляет собой граф, показанный ранее на рис. 6.22.  $\square$

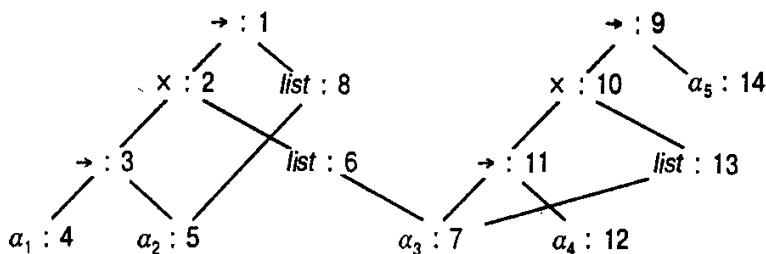


Рис. 6.25. Начальный даг с узлами, принадлежащими собственным классам эквивалентности

Если алгоритм 6.1 возвращает **true**, мы можем построить подстановку *S*, которая действует как унификатор. Пусть каждый узел *n* в полученном графе представляет выражение, связанное с *find(n)*. Таким образом, для каждой переменной  $\alpha$  *find( $\alpha$ )* дает узел *n*, являющийся представительским узлом класса эквивалентности  $\alpha$ . Выражение, представленное узлом *n*, — *S( $\alpha$ )*. Например, на рис. 6.22 мы видим, что представительский узел для  $\alpha_3$  — номер 4, который представляет  $\alpha_1$ . Представительским узлом для  $\alpha_5$  является узел 8, представляющий *list( $\alpha_2$ )*.

### Пример 6.15

Алгоритм 6.1 можно использовать для проверки структурной эквивалентности двух выражений типа

$$\begin{aligned} e &: \text{real} \rightarrow e \\ f &: \text{real} \rightarrow (\text{real} \rightarrow f) \end{aligned}$$

Графы типов для этих выражений показаны на рис. 6.26 (для удобства все узлы пронумерованы).

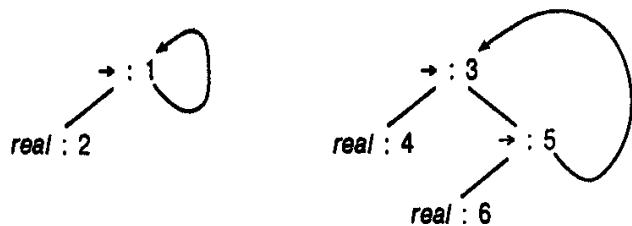


Рис. 6.26. Граф для двух циклических типов

Мы вызываем `unify(1, 3)` для проверки структурной эквивалентности этих двух выражений. Алгоритм объединяет узлы 1 и 3 в один класс эквивалентности и рекурсивно вызывает `unify(2, 4)` и `unify(1, 5)`. Поскольку 2 и 4 представляют один и тот же базовый тип, вызов `unify(2, 4)` вернет `true`. Вызов `unify(1, 5)` добавляет 5 к классу эквивалентности 1 и 3 и рекурсивно вызывает `unify(2, 6)` и `unify(1, 3)`.

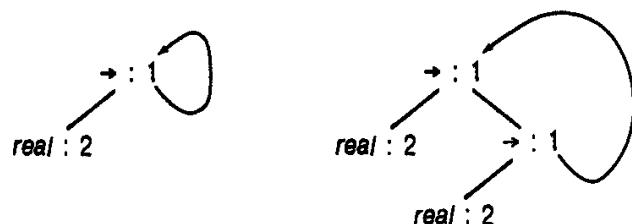


Рис. 6.27. Граф типа, показывающий классы эквивалентности узлов

Вызов `unify(2, 6)` возвращает `true`, поскольку 2 и 6 также представляют один и тот же базовый тип. Второй вызов `unify(1, 3)` немедленно завершается, поскольку узлы 1 и 3 уже объединены в один класс эквивалентности. После этого алгоритм прекращает работу, возвращая значение `true`, показывающее, что два начальных выражения в действительности эквивалентны. На рис. 6.27 представлены полученные классы эквивалентности узлов, где узлы с одинаковыми номерами принадлежат одному классу эквивалентности. □

## Упражнения

### 6.1. Запишите выражения типа для следующих типов.

- Массив указателей на действительные числа, индекс которого лежит в диапазоне от 1 до 100.
- Двухмерный массив целых чисел (т.е. массив массивов), строки которого индексированы от 0 до 9, а столбцы — от -10 до 10.
- Функции, области определения которых представляют собой функции от целых чисел, возвращающие указатели на целые числа, а области значений — записи, состоящие из целого числа и символа.

### 6.2. Предположим, что у нас имеется следующее объявление С.

```
typedef struct {
 int a, b;
} CELL, *PCELL;
CELL foo[100];
PCELL bar(x,y) int x; CELL y { ... }
```

Запишите выражения типов `foo` и `bar`.

- 6.3. Следующая грамматика определяет списки списков букв. Интерпретация символов здесь такая же, как и в случае грамматики на рис. 6.3, с добавлением типа *list*, который указывает список элементов следующего за ним типа *T*.

$$\begin{array}{lcl} P & \rightarrow & D ; E \\ D & \rightarrow & D ; D \mid id : T \\ T & \rightarrow & list \ of \ T \mid char \mid integer \\ E & \rightarrow & (L) \mid literal \mid num \mid id \\ L & \rightarrow & E , L \mid E \end{array}$$

Запишите правила трансляции, аналогичные правилам из раздела 6.2, для определения типов выражений (*E*) и списков (*L*).

- 6.4. Добавьте к грамматике из упр. 6.3 продукцию  $E \rightarrow nil$ , означающую, что выражение может быть пустым списком. Пересмотрите правила из ответа к упр. 6.2 с учетом того, что *nil* может означать пустой список элементов любого типа.
- 6.5. Используя схему трансляции из раздела 6.2, определите типы выражений в следующих программных фрагментах. Укажите типы в каждом узле дерева разбора.
- a)  $c: char; i: integer;$   
 $c \ mod \ i \ mod \ 3$
  - b)  $p: \uparrow integer; a: array [10] \ of \ integer;$   
 $a[p\uparrow]$
  - c)  $f: integer \rightarrow boolean;$   
 $i: integer; j: integer; k: integer;$   
 $\text{while } f(i) \text{ do}$   
 $k := i;$   
 $i := j \ mod \ i;$   
 $j := k$
- 6.6. Модифицируйте схему трансляции для проверки выражений из раздела 6.2 так, чтобы при обнаружении ошибки выводилось описывающее ее сообщение и проверка продолжалась, как если бы требуемый тип был обнаружен.
- 6.7. Перепишите правила проверки типов из раздела 6.2 так, чтобы они ссылались на узлы в представлении выражений типов в виде графа. Переписанные правила должны использовать структуры данных и операции, поддерживаемые таким языком программирования, как Pascal. Используйте структурную эквивалентность выражений типов в следующих случаях.
- a) Выражения типов представлены деревьями, как на рис. 6.2.
  - b) Граф типов представляет собой даг с отдельным узлом для каждого выражения типа.
- 6.8. Измените схему трансляции (рис. 6.5) для обработки следующих конструкций.
- a) Инструкции, имеющие значения. Значение присвоения представляет собой значение выражения справа от знака  $:=$ . Значение условного выражения или инструкции *while* представляет собой значение тела инструкции; значение списка инструкций является значением последней инструкции в списке.
  - b) Логические выражения. Добавьте продукцию для логических операторов *and*, *or* и *not*, а также для операторов сравнения (*<* и др.). Затем добавьте соответствующие правила трансляции, вычисляющие типы этих выражений.

- 6.9. Обобщите правила проверки типов для функций, представленные в конце раздела 6.2, для обработки *n*-арных функций.
- 6.10. Предположим, что имена типов *link* и *cell* определены так, как в разделе 6.3. Какие из следующих выражений структурно эквивалентны? Какие — с точки зрения эквивалентности имен?
- link*
  - pointer(cell)*
  - pointer(link)*
  - pointer(record((info×integer)×(next×pointer(cell))))*
- 6.11. Переформулируйте алгоритм для проверки структурной эквивалентности (рис. 6.6) так, чтобы аргументы *sequiv* были указателями на узлы дага.
- 6.12. Рассмотрим кодировку ограниченных выражений типа как последовательности битов в примере 6.1. В работе [217] двубитовые поля для конструкторов появляются в обратном порядке, с полем для внешнего конструктора, следующим за четырьмя битами базового типа, например:

| ВЫРАЖЕНИЕ ТИПА                        | КОДИРОВКА    |
|---------------------------------------|--------------|
| <i>char</i>                           | 000000 0001  |
| <i>freturns(char)</i>                 | 000011 0001  |
| <i>pointer(freturns(char))</i>        | 001101 0001  |
| <i>array(pointer(freturns(char)))</i> | 110110 0001. |

С помощью операторов С напишите код для построения представления *array(t)* по представлению *t* и наоборот, полагая, что кодирование выполняется по правилам

- работы [217];
- примера 6.1.

- 6.13. Предположим, что тип каждого идентификатора — поддиапазон целых чисел. Запишите правила проверки типов для выражений с операторами +, -, \*, div и mod, как в языке Pascal, которые назначают каждому подвыражению поддиапазон, в котором должно находиться его значение.
- 6.14. Разработайте алгоритм проверки эквивалентности типов языка программирования С (см. пример 6.4).
- 6.15. Некоторые языки программирования, наподобие PL/I, приводят логические значения к целым, где 1 обозначает *true*, а 0 — *false*. Например, выражение  $3 < 4 < 5$  группируется как  $(3 < 4) < 5$  и имеет значение *true*, поскольку истинное выражение  $3 < 4$  имеет значение 1, а  $1 < 5$ . Запишите правила трансляции для логических выражений, которые выполняют такое преобразование. Воспользуйтесь при необходимости условными инструкциями промежуточного языка для назначения целых значений переменным, представляющим значения логических выражений.
- 6.16. Обобщите алгоритмы на рис. 6.9а и 6.12б для выражений с конструкторами типов *array*, *pointer* и декартово произведение.
- 6.17. Какие из следующих рекурсивных выражений типов эквивалентны?

$$\begin{aligned} e1 &= \text{integer} \rightarrow e1 \\ e2 &= \text{integer} \rightarrow (\text{integer} \rightarrow e2) \\ e3 &= \text{integer} \rightarrow (\text{integer} \rightarrow e1) \end{aligned}$$

- 6.18.** Используя правила из примера 6.6, определите, какие из приведенных выражений имеют единственные типы (считаем, что  $z$  — комплексное число).
- $1 * 2 * 3$
  - $1 * (z * 2)$
  - $(1 * z) * z$
- 6.19.** Предположим, что мы допускаем преобразования типов из примера 6.6. При каких условиях относительно типов  $a$ ,  $b$  и  $c$  (целых или комплексных) выражение  $(a * b) * c$  будет иметь единственный тип?
- 6.20.** С использованием переменных типа выразите типы следующих функций.
- Функция  $ref$ , которая получает в качестве аргумента объект произвольного типа и возвращает указатель на этот объект.
  - Функция, которая получает в качестве аргумента массив, индексированный целыми числами, с элементами произвольного типа, а возвращает массив, элементы которого представляют собой объекты, на которые указывают элементы данного массива<sup>7</sup>.
- 6.21.** Найдите наиболее общие унификаторы выражений типа.
- $(pointer(\alpha)) \times (\beta \rightarrow \gamma)$
  - $\beta \times (\gamma \rightarrow \delta)$
- Что изменится, если в (ii) заменить  $\delta$  на  $\alpha$ ?
- 6.22.** Найдите наиболее общий унификатор для каждой пары выражений из следующего списка (либо определите, что такового не существует).
- $\alpha_1 \rightarrow (\alpha_2 \rightarrow \alpha_1)$
  - $array(\beta_1) \rightarrow (pointer(\beta_1) \rightarrow \beta_3)$
  - $\gamma_1 \rightarrow \gamma_2$
  - $\delta_1 \rightarrow (\delta_1 \rightarrow \delta_2)$
- 6.23.** Расширьте правила проверки типов из примера 6.6 для использования их с записями. Воспользуйтесь следующим дополнительным синтаксисом для выражений типа и выражений.
- ```

 $T \rightarrow record\ fields\ end$ 
 $E \rightarrow E.\ id$ 
 $fields \rightarrow fields; field | field$ 
 $field \rightarrow id : T$ 

```
- Какие ограничения на определяемые типы возникают из-за отсутствия имен типов?
- *6.24.** Разрешение перегрузки в разделе 6.5 происходит в два этапа: сначала определяется множество возможных типов каждого подвыражения, а затем оно сужается до единственного типа, после того как определен единственный тип всего выражения. Какую структуру данных вы бы использовали для разрешения перегрузки в процессе одного восходящего прохода?

⁷ Под “данным массивом” подразумевается, видимо, еще один массив указателей, создаваемый функцией. — Прим. ред.

*6.25. Разрешение перегрузки усложняется, если объявления идентификаторов необязательны. Предположим, что для перегрузки идентификаторов, представляющих функции, могут использоваться объявления, но при этом все появления необъявленного идентификатора имеют один и тот же тип. Покажите, что задача определения, имеет ли выражение в этом языке корректный тип, является NP-полной. Эта задача возникла при проверке типов в экспериментальном языке Норе (“Надежда”, [69]).

6.26. Следуя примеру 6.12, выведите полиморфный тип для map.

map : $\forall \alpha. \forall \beta. ((\alpha \rightarrow \beta) \times list(\alpha)) \rightarrow list(\beta)$

ML-определение map имеет следующий вид.

```
fun map(f, l) =  
    if null(l) then nil  
    else cons(f(hd(l)), map(f, tl(l)))
```

Типы встроенных идентификаторов в теле функции следующие:

```
null :  $\forall \alpha. list(\alpha) \rightarrow boolean;$   
nil :  $\forall \alpha. list(\alpha);$   
cons :  $\forall \alpha. (\alpha \times list(\alpha)) \rightarrow list(\alpha);$   
hd :  $\forall \alpha. list(\alpha) \rightarrow \alpha;$   
tl :  $\forall \alpha. list(\alpha) \rightarrow list(\alpha).$ 
```

*6.27. Покажите, что алгоритм унификации из раздела 6.7 определяет наиболее общий унификатор.

**6.28. Модифицируйте алгоритм унификации из раздела 6.7 так, чтобы он не унифицировал переменную с выражением, содержащим эту переменную.

**6.29. Предположим, что выражения представлены деревьями. Найдите выражения e и f , такие, что для любого унификатора S количество узлов в $S(e)$ представляет собой экспоненту от числа узлов в e и f .

6.30. Два узла называются *конгруэнтными*, если они представляют эквивалентные выражения. Даже если в исходном графе типов не было конгруэнтных узлов, после унификации они могут появиться.

а) Разработайте алгоритм для объединения класса взаимно конгруэнтных узлов в один узел.

**б) Расширьте алгоритм (а) для объединения конгруэнтных узлов до тех пор, пока не исчерпаются все пары конгруэнтных узлов.

*6.31. Выражение $g(g)$, показанное в строке 9 программы на языке С (рис. 6.28), представляет применение функции к самой себе. Объявление в строке 3 определяет в качестве типа области значений g *integer*, но тип аргумента остается неопределенным. Попытайтесь выполнить программу. Возможно, ваш компилятор выдаст предупреждение, поскольку в строке 2 g объявлена как функция, а не как указатель на функцию⁸.

⁸ В зависимости от реализации компилятора, программа может вывести как “5! = 120”, так и “0! = 120” — последнее в том случае, когда в строке 14 вначале будет выполнена ие передача значения n функции printf, а вычисление f(f), в процессе которого значение глобальной переменной n уменьшается до нуля. — Прим. ред.

```

(1)  int n;
(2)  int f(int g())
(3)  {
(4)      int m;
(5)      m = n;
(6)      if (m == 0) return 1;
(7)      else {
(8)          n = n - 1; return m * g(g);
(9)      }
(10)
(11) main()
(12) {
(13)     n = 5; printf("%d! = %d\n",n,f(f));
(14) }

```

Рис. 6.28. Программа на языке программирования C, содержащая применение функции к самой себе

- Что вы можете сказать о типе g ?
- Используйте правила проверки типов для полиморфных функций (рис. 6.18) для вывода типа g в следующей программе.

```

m : integer;
times : integer × integer → integer;
g : α;
times( m, g(g) )

```

Библиографические примечания

Базовые типы и конструкторы типов в ранних языках наподобие Fortran и Algol 60 были настолько ограничены, что проверка типов не вызывала серьезных проблем. В результате описание проверки типов в их компиляторах оказалось в обсуждении генерации кода для выражений. В работе [401] описана трансляция выражений в первоначальном компиляторе Fortran. Компилятор отслеживал, являлся ли тип выражения целым или действительным, но преобразования типов не были разрешены. Бэкус в работе [40] вспоминает, что такое решение было первоначально принято из-за сложности правил обработки выражений с операндами смешанных типов. Работа [329] представляет собой одну из ранних статей о проверке типов в компиляторе Algol; используемые при этом технологии сходны с методами, рассмотренными в разделе 6.2.

Возможности структурирования данных, такие как массивы и записи, были предсказаны еще в 40-х годах ([51]). Одним из первых языков программирования, допускавшим создание выражений типа на систематической основе, был Algol 68. Выражения типа могли быть рекурсивно определены; при этом использовалась структурная эквивалентность. Четкое отличие структурной и именной эквивалентности имеется в EL1; выбор при этом предоставляется программисту ([446]). В критике языка Pascal [453] этому вопросу уделяется особое внимание.

Комбинация преобразования типов и перегрузки может привести к неоднозначности: преобразование типа аргумента может привести к разрешению перегрузки в пользу недекватного алгоритма. В этом случае используются дополнительные ограничения на преобразования или перегрузку. Свободный подход к преобразованию типов был принят

в PL/I, где основным критерием при разработке был следующий: “*Все сойдет*. Если определенная комбинация символов имеет приемлемый смысл, то этот смысл будет признан официально” [359]. Упорядочение зачастую накладывается на множество базовых типов — например, в [186] описана решетчатая структура, наложенная на базовые типы CPL, при которой нижние типы могут приводиться к высшим.

В языках наподобие APL ([208]) и SETL ([394]) разрешение перегрузки в процессе компиляции обеспечивает повышение производительности программ ([48]). В работе [429] различается два метода — разрешения “вперед”, при котором множество возможных типов оператора определяется по его operandам, и разрешения “назад”, основанного на типе, ожидаемом в соответствии с контекстом. Использование решеток типов в работах [226, 231] позволило решить проблему ограничения типов, получаемых с помощью этих методов. Перегрузка в Ada может быть разрешена посредством одного прохода вперед и одного — назад, как описано в разделе 6.5. Это замечание можно найти во многих статьях, например [157, 209, 346, 347]. В [96] предложена рекурсивная реализация, а в [43] явного обратного прохода удалось избежать с помощью дага возможных типов.

Вывод типов изучался Карри [101] в связи с комбинаторной логикой и лямбда-исчислением [82]. Было обнаружено, что лямбда-исчисление является ядром функционального языка. При обсуждении вопросов проверки типов мы неоднократно использовали применение функции к некоторому аргументу. В лямбда-исчислении функции можно определять и применять безотносительно к типам, и Карри интересовал их “функциональный характер” и определение того, что сегодня мы можем назвать наиболее общим полиморфным типом, состоящим из выражения типа с кванторами общности (как в разделе 6.6). Продолжая работу Карри, Хиндли [187] обнаружил, что для вывода типов может применяться унификация. Независимо в работе [322] лямбда-выражениям назначались типы путем определения системы уравнений и решения ее для определения типов, связанных с переменными. Не будучи знаком с работами Хиндли, Милнер [316] также нашел, что для решения системы уравнений может быть применена унификация, и применил эту идею при выводе типов в языке программирования ML.

Проверка типов в ML описана в [71]; этот подход был применен в [313]; в [416] изучено его применение в Smalltalk 1976 ([204]). В [319] показано, каким образом может быть включено приведение типов.

В [322] отмечается, что рекурсивные или циклические типы допускают выведение типов для выражений, содержащих применение функции к себе самой. Программа на языке программирования C, представленная на рис. 6.28, происходит от программы на языке Algol из [279]. Упр. 6.31 взято из [298], где приведена семантическая модель рекурсивных полиморфных типов. Другие подходы к этому вопросу можно найти в [73, 307]. В [368] рассматривается система типов ML, теоретическое руководство по избежанию аномалий, вызванных приведением типов и перегрузкой, и полиморфные функции высшего порядка.

Унификация впервые была изучена в работе [376]. Алгоритм унификации из раздела 6.7 можно легко получить из алгоритмов для проверки эквивалентности конечных автоматов и связанных списков с циклами ([257], раздел 2.3.5, упр. 11). Почти линейный алгоритм для проверки эквивалентности конечных автоматов [191] можно найти в виде наброска реализации в [257]. Линейные алгоритмы для нециклического случая представлены в [301, 343]. Алгоритм поиска конгруэнтных узлов (см. упр. 6.30) приведен в [117].

В [112] описан генератор программ проверки типов, использующий метод соответствия шаблону для создания программы по операционно-семантической спецификации, основанной на правилах выводения.

ГЛАВА 7

Среды времени исполнения

Перед тем как приступить к рассмотрению генерации кода, мы должны связать статический исходный текст программы с действиями, которые должны выполняться для реализации программы. В процессе выполнения одно и то же имя в исходном тексте может означать различные объекты данных в целевой машине. Эта глава рассматривает связь имен и объектов данных.

Выделение и освобождение объектов данных управляется пакетом *поддержки времени исполнения* (run-time support), состоящим из программ, загружаемых с генерируемым целевым кодом. На разработку пакета влияет семантика процедур. Пакеты поддержки для языков типа Fortran, Pascal и Lisp могут быть построены с применением технологий, описываемых в данной главе.

Каждое выполнение процедуры известно как ее *активация*. Если процедура рекурсивна, несколько ее активаций могут существовать одновременно. Каждый вызов процедуры в Pascal приводит к активации, которая может управлять объектами данных, выделенными для ее работы.

Представление объектов данных в процессе работы зависит от их типа. Зачастую элементарные типы данных, такие как символы, целые и действительные числа, могут быть представлены эквивалентными объектами данных целевой машины. Однако составные данные, такие как массивы, строки или структуры, обычно представлены наборами примитивных объектов (этот вопрос рассматривается в главе 8, “Генерация промежуточного кода”).

7.1. Вопросы исходного языка

Для определенности предположим, что программа создается из процедур, как в Pascal. В этом разделе мы различаем исходный текст процедуры и ее активацию во время выполнения.

Процедуры

Определение процедуры (procedure definition) представляет собой объявление, которое, в простейшем виде, связывает идентификатор с инструкцией. Этот идентификатор — *имя процедуры*, а инструкция — *тело процедуры*. Например, код Pascal на рис. 7.1 содержит в строках 3–7 определение процедуры `readarray`; тело процедуры располагается в строках 5–7. Процедуры, возвращающие значения, во многих языках называются *функциями*; однако удобнее говорить о них как о процедурах. Вся программа будет также рассматриваться как процедура¹.

¹ В случае языка программирования С как процедуры, так и функции считаются функциями (процедура рассматривается как функция, возвращающая значение `void`). По сути же, это всего лишь два разных названия одной сущности. — Прим. ред.

процедуры. Это означает, что, как мы вскоре увидим, поток управления между процедурами может быть изображен с использованием деревьев.

Каждое исполнение тела процедуры известно как ее активация. *Время жизни* (lifetime) активации процедуры p представляет собой последовательность шагов между первым и последним шагами в выполнении тела процедуры, включая время выполнения процедур, вызванных процедурой p , процедур, вызванных этими процедурами и т.д. Вообще говоря, термин “время жизни” означает последовательность шагов в процессе выполнения программы.

В языках программирования наподобие Pascal всякий раз, когда управление передается процедуре q из процедуры p , в конечном счете оно возвращается процедуре p (за исключением фатальных ошибок в программе). Более точно, всякий раз, когда управление передается из активации процедуры p активации процедуры q , оно возвращается той же активации p .

Если a и b — активации процедуры, то их времена жизни либо не перекрываются, либо вложены. Таким образом, если управление передано b до того, как оно покинуло a , то управление должно покинуть b до того, как оно покинет a .

Это свойство вложенности времен жизни активаций можно проиллюстрировать вставкой двух инструкций печати в каждую процедуру: одну — до первой инструкции в теле процедуры, а вторую — после последней. Первая инструкция выводит `enter` с последующим именем процедуры и значениями фактических параметров; последняя выводит `leave` с той же информацией. На рис. 7.2 показан выход при выполнении программы, представленной на рис. 7.1, с добавленными инструкциями печати. Время жизни активации `quicksort(1, 9)` представляет собой последовательность шагов между печатью `enter quicksort(1, 9)` и `leave quicksort(1, 9)`. На рис. 7.2 предполагается, что значение, возвращаемое `partition(1, 9)`, равно 4.

```
Execution begins...
enter readarray
leave readarray
enter quicksort(1, 9)
enter partition(1, 9)
leave partition(1, 9)
enter quicksort(1, 3)
...
leave quicksort(1, 3)
enter quicksort(5, 9)
...
leave quicksort(5, 9)
leave quicksort(1, 9)
Execution terminated.
```

Рис. 7.2. Информация об активациях процедур в программе на рис. 7.1

Процедура является *рекурсивной*, если ее новая активация может начаться до того, как завершилась ее же более ранняя активация. На рис. 7.2 показано, что управление передается активации `quicksort(1, 9)` в строке 24, в самом начале выполнения программы, и покидает ее практически в самом конце. В промежутке имеется ряд других активаций процедуры `quicksort`, так что `quicksort` является рекурсивной.

Рекурсивная процедура p не обязательно вызывает сама себя непосредственно; p может вызвать другую процедуру q , которая в свою очередь может вызвать p через некото-

ную последовательность вызовов процедур. Для изображения того, как управление передается активациям и покидает их, рассмотрим так называемое *дерево активации* (activation tree). В таком дереве выполняются следующие условия.

1. Каждый узел представляет активацию процедуры.
2. Корень представляет активацию основной программы.
3. Узел для a является родительским для узла b тогда и только тогда, когда поток управления передается из a в b .
4. Узел для a располагается слева от узла b тогда и только тогда, когда время жизни a начинается раньше времени жизни b .

Поскольку каждый узел представляет единственную активацию и наоборот, удобнее говорить о том, что управление находится в узле, когда оно находится в активации, представленной этим узлом.

Пример 7.1

Дерево активации на рис. 7.3 построено на основании вывода программы, представленного на рис. 7.2². Для экономии места показаны только первые буквы имен процедур. Корень дерева активации соответствует всей программе `sort`. В процессе выполнения `sort` происходит активация `readarray`, представленная первым дочерним узлом (помеченым r) по отношению к корню. Следующая активация, представленная вторым дочерним узлом, соответствует вызову `quicksort` с фактическими параметрами 1 и 9. Во время этой активации вызовы `partition` и `quicksort` в строках 16–18 на рис. 7.1 приводят к активациям $p(1, 9)$, $q(1, 3)$ и $q(5, 9)$. Заметим, что активации $q(1, 3)$ и $q(5, 9)$ рекурсивны, начинаются и заканчиваются до окончания $q(1, 9)$. □

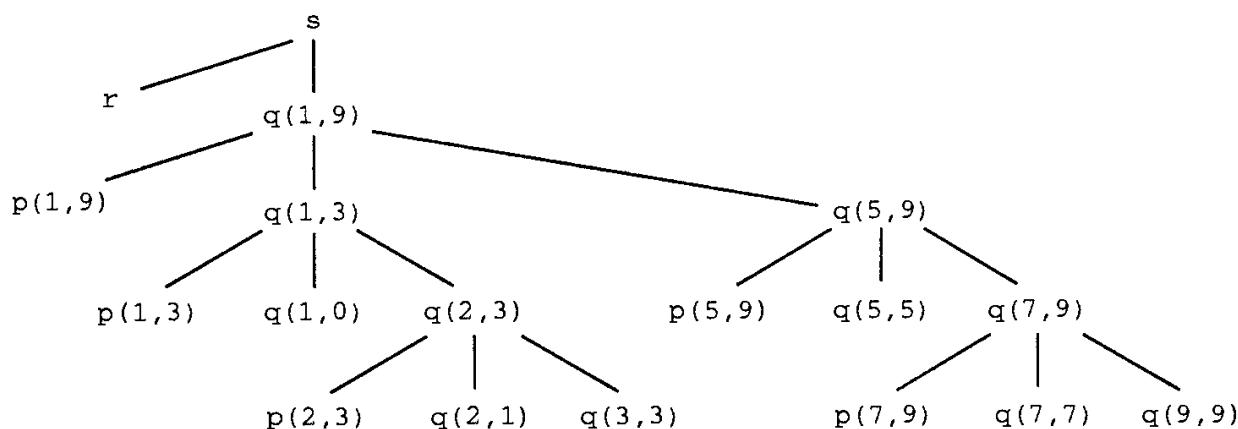


Рис. 7.3. Дерево активации, соответствующее выводу, представленному на рис. 7.2

Стеки управления

Поток управления программы соответствует обходу дерева активации в глубину, который начинается в корне, посещает узел до посещения его потомков, и рекурсивно посещает потомков каждого узла слева направо. Вывод, показанный на рис. 7.2, можно восстановить

² Действительные вызовы, осуществляемые процедурой `quicksort`, зависят от значений, возвращаемых функцией `partition` (подробно алгоритм разобран, например, в [8]). На рис. 7.3 представлено одно из возможных деревьев вызовов, согласованное с выводом, представленным на рис. 7.2 (хотя нижняя часть дерева на рис. 7.2 не показана).

обходом дерева активации, показанного на рис. 7.3, выводя `enter` при первом достижении узла и `leave` — когда в процессе обхода поддерево узла пройдено целиком.

Для отслеживания работающих активаций процедур мы можем использовать так называемый *стек управления* (control stack) (иногда встречается название *стек вызовов*. — Прим. перев.). Идея состоит в помещении в стек узла для активации в ее начале и снятии его со стека по окончании активации. Тогда содержимое стека управления в каждый момент времени указывает путь к корню по дереву активации. Когда на вершине стека управления находится узел *n*, стек содержит узлы на пути от *n* к корню.

Пример 7.2

На рис. 7.4 показаны узлы дерева активации (рис. 7.3), достигнутые при передаче управления активации, представленной $q(2, 3)$. Активации с метками r , $p(1, 9)$, $p(1, 3)$ и $q(1, 0)$ к этому моменту полностью завершены, так что пути к соответствующим узлам показаны на рисунке пунктиром. Сплошные линии отмечают путь от $q(2, 3)$ к корню.

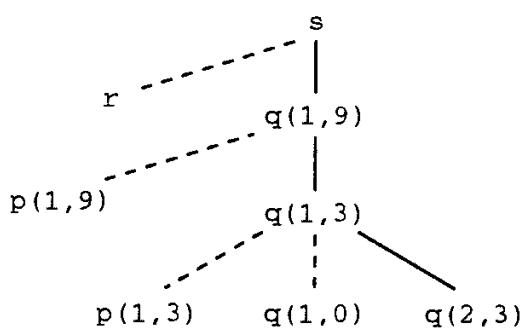


Рис. 7.4. Стек управления содержит узлы вдоль пути к корню

В этот момент стек управления содержит следующие узлы на пути к корню дерева, и только их (вершина стека справа):

s, q(1, 9), q(1, 3), q(2, 3)

Стеки управления расширяются в стеки выделения памяти, используемые при реализации языков программирования типа Pascal и C. Эта технология будет подробно рассматриваться в разделах 7.3 и 7.4.

Область видимости объявления

Объявление (декларация) в языке представляет собой синтаксическую конструкцию, которая связывает информацию с именем. Объявление может быть явным, как в приведенном фрагменте программы на языке Pascal

`var i : integer;`

или неявным. Например, в языке программирования Fortran любое имя переменной, начинающееся с *I*, считается описывающим целое число — если явно не объявлено иное.

В различных частях программы могут быть независимые объявления одного и того же имени. *Правила области видимости* (scope rules) языка определяют, какое именно объявление имени применимо к данному его вхождению в текст программы. В программе на языке программирования Pascal, приведенной на рис. 7.1, *i* объявлено трижды, в строках 4, 9 и 13; использования *i* в процедурах `readarray`, `partition` и `quicksort` не зависит одно от другого. Объявление в строке 4 соответствует использованию *i* в строке 6, т.е. оба вхождения *i* в строке 6 находятся в области видимости объявления в строке 4. Три вхождения *i* в строках 16–18 находятся в области видимости объявления *i* в строке 13.

Часть программы, в которой применимо данное объявление, называется его *областью видимости* (scope). Вхождение имени в процедуре называется *локальным* (local) в данной процедуре, если оно находится в области видимости объявления, расположенного внутри процедуры; в противном случае вхождение имени считается *нелокальным* (nonlocal) (используется также термин *глобальное имя*. — Прим. перев.). Отличие локальных и нелокальных имен распространяется на все синтаксические конструкции, которые могут содержать объявления.

Хотя область видимости является свойством объявления имени, зачастую удобнее использовать название “область видимости имени *x*” вместо “область видимости объявления имени *x*, применимая к данному вхождению *x*”. В этом смысле областью видимости *i* в строке 17 программы на рис. 7.1 является тело процедуры *quicksort*³.

В процессе компиляции для поиска объявления, применимого к данному вхождению имени, может использоваться таблица символов. Когда мы встречаем объявление, для него создается запись в таблице символов, и пока мы находимся в области видимости этого объявления, при поиске имени в таблице символов нам возвращается эта запись. Таблицы символов будут рассматриваться в разделе 7.6.

Связывание имен

Даже если каждое имя объявлено в программе только один раз, одно и то же имя может обозначать во время работы различные объекты данных. Нестрогий термин “объект данных” означает место в памяти для хранения значений.

В семантике языков программирования термин *среда* (environment) означает функцию, отображающую имя в местоположение в памяти, а термин *состояние* (state) — функцию, отображающую местоположение в памяти в хранящееся там значение, как показано на рис. 7.5. Используя термины *l-значение* и *r-значение* из главы 2, можно сказать, что среда отображает имя в *l-значение*, а состояние — *l-значение в r-значение*.



Рис. 7.5. Двухэтапное отображение имен в значения

Среды и состояния отличаются друг от друга; присвоение изменяет состояние, но не среду. Предположим, например, что адрес памяти 100, связанный с переменной *pi*, хранит 0. После присвоения *pi := 3.14* тот же адрес памяти остается связанным с переменной *pi*, но хранящееся в нем значение становится равным 3.14.

Когда среда назначает местоположение в памяти *s* имени *x*, мы говорим о том, что *x* *связано* (bound) с *s*, а назначение само по себе представляет процесс *связывания* (binding) *x*. Термин “местоположение” достаточно фигуранен — если *x* не является базовым типом, память *s* для *x* может представлять собой набор слов памяти.

Связывание представляет собой динамическую часть объявления, как показано на рис. 7.6. Как мы видели, одновременно может выполняться несколько активаций рекурсивной процедуры. В языке программирования Pascal имя локальной переменной в процедуре связывается с различными положениями в памяти при каждой активации процедуры. Технологии связывания имен локальных переменных рассматриваются в разделе 7.3.

³ В большинстве случаев такие термины, как *имя*, *идентификатор*, *переменная* и *лексема*, взаимозаменяемы без каких-либо недоразумений.

СТАТИЧЕСКОЕ ПОНЯТИЕ	ДИНАМИЧЕСКАЯ ЧАСТЬ
Определение процедуры	Активация процедуры
Объявление имени	Связывание имени
Область видимости объявления	Время жизни связывания

Рис. 7.6. Соответствие статического и динамического представлений

Организация памяти и связывание имен

Способ организации памяти и связывания имен компилятором определяется в основном ответами на приведенный далее список вопросов.

1. Могут ли процедуры быть рекурсивными?
2. Что происходит с локальными переменными при возвращении управления из активации процедуры?
3. Может ли процедура обращаться к нелокальным именам?
4. Каким образом в вызываемую процедуру передаются параметры?
5. Может ли процедура быть передана в качестве параметра?
6. Может ли процедура быть возвращена в качестве результата?
7. Может ли память выделяться динамически, под управлением программы?
8. Должна ли память освобождаться явно?

Влияние этих вопросов (вернее, ответов на них) на поддержку времени исполнения для данного языка программирования рассматривается далее в этой главе.

7.2. Организация памяти

Организация памяти в процессе работы программы, описываемая в этом разделе, может использоваться в таких языках программирования, как Fortran, Pascal и С.

Классификация памяти времени выполнения

Предположим, что компилятор получил блок памяти от операционной системы для запуска скомпилированной программы. Из сказанного в предыдущем разделе следует, что эту память можно разделить для хранения:

1. сгенерированного целевого кода;
2. объектов данных;
3. части стека управления для отслеживания активаций процедур.

Размер сгенерированного целевого кода фиксируется во время компиляции, так что компилятор может разместить его в статически определенной области, возможно в нижних адресах памяти. Аналогично во время компиляции становится известным размер некоторых объектов данных, и они также могут быть размещены в статически определяемой области, как показано на рис. 7.7. Одна из причин статического выделения памяти для большего количества объектов данных заключается в том, что адреса этих

объектов могут быть внесены в целевой код при компиляции. В Fortran память для всех объектов данных может быть выделена статически.

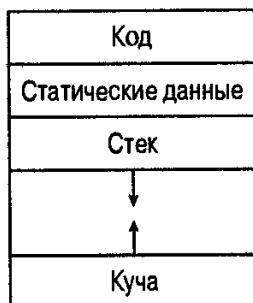


Рис. 7.7. Типичное разделение памяти времени исполнения на области кода и данных⁴

Реализации языков программирования типа Pascal и С используют расширения управляющего стека для работы с активациями процедур. При вызове выполнение активации прерывается, и информация о состоянии машины, такая как регистры компьютера и значение счетчика программы (указатель выполняющейся инструкции — *Прим. перев.*), сохраняется в стеке. При возврате управления из вызова активация продолжает работу после восстановления значений необходимых регистров и установки счетчика программы в точку, следующую непосредственно за вызовом. Объекты данных, время жизни которых содержится в данной активации, могут быть размещены в стеке вместе с другой информацией, связанной с активацией. Эта стратегия обсуждается в следующем разделе.

Отдельная область памяти времени исполнения, называемая *кучей* (heap), или свободной памятью, предназначена для хранения прочей информации. Pascal обеспечивает выделение памяти для данных под управлением программ, что будет обсуждаться в разделе 7.7; память для этих данных берется из кучи. Реализация языков, в которых времена жизни активаций не представимы деревом активаций, может использовать для хранения информации об активациях кучу. Управляемый способ выделения и освобождения памяти для данных в стеке делает размещение данных в стеке более выгодным, чем в куче.

Размеры стека и кучи могут изменяться в процессе работы программы, что показано на рис. 7.7, где стек и куча при увеличении размеров могут расти навстречу друг другу. И Pascal, и С требуют наличия как стека времени исполнения, так и кучи, но это требуется не во всех языках.

По соглашению стек растет вниз, т.е. “вершина” стека изображается внизу рисунка. Поскольку адреса памяти увеличиваются при передвижении к низу страницы, “возрастание вниз” означает переход к большим адресам. Если *top* указывает вершину стека, то смещение от вершины стека можно вычислить вычитанием адреса из *top*. На многих машинах при размещении значения *top* в регистре такое вычисление выполняется весьма эффективно. Адреса в стеке можно представить как смещение от *top*⁵.

⁴ Такое разделение типично для операционных систем, хотя и абстрагирует их реальную организацию. — *Прим. ред.*

⁵ Организация, представленная на рис. 7.7, предполагает, что память времени исполнения состоит из единого непрерывного блока, выделенного программе при запуске. Такое предположение налагает ограничение на совокупный размер стека и кучи. Если этот размер достаточно велик и редко превышается, то для большинства программ память расходуется понапрасну. Альтернатива, заключающаяся в связывании объектов в стеке и куче, может привести к высоким накладным расходам по отслеживанию вершины стека. Кроме того, целевые машины могут поддерживать иное размещение областей памяти — например, некоторые машины допускают только положительные смещения относительно адреса, содержащегося в регистре.

Записи активаций

Информация, необходимая для однократного выполнения процедуры, поддерживается с помощью непрерывного блока памяти, называемого *записью активации* (activation record), или *кадром* (frame). Этот блок состоит из набора полей, показанных на рис. 7.8. Не все языки и не все компиляторы используют каждое из этих полей; зачастую для хранения одного или нескольких из них используются регистры. В языках программирования Pascal и С обычно запись активации процедуры размещается в стеке времени исполнения при вызове процедуры и снимается со стека при передаче управления вызывающей программе или процедуре.

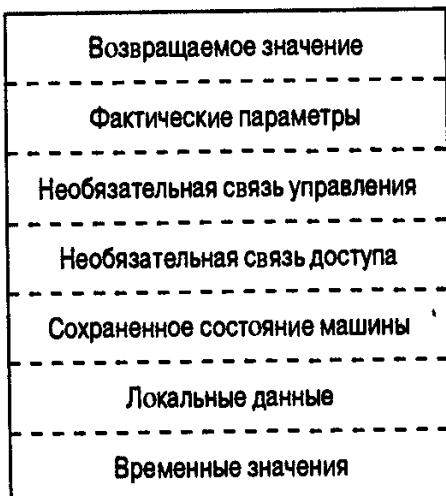


Рис. 7.8. Обобщенная запись активации

Назначение полей записи активации поясняется далее (начиная с поля временных значений).

1. Временные значения, появляющиеся, например, в процессе вычисления выражений, хранятся в поле временных значений.
2. Поле для локальных данных хранит данные, являющиеся локальными в процессе выполнения процедуры. Это поле будет дополнительно рассмотрено позже.
3. Поле сохраненного состояния машины хранит информацию о состоянии машины непосредственно перед вызовом процедуры. Эта информация включает значения счетчика программы и регистров машины, которые должны быть восстановлены при возврате управления из процедуры.
4. Необязательная связь доступа используется в разделе 7.4 для обращения к нелокальным данным, хранящимся в другой записи активации. Языки типа Fortran могут обойтись без этих связей, поскольку нелокальные данные хранятся в определенном месте. Связи доступа необходимы, например, в Pascal.
5. Необязательная связь управления указывает на запись активации вызывающей процедуры.
6. Поле для фактических параметров используется вызывающей процедурой для передачи параметров вызываемой процедуре. Мы показали место для параметров в записи активации, но на практике зачастую в целях повышения эффективности параметры передаются в регистрах.

7. Поле возвращаемого значения используется вызываемой процедурой для возврата значения вызывающей процедуре. Опять-таки, для повышения эффективности программы это значение часто возвращается в регистре.

Размер каждого из этих полей может быть определен в момент вызова процедуры. В действительности размеры почти всех полей можно определить во время компиляции. Исключение представляет только ситуация, когда у процедуры имеется локальный массив, размер которого определяется значением фактического параметра, доступного только при вызове процедуры во время работы программы. Подробнее о выделении памяти для данных переменной длины в записи активации вы узнаете из раздела 7.3.

Размещение локальных данных в процессе компиляции

Предположим, что во время работы программы память выделяется блоками последовательных байтов, где байт представляет минимальную единицу адресуемой памяти. Во многих машинах байт состоит из восьми бит, а некоторое количество байтов составляет машинное слово. Многобайтовые объекты хранятся в последовательных байтах, а их адрес определяется адресом первого байта.

Количество памяти, необходимой для некоторого имени, определяется его типом. Элементарные типы данных, такие как символ, целое или действительное число, обычно хранятся в небольшом числе байтов. В случае составных данных, таких как массивы или записи, требуемое количество памяти должно быть достаточно большим, чтобы хранить все их компоненты. Для простоты доступа к компонентам память для составных данных обычно выделяется одним непрерывным блоком байтов (подробнее об этом говорится в разделах 8.2 и 8.3).

Поле для локальных данных размечается по мере обработки объявлений в процессе компиляции. Данные переменной длины хранятся вне этого поля. Мы храним количество памяти, выделенной для предыдущих объявлений. По этому количеству мы определяем *относительный адрес* для локальной переменной относительно некоторой позиции, такой, например, как начало записи активации. Относительный адрес, или *смещение*, представляет собой разность между адресами упомянутой позиции и объекта данных.

На компоновку памяти для объектов данных сильное влияние оказывают ограничения адресации целевой машины. Например, инструкции сложения целых чисел могут требовать их *выравнивания* (align), т.е. размещения в определенных положениях в памяти — например, в адресах, кратных 4. В таком случае, хотя массиву из 10 символов достаточно для размещения в памяти 10 байт, компилятор может выделить ему 12 байт, оставляя 2 байта неиспользованными. Если же на первый план выходят вопросы экономии памяти, компилятор может *упаковать* данные, т.е. устраниТЬ неиспользуемую память; при этом в процессе работы программы могут понадобиться дополнительные действия по позиционированию упакованных данных, чтобы обеспечить возможность работы с ними.

Пример 7.3

На рис. 7.9 показана компоновка памяти, используемая компилятором С на двух машинах, условно названных нами “Машина 1” и “Машина 2”. С обеспечивает возможность работы с целыми числами трех размеров, объявленными с ключевыми словами `short`, `int` и `long`. Наборы инструкций наших машин таковы, что компилятор для Машины 1 выделяет 16, 32 и 32 бита для трех типов целых чисел, а компилятор для Машины 2 — соответственно 24, 48 и 64 бита. Для сравнения размеры на рис. 7.9 приведены в битах, хотя никакая машина не позволяет непосредственно адресовать биты.

Тип	РАЗМЕР		ВЫРАВНИВАНИЕ	
	Машина 1	Машина 2	Машина 1	Машина 2
char	8	8	8	64*
short	16	24	16	64
int	32	48	32	64
long	32	64	32	64
float	32	64	32	64
double	64	128	32	64
указатель на char	32	30	32	64
прочие указатели	32	24	32	64
структуры	≥8	≥64	32	64

* Символы в массивах выровнены по 8 бит

Рис. 7.9. Компоновка данных, используемая двумя компиляторами С

Память Машины 1 организована из байтов, состоящих из 8 бит каждый. Хотя каждый байт имеет свой адрес, набор инструкций эффективнее работает с короткими целыми числами (`short int`), расположенными в четных адресах, и прочими целыми числами в адресах, кратных 4. Компилятор размещает целые числа именно таким образом, и поэтому, например, для символа с последующим за ним коротким целым числом может быть выделена память размером в 4 байта (по 2 байта для символа и для короткого целого).

На Машине 2 каждое слово состоит из 64 бит, а для адресации слова требуется 24 бита. В пределах слова имеется 64 отдельных бита, так что для указания отдельного бита требуется 6 дополнительных бит. Конструкция машины такова, что указатель на символ в Машине 2 имеет размер 30 бит — 24 для указания слова и 6 для положения символа в слове.

Такая строгая ориентация на машинное слово приводит к тому, что на Машине 2 компилятор будет выделять память отдельными словами, даже если для представления всех возможных значений данного типа достаточно меньшего количества битов (например, для представления символа требуется 8 бит). Вот почему в таблице на рис. 7.9 показаны 64 бита выравнивания для всех типов у Машины 2. Здесь для символа с последующим за ним коротким целым будет выделена память размером 128 бит (два слова), из которых в первом слове будут использованы только 8 бит для символа, а во втором — 24 бита для короткого целого. □

7.3. Стратегии выделения памяти

В каждой из областей памяти, показанных на рис. 7.7, используется своя стратегия распределения памяти.

1. Для всех объектов данных в процессе компиляции используется *статическое распределение памяти*.
2. *Стековое распределение памяти* в процессе работы программы распределяет память в стеке.
3. *Распределение памяти в куче* выделяет и освобождает память, необходимую в процессе работы, из области данных, именуемой кучей.

Эти стратегии распределения будут применяться в данном разделе для записей активации. Мы также расскажем, каким образом целевой код процедуры обращается к памяти, связанной с локальным именем.

Статическое распределение памяти

При статическом распределении памяти имена связываются с памятью в процессе компиляции, так что пакет поддержки времени исполнения в этом случае не требуется. Поскольку связь имен с местоположениями в памяти в процессе работы не изменяется, всякий раз при активации процедур их имена связываются с одними и теми же адресами памяти. Это свойство позволяет значениям локальных имен *сохраняться* (retain) между активациями процедуры. Таким образом, когда управление передается процедуре, значения таких локальных переменных те же, что и в момент, когда управление последний раз покинуло данную процедуру.

Исходя из типа имени, компилятор вычисляет необходимое для него количество памяти (этот вопрос рассматривался в предыдущем разделе). Адрес этой памяти состоит из смещения от конца записи активации процедуры. В конечном счете компилятор должен принять решение о расположении записей активации по отношению к целевому коду и друг к другу. После принятия такого решения положение каждой записи активации (а следовательно, и памяти для каждого имени в записи активации) фиксируется. В процессе компиляции, таким образом, можно определить адреса, в которых находятся необходимые при работе данные, и внести их в целевой код. Аналогично в процессе компиляции известны адреса памяти, в которой сохраняется информация при вызове процедуры.

Однако использование только статического распределения памяти приводит к ряду ограничений.

1. Размеры объектов данных и ограничения на их размещение в памяти должны быть известны в процессе компиляции.
2. Рекурсивные процедуры существенно ограничены, поскольку все активации процедуры используют одно и то же связывание локальных имен.
3. Структуры данных не могут создаваться динамически, поскольку не имеется механизма выделения памяти в процессе работы программы.

Язык программирования Fortran разрешает статическое распределение памяти. Программа на языке Fortran состоит из основной программы, подпрограмм и функций (будем называть и те, и другие процедурами), как программа на языке Fortran 77, представленная на рис. 7.10. Расположение кода и записей активации этой программы показано на рис. 7.11. В записи активации для CNSUME размещаются локальные переменные BUF, NEXT и C. Память, выделенная для BUF, хранит строку из 50 символов. За ней следует пространство для хранения целого числа NEXT и символа C. То, что переменная NEXT объявлена и в процедуре PRDUCE, не вызывает никаких проблем, поскольку память для локальных переменных двух процедур выделяется в соответствующих записях активации.

```
(1)      PROGRAM CNSUME
(2)          CHARACTER * 50 BUF
(3)          INTEGER NEXT
(4)          CHARACTER C, PRDUCE
(5)          DATA NEXT /1/, BUF //' '/
```

```

(6) 6      C = PRDUCE()
(7)       BUF(NEXT:NEXT) = C
(8)       NEXT = NEXT + 1
(9)       IF ( C .NE. ' ' ) GOTO 6
(10)      WRITE (*, '(A)') BUF
(11)      END
(12)      CHARACTER FUNCTION PRDUCE()
(13)      CHARACTER * 80 BUFFER
(14)      INTEGER NEXT
(15)      SAVE BUFFER, NEXT
(16)      DATA NEXT /81/
(17)      IF ( NEXT .GT. 80 ) THEN
(18)          READ (*, '(A)') BUFFER
(19)          NEXT = 1
(20)      END IF
(21)      PRDUME = BUFFER(NEXT:NEXT)
(22)      NEXT = NEXT + 1
(23)      END

```

Рис. 7.10. Программа на языке программирования Fortran 77

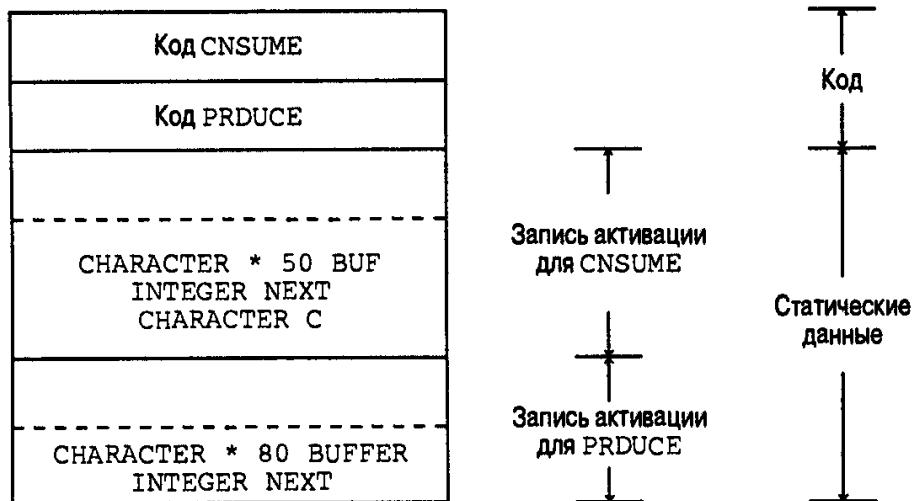


Рис. 7.11. Статическая память для локальных идентификаторов программы на языке Fortran 77

Поскольку размеры выполнимого кода и записей активации известны во время компиляции, возможна и другая организация памяти, отличная от приведенной на рис. 7.11. Компилятор Fortran может, например, разместить запись активации процедуры вместе с кодом этой процедуры. В некоторых компьютерных системах можно оставить относительное положение записей активации неопределенным и позволить редактору связей осуществить привязку записей активации к выполнимому коду.

Пример 7.4

Программа на рис. 7.10 полагается на то, что значения локальных переменных сохраняются между активациями процедуры. Инструкция SAVE в языке программирования Fortran 77 указывает, что значения локальных переменных в начале активации должны быть такими же, как и в конце предыдущей активации данной процедуры. Начальные значения этих локальных переменных могут быть указаны с помощью инструкции DATA.

Инструкция в строке 18 процедуры PRDUCE считывает строку текста в буфер, из которого затем последовательные вызовы процедуры возвращают по одному символу. Главная программа CNSUME также имеет буфер, в котором символы накапливаются до тех пор, пока не встретится символ пробела. При вводе

hello world

символы, возвращаемые последовательными активациями PRDUCE, показаны на рис. 7.12; выход программы при этом будет следующим:

hello

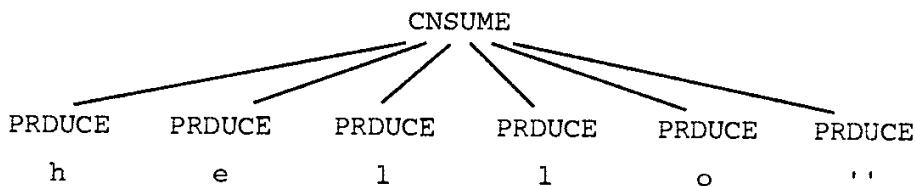


Рис. 7.12. Символы, возвращаемые активациями PRDUCE

Буфер, в который PRDUCE считывает строки, должен сохранять свое значение между активациями. Инструкция SAVE в строке 15 гарантирует, что при возвращении управления PRDUCE локальные переменные BUFFER и NEXT имеют те же значения, что и при предыдущем выходе управления из PRDUCE. При первой передаче управления PRDUCE значение локальной переменной NEXT получается из инструкции DATA в строке 16 (таким образом NEXT инициализируется значением 81). □

Стековое распределение памяти

Стековое распределение памяти основано на идее стека управления. Память организована в виде стека, и записи активации помещаются в стек и снимаются со стека в начале и конце активации соответственно. При каждом вызове процедуры память для локальных переменных содержится в записи активации для этого вызова. Таким образом, локальные переменные при каждом вызове связываются с новой областью памяти, поскольку при вызове в стек вносится новая запись активации. Более того, значения локальных переменных уничтожаются по окончании активации, т.е. значения локальных переменных становятся недоступны, так как выделенная им память освобождается при снятии записи активации со стека.

Сначала мы рассмотрим распределение стековой памяти в случае, когда размеры всех записей активации известны в процессе компиляции. Ситуации, когда во время компиляции полная информация о размерах недоступна, будут рассмотрены позже.

Предположим, что регистр *top* указывает вершину стека. В процессе работы запись активации может быть выделена и освобождена путем увеличения или уменьшения значения *top* на размер записи. Если процедура *q* имеет запись активации размера *a*, то *top* увеличивается на *a* непосредственно перед выполнением целевого кода *q*. Когда управление возвращается из *q*, *top* уменьшается на *a*.

Пример 7.5

На рис. 7.13 показаны записи активации, размещаемые в стеке времени исполнения при прохождении дерева активации, представленного на рис. 7.3. Пунктирные линии в дереве указывают уже завершенные активации. Выполнение начинается с активации

процедуры *s*. Когда управление достигает первого вызова в теле *s*, процедура *r* активируется и ее запись активации размещается в стеке. По возвращении управления из этой активации запись снимается со стека, оставляя там только одну запись — для *s*. Затем управление передается вызову *q* с фактическими параметрами 1 и 9, и на вершине стека для активации *q* выделяется запись активации. Пока управление находится в данной активации, ее запись находится на вершине стека.

Между двумя последними снимками состояний стека прошло несколько вызовов. В последнем снимке на рис. 7.13 за время жизни *q*(1, 3) начались и завершились активации *p*(1, 3) и *q*(1, 0); соответственно, их записи были размещены в стеке и сняты с него, а на вершине стека осталась запись для активации *q*(1, 3). □

В процедуре Pascal мы, как упоминалось в разделе 7.2, можем определить относительный адрес локальных данных в записи активации. Предположим, что в процессе работы *top* помечает положение конца записи. Адрес локального имени *x* в целевом коде процедуры может, таким образом, быть записан как *dx*(*top*); это указывает на то, что данные, связанные с *x*, могут быть найдены по адресу, полученному добавлением *dx* к значению регистра *top*. Заметим, что адрес может быть получен как смещение от значения любого другого регистра *r*, указывающего на фиксированную позицию в записи активации.

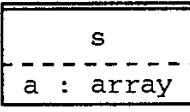
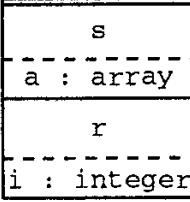
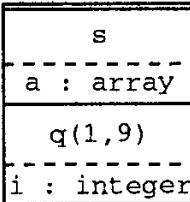
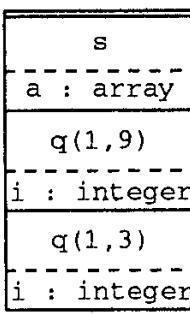
ПОЛОЖЕНИЕ В ДЕРЕВЕ АКТИВАЦИИ	ЗАПИСИ АКТИВАЦИИ В СТЕКЕ	ПРИМЕЧАНИЯ
<i>s</i>		Кадр для <i>s</i>
<i>r</i> —> <i>s</i>		Активируется <i>r</i>
<i>r</i> —> <i>s</i> <i>r</i> —> <i>q</i> (1, 9)		Кадр для <i>r</i> снимается со стека, и размещается кадр <i>q</i> (1, 9)
<i>r</i> —> <i>s</i> <i>r</i> —> <i>q</i> (1, 9) <i>p</i> (1, 9) —> <i>q</i> (1, 3) <i>p</i> (1, 3) —> <i>q</i> (1, 0)		Управление вернулось в <i>q</i> (1, 3)

Рис. 7.13. Размещение записей активации в растущем вниз стеке

Последовательности вызовов

Вызовы процедур реализуются путем генерации так называемых *последовательностей вызовов* (или *вызывающих последовательностей*) в целевом коде. Последовательность вызова выделяет запись активации и заполняет ее поля необходимой информацией. Последовательность возврата восстанавливает состояние машины, так что вызывающая процедура может продолжать работу.

Последовательности вызовов и записи активации различаются даже в разных реализациях одного и того же языка. Код в последовательности вызова зачастую разделяется между вызывающей и вызываемой процедурами. Не существует четкого разграничения задач времени исполнения между вызывающей и вызываемой процедурами — исходный язык, целевая машина и операционная система налагают свои требования, в силу которых может оказаться предпочтительным то или иное решение⁶.

Принцип, который помогает при разработке вызывающих последовательностей и записей активации, заключается в том, что поля, размеры которых определены заранее, размещаются в средине записи. В обобщенной записи активации, представленной на рис. 7.8, связь управления, связь доступа и поле состояния машины расположены в центре записи. Решение о том, использовать ли связи управления и доступа, является частью конструирования компилятора, так что эти поля фиксируются в процессе построения компилятора. Если при каждом вызове сохраняется одно и то же количество информации о состоянии машины, то сохранение и восстановление состояния для всех активаций выполняется посредством одного и того же кода. Более того, программы типа отладчиков могут легче декодировать содержимое стека при возникновении ошибки.

Несмотря на то что размер поля для временных значений в конечном счете фиксируется во время компиляции, он может быть неизвестен на начальной стадии компиляции. Аккуратная генерация кода или оптимизация могут уменьшить необходимое для работы процедуры количество временных переменных, так что на начальной стадии компиляции размер этого поля неизвестен. Поэтому в обобщенной записи активации это поле представлено после поля локальных данных, и изменения его размера не будут влиять на смещение объектов данных по отношению к полям в центре записи.

Поскольку каждый вызов имеет собственные фактические параметры, вызывающая процедура обычно вычисляет фактические параметры и передает их в запись активации вызываемой процедуры. Методы передачи параметров обсуждаются в разделе 7.5. В стеке времени исполнения запись активации вызывающей процедуры находится непосредственно под записью активации вызываемой процедуры, как показано на рис. 7.14. Имеется определенный выигрыш от размещения полей параметров и возможного возвращаемого значения вслед за записью активации вызывающей процедуры. Вызывающая процедура может в таком случае получить доступ к этим полям, используя смещения от конца собственной записи активации, без полной информации о компоновке записи активации вызываемой программы. В частности, вызывающая процедура ничего не обязана знать о локальных переменных вызываемой процедуры. Выигрыш от сокрытия этой информации заключается в рассматриваемой ниже возможности работы с процедурами с переменным числом аргументов (типа `printf` в языке программирования С).

⁶ Если процедура вызывается *n* раз, то часть последовательности вызова в вызывающих процедурах генерируется *n* раз, в то время как часть последовательности вызова в вызываемой процедуре генерируется лишь единожды. Следовательно, желательно разместить как можно большую часть последовательности вызова в коде вызываемой процедуры.

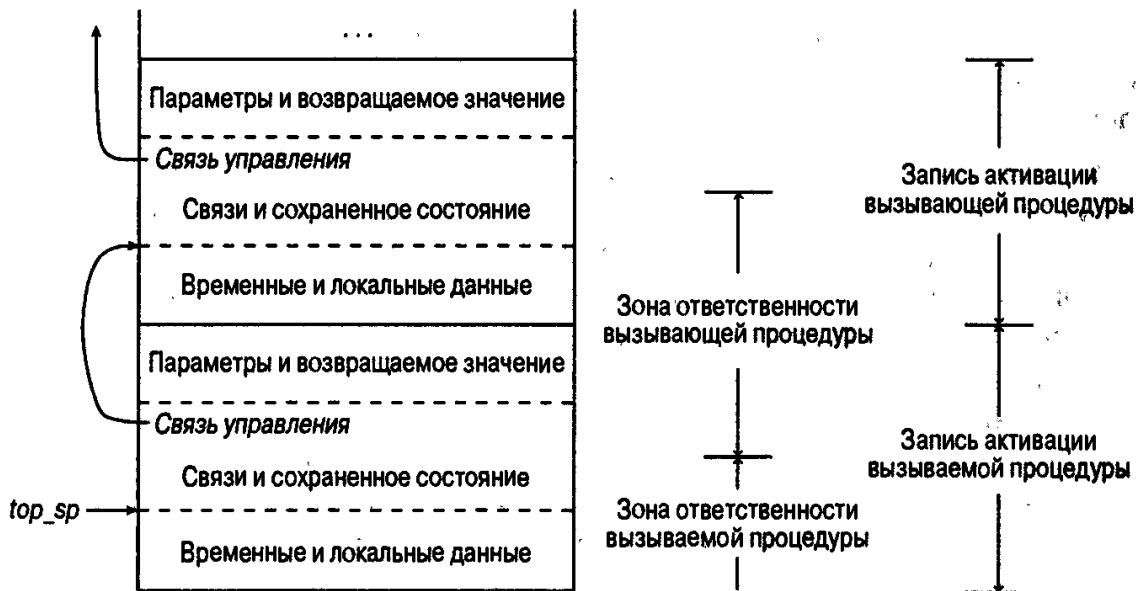


Рис. 7.14. Разделение задач между вызывающей и вызываемой процедурами

Языки типа Pascal требуют, чтобы локальные массивы процедуры имели размеры, известные во время компиляции. Однако зачастую размер локального массива может зависеть от значения параметров, переданных процедуре, и в этом случае размер всех локальных данных процедуры не может быть определен до вызова процедуры. Технологии работы с данными переменной длины обсуждаются далее в этом разделе.

Сказанное выше приводит к следующей последовательности вызова. Как и на рис. 7.14, регистр *top_sp* указывает на конец поля состояния машины в записи активации. Этот адрес известен вызывающей процедуре, поэтому она может отвечать за установку *top_sp* перед передачей управления вызываемой процедуре. Код вызываемой процедуры может обращаться к своим временным и локальным данным с использованием смещений от *top_sp*. Вызывающая последовательность выглядит следующим образом.

1. Вызывающая процедура вычисляет фактические параметры.
2. Вызывающая процедура сохраняет адрес возврата и старое значение *top_sp* в записи активации вызываемой процедуры, после чего увеличивает значение *top_sp* до положения, показанного на рис. 7.14. Таким образом, *top_sp* перемещается за локальные и временные данные вызывающей процедуры, параметры и поле состояния машины вызываемой процедуры.
3. Вызываемая процедура сохраняет значения регистров и другую информацию о состоянии машины.
4. Вызываемая процедура инициализирует локальные данные и начинает выполнение.

Возможная последовательность возврата выглядит следующим образом.

1. Вызываемая процедура размещает возвращаемое значение после записи активации вызывающей программы.
2. Используя информацию в поле состояния машины, вызываемая процедура восстанавливает *top_sp* и другие регистры и передает управление по адресу возврата в коде вызывающей процедуры.

3. Хотя значение `top_sp` было уменьшено, вызывающая процедура может скопировать возвращаемое значение в собственную запись активации и использовать его в вычислениях.

Приведенные выше последовательности вызова позволяют использовать процедуры с переменным числом аргументов. Заметим, что в момент компиляции известно количество параметров, передаваемых вызываемой функции. Следовательно, вызывающей процедуре известен размер поля параметров. Однако целевой код вызываемой процедуры должен быть готов обработать и другие вызовы, т.е. он ожидает вызов, а затем изучает поле параметров записи активации. При использовании организации данных, показанной на рис. 7.14, информация о параметрах размещается после поля состояния машины, так что вызываемая процедура в состоянии найти ее. Рассматривая в качестве примера функцию `printf` языка программирования C, мы видим, что первый аргумент функции определяет количество и вид остальных аргументов, так что как только функция `printf` находит первый аргумент, найти остальные для нее не представляет никакой сложности.

Данные переменной длины

Общая стратегия обработки данных переменной длины предложена на рис. 7.15, где процедура `r` имеет три локальных массива. Память для этих массивов не является частью записи активации `r`; в запись активации входят только указатели на начало каждого из массивов. Относительные адреса этих указателей известны во время компиляции, так что целевой код может обращаться к элементам массивов посредством этих указателей.

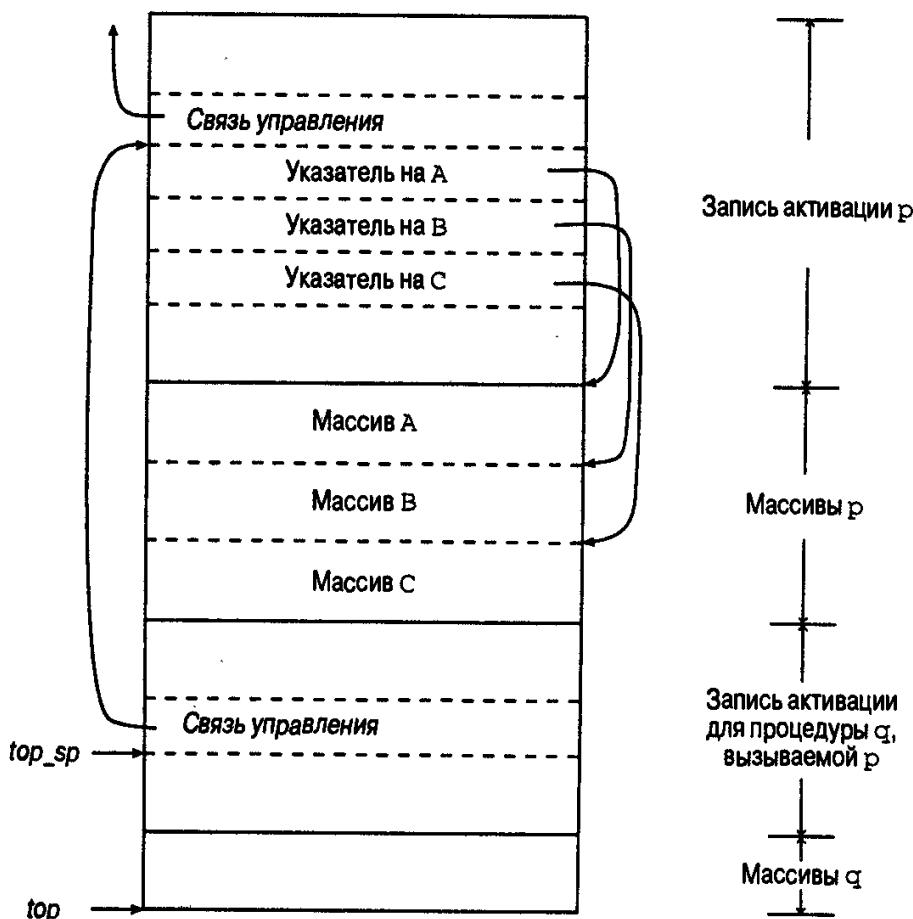


Рис. 7.15. Доступ к динамически размещаемым массивам

На рис. 7.15 показана также процедура *q*, вызываемая рассматриваемой процедурой *p*. Запись активации для *q* начинается после массивов *p*, а массивы переменной длины в *q* следуют за ней.

Обращение к данным в стеке происходит с помощью двух указателей — *top* и *top_sp*. Первый из них отмечает фактическую вершину стека и указывает, где начнется новая запись активации. Второй указатель используется для поиска локальных данных. Для согласованности с организацией данных, представленной на рис. 7.14, положим, что *top_sp* указывает на конец поля состояния машины. На рис. 7.15 *top_sp* указывает на конец этого поля в записи активации для *q*. В поле имеется управляющая связь с предыдущим значением *top_sp*, соответствующим активации вызывающей процедуры *p*.

Код для возврата значений *top* и *top_sp* может быть сгенерирован во время компиляции с использованием размеров полей в записях активации. При возврате из процедуры *q* новым значением *top* является *top_sp* минус длина полей состояния машины и параметров в записи активации *q*. Эта длина известна в процессе компиляции, как минимум, вызывающей процедуре. После установки *top* новое значение *top_sp* может быть скопировано из связи управления *q*.

Висячие ссылки

При освобождении памяти может возникнуть так называемая проблема *висячих ссылок*, т.е. ссылок на освобожденную память. Использование таких ссылок является логической ошибкой, поскольку значение освобожденной памяти неопределено в соответствии с семантикой большинства языков программирования. Хуже того, позже такая память может быть вновь выделена для других данных, что может привести к весьма странным ошибкам в работе программы.

Пример 7.6

Процедура *dangle* в С-программе, представленной на рис. 7.16, возвращает указатель на память, связанную с локальной переменной *i*. Этот указатель создается путем применения оператора & к переменной *i*. При возврате управления в функцию *main* из функции *dangle*, память для локальных переменных освобождается и может использоваться для других целей. Поскольку *p* в функции *main* ссылается на эту память, использование *p* порождает проблему висячей ссылки (одна из весьма частых ошибок начинающих программистов на С, в особенности при работе со строками. — Прим. ред.). □

```
main()
{
    int *p;
    p = dangle();
}
int *dangle()
{
    int i = 23;
    return &i;
}
```

Рис. 7.16. С-программа, в которой *p* указывает на освобожденную память

Распределение памяти в куче

Стратегия стекового распределения не может быть использована, если возможно одно из следующих условий.

1. Значения локальных переменных сохраняются по окончании активации.
2. Время жизни вызываемой процедуры превышает время жизни вызывающей. Такая ситуация не может возникнуть в языках программирования, в которых поток управления корректно изображается деревом активаций.

В каждом из этих случаев освобождение записей активации не обязано выполняться в соответствии со стековым правилом “последним вошел — первым вышел”, а значит, память не может быть организована в виде стека.

Распределение кучи разделяет непрерывную память на части, необходимые для записей активации или других объектов. Эти части могут освобождаться в любом порядке, так что через некоторое время куча состоит из чередующихся областей выделенной и свободной памяти.

Различия между выделением памяти в стеке и куче можно проследить на рис. 7.17 и 7.13. На рис. 7.17 запись активации процедуры r сохраняется по окончании активации. Таким образом, запись для новой активации $q(1, 9)$ не может следовать за записью активации s физически, как это было на рис. 7.13. Теперь, при освобождении записи активации для r , в куче появится свободное место между записями активации для s и $q(1, 9)$. Использование этого блока — задача диспетчера кучи.

ПОЛОЖЕНИЕ В ДЕРЕВЕ АКТИВАЦИИ	ЗАПИСИ АКТИВАЦИИ В КУЧЕ	ПРИМЕЧАНИЯ
r — s $q(1, 9)$		Сохраненная запись активации для r

Рис. 7.17. Записи активаций в куче не обязаны быть смежными

Эффективное управление кучей представляет собой специализированный вопрос теории структур данных; некоторые из применимых для этого технологий рассматриваются в разделе 7.8⁷. В целом, использование диспетчера кучи связано с некоторыми дополнительными расходами памяти и времени. С точки зрения эффективности может оказаться полезным следующий способ обработки малых записей активации или записей предсказуемого размера.

1. Для каждого интересующего нас размера создается связанный список свободных блоков этого размера.

⁷ Теоретическое рассмотрение вопросов динамического выделения памяти можно найти в книге Кнут Д. Э. Искусство программирования. Т. 1. Основные алгоритмы, 3-е изд. — М.: Издательский дом “Вильямс”, 2000 (раздел 2.5, “Динамическое выделение памяти”). — Прим. ред.

- При возможности запрос на память размером s удовлетворяется блоком размера s' , где s' — наименьший размер, больший или равный s среди имеющихся списков свободных блоков. Когда блок освобождается, он возвращается в список, из которого был получен.
- Для больших блоков памяти используется диспетчер кучи.

Этот подход ускоряет выделение и освобождение небольших участков памяти, так как получение блока из списка и возвращение его обратно — достаточно эффективные операции. Для больших участков памяти мы предполагаем, что их использование требует времени, по сравнению с которым время выделения памяти с помощью рассмотренной технологии оказывается пренебрежимо малым.

7.4. Доступ к нелокальным именам

Рассмотренные в предыдущем разделе стратегии выделения памяти адаптируются здесь для осуществления доступа к нелокальным именам. Хотя рассмотрение вопроса основано на стековом распределении записей активации, те же идеи применимы и к распределению памяти в куче.

Правила области видимости языка определяют способ работы со ссылками на нелокальные имена. Общее правило, именуемое *правилом лексической* (или *статической*) *области видимости*, определяет объявление, применяемое к имени, исключительно рассматрением текста программы. Pascal, C и Ada относятся к множеству тех языков программирования, которые используют лексическую область видимости с добавлением условия “ближайшее вложенное”, обсуждаемого ниже. Альтернативное *правило динамической области видимости* определяет применимое к имени объявление во время выполнения программы посредством рассмотрения текущих активаций. Динамическая область видимости используется в таких языках, как Lisp, APL и Snobol.

Мы начнем с рассмотрения блоков и правила “ближайшее вложенное”. Затем рассмотрим нелокальные имена в языках типа C, где используются лексические области видимости, где все нелокальные имена могут быть связаны со статически выделенной памятью и где запрещены вложенные объявления процедур.

В языках типа Pascal, которые используют вложенные объявления процедур и лексические области видимости, имена, принадлежащие различным процедурам, могут быть частью среды. Мы рассмотрим два пути поиска записей активации, содержащих память, связанную с нелокальными именами: связи доступа и дисплеи (displays).

Последний подраздел посвящен динамическим областям видимости.

Блоки

Блок представляет собой инструкцию, содержащую объявления собственных локальных данных. Концепция блока уходит корнями в Algol. В С синтаксис блока имеет следующий вид.

{ *Объявления Инструкции* }

Основная характеристика блоков заключается во вложенности их структуры. Начало и конец блока помечаются разделителями (в С для этой цели используются фигурные скобки { }; в Algol — ключевые слова begin и end). Разделители гарантируют, что блоки либо не зависят друг от друга, либо один блок является вложенным в другой.

Иными словами, невозможно такое перекрытие блоков B_1 и B_2 , что первым начинается блок B_1 , после чего блок B_2 , затем заканчивается блок B_1 , а затем — B_2 . Это свойство вложенности блоков иногда называют *блочной структурой*.

Область видимости в языке с блочной структурой определяется правилом ближайшего вложенного.

1. Область видимости объявления в блоке B включает блок B .
2. Если имя x не объявлено в блоке B , то все появления x в B находятся в области видимости объявления x во внешнем окружающем блоке B' , таком, что
 - i) B' содержит объявление x ;
 - ii) B' является ближайшим окружающим B блоком среди всех, содержащих объявление x .

```
main()
{
    int a = 0;
    int b = 0;
    {
        .   int b = 1;
        .
        .   int a = 2;
        .
        B2   .
        .   printf("%d %d\n", a, b);
B0  B1   }
        .
        .
        .   int b = 3;
        .
        B3   .
        .   printf("%d %d\n", a, b);
        .
        printf("%d %d\n", a, b);
    }
    printf("%d %d\n", a, b);
}
```

Объявление	Область видимости
int a = 0;	$B_0 - B_2$
int b = 0;	$B_0 - B_1$
int b = 1;	$B_1 - B_3$
int a = 2;	B_2
int b = 3;	B_3

Рис. 7.18. Блоки в C-программе

Каждое объявление на рис. 7.18 инициализирует объявляемое имя числом, указывающим номер блока, в котором оно появляется. Область видимости b в блоке B_0 не включает B_1 , поскольку в блоке B_1 переменная b объявлена заново (на рис. 7.18 это показано строкой B_0-B_1). Такой разрыв называется *дырой* (hole) в области видимости объявления.

Правило ближайшего вложенного отражено в выходе программы, представленной на рис. 7.18. Контроль передается блоку из точки непосредственно перед ним, а затем из блока в точку непосредственно за ним в исходном тексте. Инструкции печати, таким образом, выполняются в порядке B_2, B_3, B_1 и B_0 (в порядке, в котором управление покидает блоки). Значения a и b в этих блоках следующие.

```
2 1
0 3
0 1
0 0
```

Блочная структура может быть реализована с использованием стекового распределения памяти. Поскольку область видимости объявления не выходит за пределы блока, в котором оно записано, память для объявленного имени может быть выделена при входе в блок и освобождена при выходе из него. Это эквивалентно использованию блока как “процедуры без параметров”, вызываемой только из одной точки непосредственно перед блоком и возвращающей управление в точку непосредственно за блоком. Нелокальная среда для блока может поддерживаться с использованием технологий для процедур, которые будут рассмотрены позже в этой главе. Заметим, однако, что блоки проще процедур, поскольку им не передаются параметры, а поток управления строго следует статическому тексту программы⁸.

Другая реализация предусматривает однократное выделение памяти для всего тела процедуры. При наличии блоков в процедуре учитывается память, необходимая для объявлений внутри блоков. Для блока B_0 (рис. 7.18) можно выделить память так, как показано на рис. 7.19. Индексы у локальных переменных a и b на этом рисунке определяют блок, в котором расположено соответствующее объявление. Заметим, что для a_2 и b_3 может быть назначена одна и та же область памяти, поскольку блоки 2 и 3 не пересекаются.

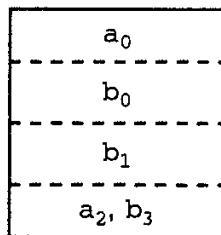


Рис. 7.19. Память для имен, объявленных на рис. 7.18

При отсутствии данных переменной длины максимальное количество памяти, необходимой для выполнения блока, может быть определено в процессе компиляции (с данными переменной длины можно работать с применением указателей, как описано в разделе 7.3). При определении этого размера мы для надежности полагаем, что в программе будут пройдены все возможные пути управления, т.е., например, что будут пройдены *и then*, *и else* части условной инструкции и что в цикле *while* будут выполняться все инструкции.

Лексическая область видимости без вложенных процедур

Правила лексической области видимости для С проще, чем для Pascal (они обсуждаются позже), поскольку в С запрещено вложенное определение процедур. Это означает, что определение процедуры не может находиться внутри другой процедуры. Программа С, как и на рис. 7.20, состоит из последовательности объявлений переменных и процедур (в С они называются функциями). Если в некоторой функции имеется нелокальная ссылка на имя a , то a должно быть объявлено вне любой функции. Область видимости объявления вне функции состоит из тел функций, следующих за объявлением (с дырами, ес-

⁸ Переход из блока в окружающий блок может быть реализован снятием со стека записей активации для окружающих блоков, границы которых при этом переходе пересекаются. В ряде языков программирования разрешен переход виутрь блока. Перед тем как управление будет передано таким образом внутрениему блоку, для внутренних блоков, границы которых пересекаются при таком переходе, должны быть созданы записи активации. Таким образом в них будут инициализированы локальные переменные — определяется семантикой языка.

ли это же имя объявлено в функции). На рис. 7.20 нелокальные появления *a* в *readarray*, *partition* и *main* ссылаются на массив, объявленный в строке 1.

```
(1) int a[11];
(2) readarray() { ... a ... }
(3) int partition(y,z) int y,z; { ... a ... }
(4) quicksort(m,n) int m, n; { ... }
(5) main() { ... a ... }
```

Рис. 7.20. С-программа с нелокальным *a*

При отсутствии вложенных процедур для языка с лексическими областями видимости типа С можно использовать стратегию стекового распределения памяти для локальных имен из раздела 7.3. Память для всех имен, объявленных вне процедур, может быть выделена статически. Положение этой памяти известно в процессе компиляции, так что если некоторое имя нелокально в теле процедуры, мы можем просто воспользоваться статически определенным адресом. Любое другое имя должно быть локальным по отношению к активации на вершине стека и доступным посредством указателя *top*. Вложенные процедуры нарушают работоспособность этой схемы, поскольку нелокальные данные в этом случае могут попросту оказаться глубже в стеке (являясь локальными данными внешней процедуры).

Важное преимущество статического выделения памяти для нелокальных имен заключается в том, что объявленные процедуры могут свободно передаваться в качестве параметра другим процедурам и возвращаться как результат вызова (в С функция передается как параметр путем передачи указателя на нее). При использовании лексической области видимости и отсутствии вложенных процедур любое нелокальное для одной процедуры имя является нелокальным для всех остальных процедур, и его статический адрес может использоваться всеми процедурами независимо от того, каким образом они активируются. Аналогично, если процедура возвращается как результат, нелокальные имена в возвращаемой процедуре ссылаются на статически выделенную для них память.

```
(1) program pass(input, output);
(2)     var @m: integer;
(3)     function f(n: integer) : integer;
(4)         begin f := @m + n end { f }
(5)     function g(n: integer) : integer;
(6)         begin g := @m * n end { f }
(7)     procedure b(function h (n: integer) : integer);
(8)         begin write(h(2)) end { b }
(9)     begin
(10)        @m := 0;
(11)        b(f); b(g); writeln
(12)    end.
```

Рис. 7.21. Pascal-программа с нелокальным *m*

Рассмотрим, например, Pascal-программу, приведенную на рис. 7.21. Все вхождения имени *m*, отмеченные на рис. 7.21 кружком, находятся в области видимости объявления в строке 2. Поскольку *m* нелокально для всех процедур программы, память для нее может быть выделена статически. При выполнении процедуры *f* и *g* могут использовать для

доступа к *m* статический адрес. То, что *f* и *g* передаются в качестве параметров, влияет только на их вызов, но не на их доступ к значению *m*.

Более подробно, вызов *b(f)* в строке 11 связывает функцию *f* с формальным параметром *h* процедуры *b*. Так, когда в строке 8 вызывается формальный параметр *h* (*write(h(2))*), активируется функция *f*. Активация *f* возвращает 2, поскольку *m* имеет значение 0, а формальный параметр *n* — значение 2. Затем вызов *b(g)* связывает *g* с *h*; в этом случае вызов *h* активирует *g*. Вывод программы — “2 0”.

Лексическая область видимости при наличии вложенных процедур

Нелокальное имя *a* в процедуре Pascal находится в области видимости ближайшего из вложенных объявлений *a* в статическом тексте программы.

```
(1)  program sort(input, output);
(2)    var a : array [0..10] of integer;
(3)      x : integer;
(4)    procedure readarray;
(5)      var i : integer;
(6)      begin ... a ... end { readarray };
(7)    procedure exchange( i, j : integer);
(8)      begin
(9)        x := a[i]; a[i] := a[j]; a[j] := x
(10)       end { exchange };
(11)    procedure quicksort( m, n : integer);
(12)      var k, v : integer;
(13)      function partition( y, z : integer): integer;
(14)        var i, j: integer;
(15)        begin ... a ...
(16)          ... v ...
(17)          ... exchange(i,j); ...
(18)        end { partition };
(19)      begin ... end { quicksort };
(20)    begin ... end { sort }.
```

Рис. 7.22. Pascal-программа с вложенными процедурами

Вложенность определений процедур в Pascal-программе на рис. 7.22 можно указать отступами следующим образом.

```
sort
  readarray
  exchange
  quicksort
    partition
```

Появление *a* в строке 15 (рис. 7.22) находится в функции *partition*, вложенной в процедуру *quicksort*. Ближайшее вложенное объявление *a* находится в строке 2 в процедуре, состоящей из всей программы. Правило ближайшего вложенного применимо и к именам процедур. Так, процедура *exchange*, вызываемая процедурой *partition* в строке 17, является нелокальной по отношению к *partition*. Применяя правило ближайшего вложенного, мы вначале проверяем, не определена ли процедура *exchange* в *quicksort*; поскольку это не так, мы ищем ее объявление в программе *sort*.

Глубина вложенности

Понятие *глубина вложенности* (nesting depth) процедуры используется ниже при реализации лексической области видимости. Пусть имя программы представляет уровень вложенности 1; затем мы добавляем по 1 к глубине вложенности при переходе от внешней процедуры к вложенной. На рис. 7.22 процедура `quicksort` в строке 11 имеет глубину вложенности 2, а `partition` в строке 13 — глубину вложенности 3. С каждым именем в программе мы связываем глубину вложенности процедуры, в которой оно объявлено. Таким образом, имена `a`, `v` и `i`, встречающиеся в строках 15–17 в `partition`, имеют глубину вложенности 1, 2 и 3 соответственно.

Связи доступа

Непосредственная реализация лексической области видимости для вложенных процедур может быть получена путем добавления указателя, называемого *связью доступа* (access link) к каждой записи активации. Если процедура `r` в исходном тексте вложена в процедуру `q` непосредственно, то связь доступа в записи активации `r` указывает на связь доступа в записи последней активации `q`.

Снимки стека времени исполнения в процессе выполнения программы на рис. 7.22 приведены на рис. 7.23 (для краткости на рисунке приведены только первые буквы имен процедур, а связи доступа указаны аббревиатурой *acl*). Связь доступа у активации `sort` пуста, поскольку у нее нет окружающей ее процедуры. Связь доступа в каждой активации `quicksort` указывает на запись для `sort`. Обратите внимание, что на рис. 7.23 в связь доступа в записи активации `partition(1, 3)` указывает на связь доступа в записи последней активации `quicksort`, а именно `quicksort(1, 3)`.

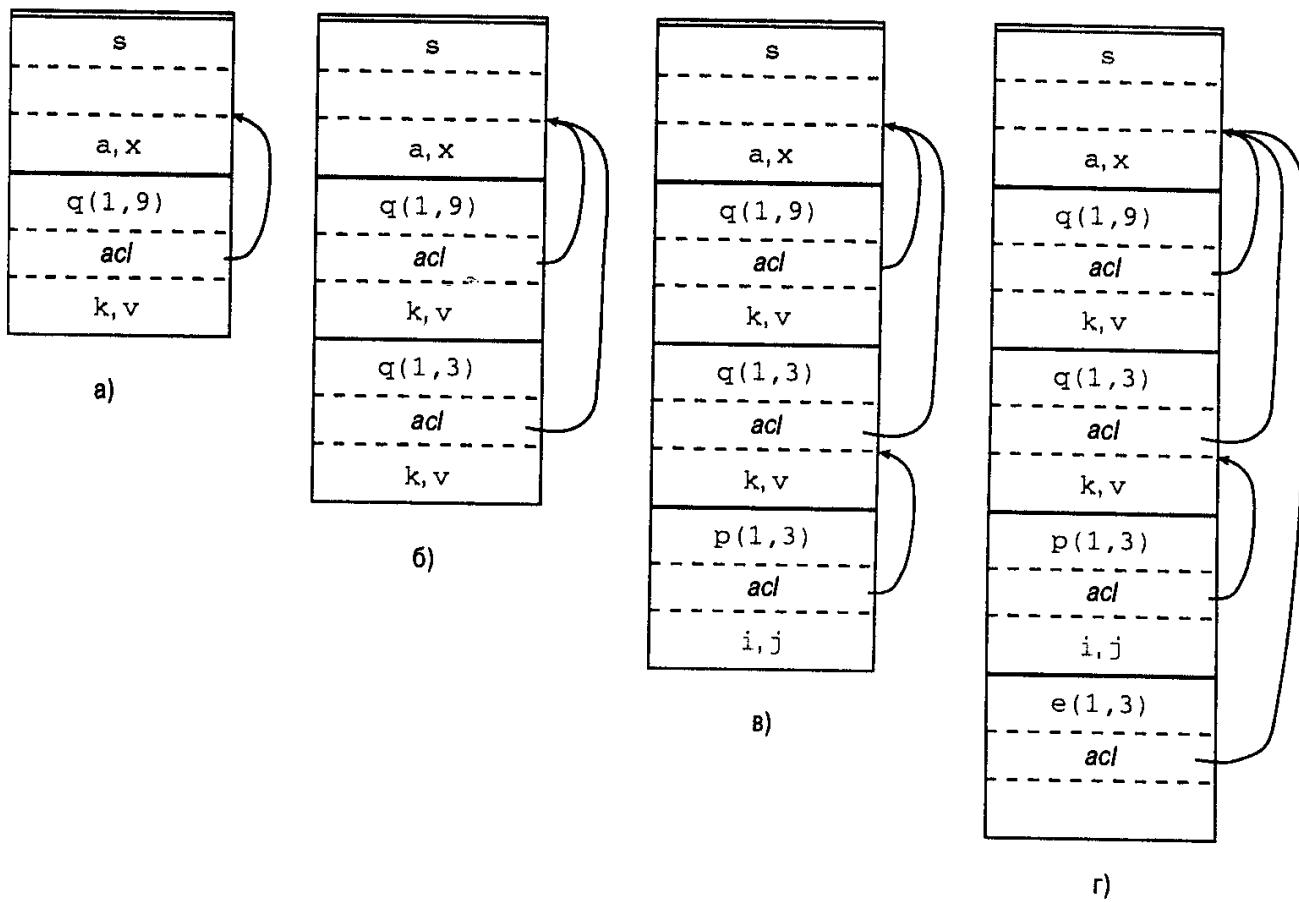


Рис. 7.23. Связи доступа для поиска памяти для нелокальных имен

Предположим, что процедура p с глубиной вложенности n_p обращается к нелокальному имени a с глубиной вложенности $n_a \leq n_p$. Область памяти, выделенная a , может быть найдена следующим образом.

1. Когда управление находится в p , запись активации для p располагается на вершине стека. Проследуем по $n_p - n_a$ связям доступа от записи, находящейся на вершине стека (значение $n_p - n_a$ может быть вычислено в процессе компиляции).
2. После прохода по $n_p - n_a$ связям мы достигаем записи активации для процедуры, по отношению к которой a является локальным именем. Как обсуждалось в предыдущем разделе, выделенная ему память имеет фиксированное положение в записи, в частности определенное смещение относительно связи доступа.

Следовательно, адрес нелокального имени a в процедуре p задается следующей, вычисляемой в процессе компиляции и хранящейся в таблице символов, парой значений ($n_p - n_a$, смещение в содержащей a записи активации). Первый компонент указывает, сколько связей доступа должно быть пройдено.

Например, в строках 15–16 (рис. 7.22) процедура `partition` с глубиной вложенности 3 обращается к нелокальным переменным a и v с глубиной вложенности 1 и 2, соответственно. Записи активации, в которых хранятся эти переменные, находятся прохождением соответственно $3-1=2$ и $3-2=1$ связей доступа от записи активации `partition`.

Код настройки связей доступа является частью последовательности вызова. Предположим, что процедура p глубины вложенности n_p вызывает процедуру x глубины вложенности n_x . Код настройки связи доступа вызываемой процедуры зависит от того, является ли вызываемая процедура вложенной в вызывающую. Возможны 2 случая.

1. $n_p < n_x$. Поскольку вызываемая процедура x вложена глубже p , она должна быть объявлена в p (иначе из p она будет недоступна). Так происходит, например, при вызове `quicksort` из `sort` на рис. 7.23 α и при вызове `partition` из `quicksort` на рис. 7.23 β . В этом случае связь доступа в вызываемой процедуре должна указывать на связь доступа в записи активации вызывающей процедуры, находящейся непосредственно за текущей ниже по стеку.
2. $n_p \geq n_x$. Согласно правилам области видимости, процедуры с глубинами вложенности 1, 2, ..., n_x-1 , окружающие вызываемую и вызывающую процедуры, должны быть одними и теми же, как при вызове `quicksort` самой себя на рис. 7.23 β и при вызове функцией `partition` процедуры `exchange` на рис. 7.23 γ . Следуя $n_p - n_x + 1$ связи доступа от вызывающей процедуры, мы достигаем записи последней активации процедуры, которая статически наиболее тесно охватывает и вызывающую, и вызываемую процедуры. Достигнутая при этом связь доступа является той связью, на которую должна указывать связь вызываемой процедуры. Значение $n_p - n_x + 1$ может быть вычислено в процессе компиляции.

Параметры-процедуры

Правила лексической области видимости применимы и в том случае, когда вложенная процедура передается в качестве параметра. В Pascal-программе (рис. 7.24) функция f (строки 6–7) использует нелокальную переменную m (все появления m в листинге отмечены кружками). В строке 8 процедура c присваивает переменной m нулевое значение, а затем передает процедуре b в качестве параметра процедуру f . Заметим, что область видимости объявления m в строке 5 не включает тело процедуры b в строках 2–3.

```

(1) program param(input, output);
(2)     procedure b(function h(n: integer): integer);
(3)         begin writeln(h(2)) end { b };
(4)     procedure c;
(5)         var m: integer;
(6)         function f(n: integer): integer;
(7)             begin f := m+n end { f };
(8)             begin m := 0; b(f) end { c };
(9)         begin
(10)             c
(11)         end.

```

Рис. 7.24. Связь доступа должна быть передана вместе с фактическим параметром f

В теле b инструкция writeln(h(2)) активирует f, поскольку формальный параметр h ссылается на эту функцию. Таким образом, writeln выводит результат вызова f(2).

Каким образом должна быть настроена связь доступа для активации f? Ответ состоит в том, что вложенная процедура, передаваемая в качестве параметра, должна передаваться вместе со своей связью доступа, как показано на рис. 7.25. Когда процедура c передает f, она определяет связь доступа для f, как если бы подготавливался вызов f. Эта связь передается вместе с f процедуре b. Затем, при активации f из b, переданная связь используется для настройки связи доступа в записи активации f.

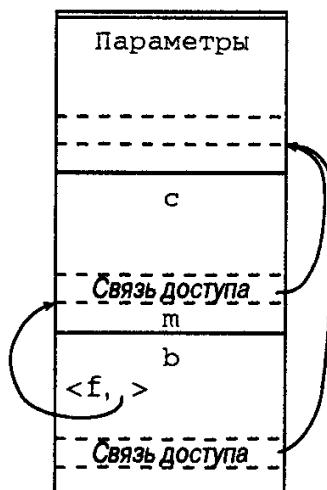


Рис. 7.25. Фактический параметр f передается вместе со связью доступа

Дисплеи

Более быстрый, по сравнению с методом использования связей доступа, способ обращения к нелокальным именам можно получить с помощью массива *d* указателей на записи активации, который называется *дисплеем*⁹ (display). Дисплей поддерживается таким образом, чтобы память для нелокального а с глубиной вложенности *i* находилась в записи активации, на которую указывает элемент дисплея *d[i]*.

⁹ В русскоязычной литературе по компиляции этот термин появился в 60-е годы в виде кальки английского *display* — выставка, показ. Ввиду локальности его использования мы оставляем его таким же. По сути же, он означает стек ссылок на записи активации. — Прим. ред.

Предположим, что управление находится в активации процедуры p с глубиной вложенности j . Тогда первые $j-1$ элементов дисплея указывают на предыдущие активации процедур, лексически охватывающих процедуру p , и $d[j]$ указывает на активацию p . Использование дисплея в целом быстрее, чем следование по цепочке связей доступа, поскольку запись активации, хранящая локальное имя, обнаруживается обращением к элементу d и переходом всего лишь по одному указателю.

Простая схема поддержки дисплея использует связи доступа в дополнение к нему. Частью вызывающей и возвращающей последовательностей является обновление дисплея путем следования по цепочке связей доступа. При следовании связи к записи активации с глубиной вложенности n , элемент дисплея $d[n]$ устанавливается на эту запись активации. По сути, дисплей дублирует информацию цепочки связей доступа.

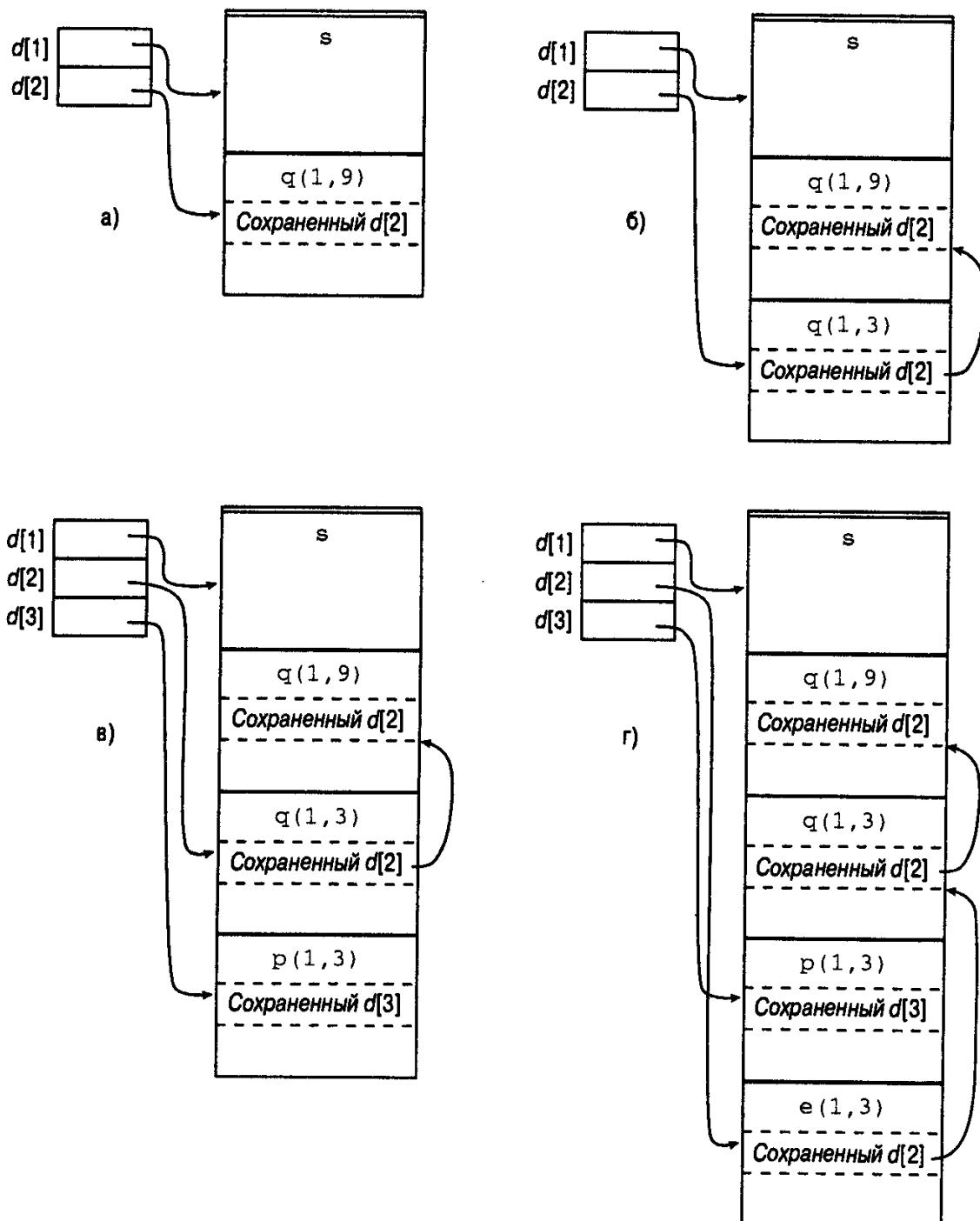


Рис. 7.26. Поддержка дисплея при обычном вызове процедур

Эту простую схему можно усовершенствовать. Метод, иллюстрируемый на рис. 7.26, требует меньшей работы при входе в процедуру и выходе из нее при обычном вызове, когда процедура не передается в качестве параметра. На рис. 7.26 дисплей состоит из отдельного глобального массива. Снимки на рисунке соответствуют выполнению исходного текста, приведенного на рис. 7.22 (здесь мы вновь используем только первые символы имен процедур).

На рис. 7.26 a показана ситуация непосредственно перед началом активации $q(1, 3)$. Поскольку `quicksort` имеет глубину вложенности 2, новая активация $q(1, 3)$ воздействует на элемент дисплея $d[2]$. Это действие показано на рис. 7.26 b , где $d[2]$ теперь указывает на новую запись активации; старое значение сохранено в новой записи активации¹⁰. Сохраненное значение потребуется позже, для восстановления дисплея в состояние, показанное на рис. 7.26 a , когда управление вернется в активацию $q(1, 9)$.

Дисплей изменяется, когда происходит новая активация, и должен быть восстановлен при возврате управления из нее. Правила области видимости Pascal и других языков с лексической областью видимости позволяют поддерживать дисплей с помощью следующих шагов. Мы рассмотрим только простейший случай, когда процедура не передается в качестве параметра (см. упр. 7.8). При настройке записи активации для процедуры глубины вложенности j мы выполняем следующее.

1. Сохраняем значение $d[i]$ в новой записи активации.
2. Устанавливаем $d[i]$ на новую запись активации.

Непосредственно перед завершением активации $d[i]$ принимает ранее сохраненное значение.

Эти шаги обосновываются следующим образом. Предположим, что процедура глубины вложенности j вызывает процедуру глубины i . Возможны два случая, зависящих от того, вложена ли вызываемая процедура в исходном тексте в вызывающую или нет (что уже обсуждалось при рассмотрении связей доступа).

1. $j < i$. Тогда $i=j+1$, и вызываемая процедура вложена в вызывающую. Первые j элементов дисплея, таким образом, не требуют изменений, и мы устанавливаем $d[i]$ на новую запись активации. Этот случай иллюстрируется на рис. 7.26 a , где `sort` вызывает `quicksort`, и на рис. 7.26 b , где `quicksort` вызывает `partition`.
2. $j \geq i$. В этом случае, как выяснялось ранее при рассмотрении связей доступа, охватывающие процедуры глубин вложенности $1, 2, \dots, i-1$ для вызывающей и вызываемой процедур должны быть одними и теми же. В этом случае мы сохраняем старое значение $d[i]$ в новой записи активации и устанавливаем $d[i]$ на новую запись активации. Дисплей корректно работает, поскольку первые $i-1$ элементов остаются неизменными.

Пример второго случая с $i=j=2$ приведен на рис. 7.26 b , где рекурсивно вызывается `quicksort`. Более интересный пример — когда активация $p(1, 3)$ глубины вложенности 3 вызывает $e(1, 3)$ глубины вложенности 2; охватывающая их процедура `s` имеет глубину вложенности 1 (см. рис. 7.26 c ; текст программы приведен на рис. 7.22). Обратите внимание на то, что при вызове $e(1, 3)$ значение $d[3]$, принадлежащее $p(1, 3)$, ос-

¹⁰ Заметим, что $q(1, 9)$ также сохраняет $d[2]$, хотя может случиться так, что второй элемент дисплея никогда не использовался до этого и его не обязательно восстанавливать. Проще сохранять $d[2]$ для всех вызовов q , чем в процессе выполнения программы принимать решение о необходимости такого сохранения.

тается в дисплее, хотя и не может быть доступно, пока управление находится в е. Если е вызовет другую процедуру глубины вложенности 3, она сохранит значение $d[3]$ и восстановит его при возврате управления е. Таким образом, мы можем показать, что каждая процедура видит корректный дисплей для всех глубин вложенности вплоть до собственной глубины.

Имеется ряд мест, где может содержаться дисплей. Если у нас достаточно регистров, то дисплей, рассматриваемый как массив, может быть набором регистров. Заметим, что компилятор в состоянии определить максимальную длину этого массива — она равна максимальной глубине вложенности процедур в программе. Если регистров недостаточно, дисплей может размещаться в статически выделенной памяти и все обращения к записям активации начинаются с использования косвенной адресации посредством соответствующего указателя дисплея. Этот подход вполне обоснован на машине с косвенной адресацией, хотя каждое косвенное обращение и стоит цикла обращения к памяти. Еще одна возможность — хранить дисплей в стеке времени исполнения и создавать новую копию при каждом входе в процедуру.

Динамическая область видимости

При использовании динамической области видимости новая активация наследует существующую привязку нелокальных имен к памяти. Нелокальное имя а в вызываемой активации ссылается на ту же память, что и в вызывающей активации. Новые связи устанавливаются только для локальных имен вызываемой процедуры; эти имена связываются с памятью в новой записи активации.

```
(1)  program dynamic(input, output);
(2)    var r: real;
(3)    procedure show;
(4)      begin write(r:5:3) end;
(5)    procedure small;
(6)      var r: real;
(7)      begin r := 0.125; show end;
(8)    begin
(9)      r := 0.25;
(10)     show; small; writeln;
(11)     show; small; writeln
(12)   end.
```

Рис. 7.27. Выход этой программы зависит от того, используется ли лексическая или динамическая область видимости

Программа, показанная на рис. 7.27, иллюстрирует динамическую область видимости. Процедура show в строках 3, 4 выводит значение нелокальной переменной r. При использовании лексической области видимости языка Pascal нелокальная переменная r располагается в области видимости объявления в строке 2. В результате выход программы будет следующим.

```
0.250 0.250
0.250 0.250
```

Однако при применении динамической области видимости программа выведет следующее.

```
0.250 0.125
0.250 0.125
```

Когда процедура `show` вызывается в строках 10, 11 главной программы, выводится 0.250, поскольку используется переменная `r`, локальная по отношению к главной программе. Однако при вызове `show` в строке 7 из `small` выводится значение 0.125, поскольку используется переменная `r`, локальная по отношению к `small`.

Следующие два подхода к реализации динамической области видимости имеют некоторое сходство с использованием связей доступа и дисплеев в реализации лексической области видимости.

1. *Глубокий доступ* (deep access). Концептуально динамическая область памяти получается, если связи доступа указывают на ту же запись активации, что и связи управления. Простейшая реализация состоит в том, чтобы обойтись без связей доступа и использовать связи управления для поиска в стеке первой записи активации, содержащей память для нелокального имени. Термин *глубокий доступ* происходит от того, что поиск может “глубоко” погрузиться в стек. Глубина поиска зависит от входных параметров программы и не может быть определена в процессе компиляции.
2. *Мелкий доступ* (shallow access). В этом случае идея заключается в хранении текущего значения каждого имени в статически выделяемой памяти. При новой активации процедуры `r` локальное имя `n` в `r` получает статически выделенную для `n` память. Предыдущее значение `n` может храниться в записи активации для `r` и должно быть восстановлено по окончании активации `r`.

Компромисс между двумя подходами основывается на том, что при глубоком доступе обращение к нелокальным переменным осуществляется дольше, но при этом нет никаких накладных расходов, связанных с началом и окончанием активации. Мелкий же доступ обеспечивает непосредственное обращение к нелокальным переменным, но при этом затрачивает время на поддержку их значений в начале и при окончании активации. При передаче функций как параметров и возвращении как результатов, более простая реализация получается при использовании глубокого доступа.

7.5. Передача параметров

Когда одна процедура вызывает другую, обычный метод сообщения между ними состоит в использовании нелокальных имен и параметров вызываемой процедуры. В процедуре, приведенной на рис. 7.28 и предназначеннной для обмена значениями `a[i]` и `a[j]`, используются как нелокальные имена, так и параметры (нелокальным именем по отношению к процедуре является массив `a`, а параметрами — `i` и `j`).

```
(1) procedure exchange(i, j: integer);
(2)     var x: integer;
(3)     begin
(4)         x := a[i]; a[i] := a[j]; a[j] := x;
(5)     end
```

Рис. 7.28. Pascal-процедура обмена значениями с нелокальными именами и параметрами

В этом разделе обсуждается ряд методов сопоставления фактических и формальных параметров. Это — передача по значению, передача по ссылке, копирование—вос-

становление, передача по имени и макрорасширения. Знать метод передачи параметров, используемый в языке или компиляторе, крайне важно, так как от этого метода может зависеть результат работы программы.

Почему существует так много методов? Различные методы вытекают из различных интерпретаций смысла выражений. В выражении типа $a[i]:=a[j]$ выражение $a[j]$ представляет значение, в то время как $a[i]$ — местоположение в памяти, куда помещается значение $a[j]$. Решение о том, каким образом трактовать выражение — как значение или адрес памяти — определяется тем, где по отношению к оператору присвоения располагается выражение — слева или справа. Как и в главе 2, “Простой однопроходный компилятор”, термины *l*-значение и *r*-значение представляют собой соответственно адрес памяти и значение выражения. Префиксы *l*- и *r*- происходят от положения выражения слева или справа от оператора присвоения.

Различия между способами передачи параметров определяются в первую очередь тем, представляет ли фактический параметр *l*- или *r*-значение или попросту является текстом, представляющим самого себя.

Передача по значению

Этот способ, по существу, представляет собой простейший метод передачи параметров. При этом происходит вычисление фактических параметров, и полученные *r*-значения передаются вызываемой процедуре. Таким образом происходит передача параметров в C; в Pascal параметры также обычно передаются по значению. Метод передачи по значению реализуется следующим образом.

1. Формальный параметр рассматривается как локальное имя, так что память для него выделяется в записи активации вызываемой процедуры.
2. Вызывающая процедура вычисляет фактические параметры и помещает их *r*-значения в память, выделенную для формальных параметров.

Отличительная особенность этого метода заключается в том, что операции над формальными параметрами не влияют на значения в записи активации вызывающей программы. Если удалить ключевое слово *var* в строке 3 на рис. 7.29, Pascal передаст *x* и *y* в процедуру *swap* по значению. Вызов *swap(a, b)* в строке 12 оставит значения *a* и *b* нетронутыми. При передаче по значению вызов *swap(a, b)* эквивалентен следующей последовательности.

```
x := a
y := b
temp := x
x := y
y := temp
```

Здесь *x*, *y* и *temp* локальны по отношению к *swap*. Хотя приведенные присвоения и изменяют значения локальных переменных, эти изменения оказываются потерянными при возврате управления из вызова и уничтожении записи активации для процедуры *swap*. Таким образом, вызов не влияет на запись активации вызывающей программы.

```
(1) program reference(input, output);
(2) var a, b: integer;
(3) procedure swap(var x, y: integer);
(4)     var temp: integer;
(5) begin
```

```

(6)      temp := x;
(7)      x := y;
(8)      y := temp;
(9)  end;
(10) begin
(11)    a := 1; b := 2;
(12)    swap(a, b);
(13)    writeln('a =', a); writeln('b =', b)
(14) end.

```

Рис. 7.29. Pascal-программа с процедурой swap

Процедура с передачей параметров по значению может влиять на вызывающую процедуру посредством нелокальных имен (см., например, процедуру *exchange* на рис. 7.28) или указателей путем их явной передачи по значению. В С-программе (рис. 7.30) *x* и *y* объявлены в строке 2 как указатели на целые числа; оператор & в вызове *swap (&a, &b)* в строке 8 приводит к тому, что в *swap* передаются указатели на *a* и *b*. Выход этой программы — “*a is now 2, b is now 1*”.

```

(1) swap(x,y)
(2) int *x, *y;
(3) {   int temp
(4)     temp = *x; *x = *y; *y = temp;
(5) }
(6) main()
(7) {   int a = 1, b = 2;
(8)     swap(&a, &b);
(9)     printf("a is now %d, b is now %d\n",a,b);
(10) }

```

Рис. 7.30. С-программа, использующая передачу указателей по значению

Использование указателей в этом примере, по сути, показывает, каким образом применение передачи параметров по ссылке допускает обмен значениями.

Передача по ссылке

При передаче параметров *по ссылке* (известной также, как *передача по адресу*) вызывающая процедура передает вызываемой указатель на адрес памяти каждого фактического параметра.

- Если фактический параметр представляет собой имя или выражение, имеющее *l*-значение, то вызываемой процедуре передается *l*-значение.
- Если фактический параметр представляет собой выражение, например *a+b* или 2, не имеющее *l*-значения, то результат вычисления выражения размещается в новом месте в памяти, адрес которого и передается процедуре¹¹.

Обращение к формальному параметру в вызываемой процедуре превращается в целевом коде в косвенное обращение через переданный указатель.

¹¹ Здесь, в зависимости от используемого языка программирования, могут действовать дополнительные условия. Так, например, C++ запрещает инициализацию неконстантной ссылки временным объектом. — Прим. ред.

Пример 7.7

Рассмотрим процедуру swap на рис. 7.29. Вызов swap с фактическими параметрами i и $a[i]$ приведет к тому же результату, что и следующая последовательность шагов.

1. Скопировать адреса (l -значения) i и $a[i]$ в запись активации вызываемой процедуры, скажем в ячейки памяти arg1 и arg2 , соответствующие x и y .
2. Установить temp равным содержимому ячейки памяти, на которую указывает arg1 (т.е. установить temp равным I_0 , где I_0 — начальное значение i). Это соответствует присвоению $\text{temp} := x$ в строке 6 в определении swap.
3. Установить содержимое ячейки, на которую указывает arg1 , равным значению, хранящемуся по адресу, на который указывает arg2 (т.е. выполнить $i := a[I_0]$). Это аналогично присвоению $x := y$ в строке 7 процедуры swap.
4. Установить содержимое ячейки, на которую указывает arg2 , равным значению temp , т.е. выполнить $a[I_0] := i$. Это соответствует присвоению $y := \text{temp}$. □

Передача параметров по ссылке используется во многих языках программирования. В Pascal по ссылке передаются параметры, описанные с использованием ключевого слова var. Массивы обычно передаются по ссылке.

Копирование-восстановление

“Гибридом” описанных способов передачи является **копирование-восстановление**, известное также как **копирование внутрь и наружу** (copy-in copy-out) и **значение-результат** (value-result).

1. Перед передачей управления вызываемой процедуре вычисляются фактические параметры. r -значения фактических параметров передаются вызываемой процедуре по значению. Однако в дополнение к этому перед вызовом определяются l -значения тех фактических параметров, которые их имеют.
2. После возвращения управления в вызывающую процедуру текущие r -значения формальных параметров копируются в l -значения фактических параметров с использованием вычисленных перед вызовом l -значений. Естественно, копируются только те фактические параметры, которые имеют l -значения.

Первый шаг выполняет “копирование внутрь” значений фактических переменных в запись активации вызываемой процедуры (в память, выделенную для формальных параметров). Второй шаг состоит в “копировании наружу” окончательных значений формальных параметров в запись активации вызывающей процедуры (т.е. в l -значения, вычисленные для фактических переменных перед вызовом).

Заметим, что $\text{swap}(i, a[i])$ корректно работает при использовании копирования-восстановления, поскольку адрес $a[i]$ вычисляется и сохраняется вызывающей процедурой перед вызовом. Таким образом, окончательное значение формального параметра y , которое является исходным значением i , копируется в нужную ячейку памяти, хотя положение $a[i]$ изменилось при вызове в связи с изменением значения i .

Копирование-восстановление используется некоторыми реализациями Fortran (хотя другие предпочитают передачу параметров по ссылке). Различия между этими двумя методами могут проявиться, если вызываемая процедура имеет несколько вариантов обращения к записи активации вызывающей процедуры. На рис. 7.31 активация $\text{unsafe}(a)$ в строке 6 может обращаться к a как к нелокальной переменной или посредством фор-

мального параметра x . При передаче параметров по ссылке присвоения значений как x , так и a непосредственно изменяют a , что дает в результате значение c , равное 0. При использовании копирования-восстановления значение a перед вызовом, равное 1, копируется в формальный параметр x . Окончательное значение x , равное 2, копируется в I -значение a непосредственно перед возвратом управления, так что окончательное значение a равно 2.

```
(1) program copyout(input, output)
(2)   var a: integer;
(3)   procedure unsafe(var x: integer);
(4)     begin x := 2; a := 0 end;
(5)   begin
(6)     a := 1; unsafe(a); writeln(a)
(7)   end.
```

Рис. 7.31. Выход этой программы различен при передаче параметра по ссылке и при копировании-восстановлении

Передача по имени

Передача по имени традиционно определяется следующим правилом копирования языка Algol.

1. Процедура рассматривается как макрос, т.е. ее тело при вызове изменяется в вызывающей процедуре таким образом, что фактические параметры побуквенно подставляются вместо формальных. Такую буквальную подстановку называют *макрорасширением* (macro-expansion), или *подстановкой тела* (in-line¹² expansion).
2. Локальные имена вызываемой процедуры содержатся отдельно от имен вызывающей процедуры. Можно считать, что каждое локальное имя вызываемой процедуры систематически переименовывается и получает уникальное имя перед макрорасширением.
3. При необходимости сохранения их целостности фактические параметры берутся в скобки.

Пример 7.8

Вызов $\text{swap}(i, a[i])$ из примера 7.7 будет реализован в виде следующего кода.

```
temp := i
      i := a[i]
a[i] := temp
```

Таким образом, при передаче параметров по имени swap , как и ожидалось, устанавливает i равным $a[i]$, но при этом неожиданно устанавливает равным I_0 (где I_0 — начальное значение i) элемент массива $a[a[I_0]]$, а не $a[I_0]$. Это происходит потому, что положение x в присвоении $x:=\text{temp}$ в процедуре swap не вычисляется до тех пор, пока оно не становится необходимым, а к тому времени значение i уже изменено. По-

¹² Заметим, что in-line expansion и inline-функции C++ представляют два разных понятия, которые не следует путать. — Прим. ред.

видимому, корректно работающая версия swap не может быть создана при условии использования передачи параметров по имени (см. [133]). □

Хотя передача параметров по имени представляет в первую очередь теоретический интерес, концептуально близкая технология подстановки тела может уменьшить время работы программы. При вызове процедур имеются определенные накладные расходы, связанные с выделением памяти для записи активации, сохранения состояния машины, настройки связей и передачи управления. Для небольших процедур код, связанный с этими расходами, может превысить размер самого тела процедуры. Существенно более эффективным может оказаться использование подстановки тела вызываемой процедуры в код вызывающей процедуры, даже если при этом программа несколько вырастет в размере. В следующем примере такое встраивание кода применяется к процедуре, использующей передачу параметров по значению.

Пример 7.9

Предположим, что функция f в присвоении $x := f(A) + f(B)$ использует передачу параметров по значению. Здесь фактически параметры A и B представляют собой выражения. Подстановка выражений A и B вместо всех появлений формальных параметров в теле f приводит к передаче параметров по имени (вспомните $a[i]$ из последнего примера).

Однако для вычисления фактических параметров перед выполнением тела процедуры можно использовать новые временные переменные.

```
t1 := A;  
t2 := B;  
t3 := f(t1);  
t4 := f(t2);  
x := t3 + t4;
```

Теперь при подстановках тела функции f все появления формального параметра заменяются на t_1 и t_2 соответственно¹³. □

Обычная реализация передачи параметров по имени заключается в передаче вызываемой процедуре подпрограмм без параметров, обычно называемых *thunks*¹⁴, которые могут вычислить *l*- и *r*-значения фактических параметров. Как и любая процедура, передаваемая как параметр в языке с лексической областью видимости, thunk содержит связь доступа, указывающую на текущую запись активации для вызывающей процедуры.

7.6. Таблицы символов

Для отслеживания области видимости и информации о связях имен компилятор использует таблицу символов. Она просматривается всякий раз, когда в исходном тексте встречается некоторое имя. При обнаружении нового имени или новой информации об имеющемся имени в таблицу символов вносятся соответствующие изменения.

¹³ С временными переменными связаны скрытые расходы. Они могут вызвать дополнительный расход памяти в записи активации. Если же локальные переменные инициализируются в записи активации, то дополнительные временные переменные приводят также к напрасным затратам времени.

¹⁴ Общепринятое перевода этого искусственного термина на русский язык не имеется, поэтому в данной книге он оставлен в оригинальном виде. — Прим. перев.

Механизм таблицы символов должен обеспечивать эффективный поиск и добавление в таблицу символов. В этом разделе представлены два механизма работы с таблицей символов — линейные списки и хеш-таблицы. Мы оценим каждую схему с точки зрения времени, необходимого для добавления *и* записей *и* запросов. Линейный список более прост в реализации, но его производительность при больших *и* и *е* невысока. Хеширование обеспечивает более высокую производительность, но за счет большего количества памяти и несколько больших усилий по программированию. Оба механизма легко адаптируются для работы в соответствии с правилом ближайшей вложенной области видимости.

При работе с таблицей символов очень полезна возможность ее динамического роста в процессе компиляции. Если размер таблицы символов зафиксирован при написании компилятора, то этот размер должен быть выбран достаточно большим, чтобы быть способным справиться с любым исходным текстом. Такой фиксированный размер будет слишком большим для большинства программ (и недостаточным для некоторых).

Записи таблицы символов

Каждый элемент таблицы символов соответствует объявлению имени. Формат элементов не обязан быть унифицированным, поскольку информация об имени зависит от его использования в программе. Каждый элемент таблицы может быть реализован в виде записи, состоящей из последовательных слов памяти. Может оказаться удобным — в целях унификации таблицы — хранить часть информации об имени вне таблицы, а в таблице содержать только указатель на эту информацию.

Информация вносится в таблицу символов неоднократно. Ключевые слова вносятся в нее изначально (если вносятся вообще). Лексический анализатор из раздела 3.4 просматривает последовательности букв и цифр в таблице символов для определения того, найдено ли обычное имя или ключевое слово. При таком подходе ключевые слова должны находиться в таблице символов еще до того, как лексический анализатор приступит к работе. Однако, если лексический анализатор может самостоятельно обрабатывать ключевые слова, то их незачем вносить в таблицу символов. Если язык не резервирует ключевые слова, они должны быть внесены в таблицу символов с предупреждением об их возможном использовании в качестве ключевых.

Запись может быть размещена в таблице символов, когда роль описываемого ею имени становится ясна, а значения его атрибутов известны. В некоторых случаях запись может быть инициализирована лексическим анализатором, как только имя появляется во входном потоке. Часто одно и то же имя может обозначать несколько различных объектов, возможно даже в одном блоке или процедуре. Например, объявление C

```
int x;  
struct x { float y, z; };
```

 (7.1)

использует x и как целое число, и как имя структуры с двумя полями. В таких случаях лексический анализатор может передать синтаксическому анализатору только имя (или указатель на лексему, образующую это имя), а не указатель на запись в таблице символов. Запись в таблице символов в таком случае создается тогда, когда выясняется синтаксическая роль этого имени. В случае описания (7.1) в таблице символов будет создано две записи — для целого x и для структуры x.

Атрибуты имени вводятся в соответствии с объявлениями, которые могут быть и неявными. Метки часто являются лишь идентификаторами с двоеточиями, так что единст-

венным действием, связанным с распознаванием такого идентификатора, может быть занесение его в таблицу символов. Аналогично синтаксис объявления процедуры определяет, что некоторые идентификаторы являются формальными параметрами.

Символы имени

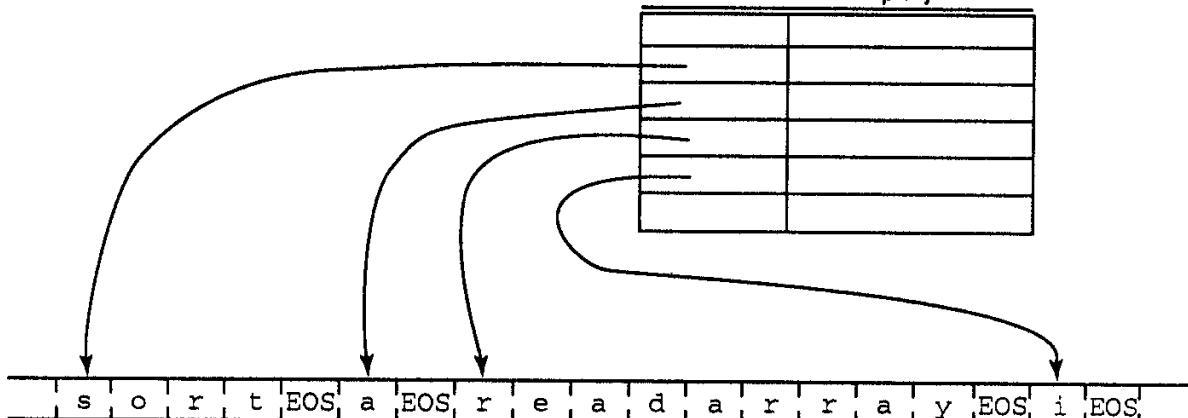
Как и в главе 3, существует различие между токеном `id` некоторого идентификатора или имени, лексемой, состоящей из образующей имя строки символов, и атрибутами этого имени. Работать со строками символов может оказаться неудобно, поэтому компилятор часто вместо лексем использует для имен некоторое представление фиксированного размера. Лексема необходима, когда запись впервые вносится в таблицу символов и когда мы ищем в таблице найденную во входном потоке лексему для определения, не была ли она уже внесена в таблицу ранее. Обычно представление имени является просто указателем на его запись в таблице символов.

Если в языке имеется ограничение на предельный размер имени, то символы имени могут храниться в записях таблицы символов, как показано на рис. 7.32а. Если ограничения на длину имени отсутствуют (или предельный размер очень редко достигается), можно использовать косвенную схему хранения, показанную на рис. 7.32б. Вместо выделения в каждой записи таблицы памяти максимального размера для хранения лексемы мы можем использовать память более эффективно, если будем хранить в таблице символов только указатель на лексему. В записи для имени мы размещаем только указатель на отдельный массив символов (*таблицу строк*), дающий позицию первого символа лексемы. Такая косвенная схема, изображенная на рис. 7.32б, позволяет размеру поля имени в записи таблицы символов оставаться постоянным.

а) В поле фиксированного размера внутри записи

Имя	Атрибуты
s o r t	
a	
r e a d a r r a y	
i	

Имя Атрибуты



б) В отдельном массиве

Рис. 7.32. Хранение символов имени

Лексема, составляющая имя, должна быть сохранена полностью, чтобы гарантировать, что все появления одного и того же имени могут быть связаны с одной и той же записью в таблице символов. Однако мы должны различать появления одной и той же лексемы в областях видимости разных объявлений.

Информация о выделенной памяти

В таблице символов, помимо прочего, хранится и информация о памяти, которая будет связана с именами в процессе работы программы. Сначала рассмотрим имена со статически выделяемой для них памятью. Если целевой код представляет собой язык ассемблера, мы можем оставить задачу выделения памяти для различных имен ему. Все, что нам нужно сделать, — это после генерации ассемблерного кода сканировать таблицу символов и создать для каждого имени определение данных на языке ассемблера для добавления его к сгенерированной ассемблерной программе.

Если же компилятор генерирует машинный код, то должно быть установлено положение каждого объекта данных относительно некоторой фиксированной позиции, например начала записи активации. То же применимо и к блокам данных, загружаемых в качестве отдельного от программы модуля. Например, в языке Fortran блоки COMMON загружаются отдельно; должны быть также определены положения имен относительно начала блока COMMON, в котором они находятся. По причинам, обсуждаемым в разделе 7.9, подход из раздела 7.3 должен быть модифицирован для языка Fortran, в котором мы должны назначить смещения именам после просмотра всех объявлений процедуры и обработки инструкций EQUIVALENCE.

В случае имен, которые будут размещены в стеке или куче, компилятор не выделяет память вообще, а только планирует записи активации для каждой процедуры, как описывалось в разделе 7.3.

Использование списков для представления таблицы символов

Простейшая и наиболее легкая в реализации структура данных для представления таблицы символов — линейный список записей, показанный на рис. 7.33. Мы используем единый массив (или, что эквивалентно, несколько массивов) для хранения имен и связанной с ними информации. Новые имена добавляются в список в том порядке, в котором они встречаются в исходном тексте. Положение конца списка отмечается указателем *available*, указывающим, куда будет вноситься запись о следующем имени. Поиск имени в списке производится в обратном порядке — от указателя *available* к началу списка. При обнаружении имени связанная с ним информация может быть найдена в следующих за именем словах памяти. Если имя не найдено до достижения начала массива, значит, искомое имя в списке отсутствует.

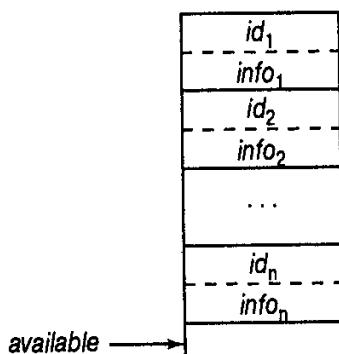


Рис. 7.33. Линейный список записей

Заметим, что создание записи для имени и поиск имени в таблице символов представляют собой две независимые операции — мы можем выполнять одну из них без другой. В языках с блочной структурой появление имени находится в области видимости ближайшего вложенного объявления этого имени. Это правило области видимости можно реализовать путем создания новой записи для имени при каждом его объявлении. Новая запись для имени располагается в памяти непосредственно за указателем *available*; указатель при этом увеличивается на размер записи таблицы символов. Поскольку записи вносятся по порядку, начиная от начала массива, они располагаются в порядке их создания, и поиск от указателя *available* к началу массива гарантирует, что мы находим запись для имени, созданную последней.

Если таблица символов содержит *n* имен, операция внесения нового имени без проверки его наличия в таблице требует фиксированного времени. Если многократные записи для одного имени не разрешены, то для выяснения, имеется ли данное имя в таблице, мы должны осуществить поиск, время которого пропорционально *n*. В среднем при поиске в таблице информации, связанной с именем, мы должны просмотреть $n/2$ записей, так что стоимость одного запроса также пропорциональна *n*. Таким образом, общее время, необходимое для внесения *n* имен и выполнения *e* запросов к таблице символов, не больше, чем $c_1(n+e)$, где константа *c* представляет время, необходимое для выполнения нескольких машинных операций. В программе среднего размера у нас может оказаться $n \sim 100$ и $e \sim 1000$, так что для работы с таблицей символов потребуется несколько сотен тысяч машинных операций. Это не так уж и страшно, поскольку речь идет всего лишь о реальном времени, меньшем секунды. Однако стоит увеличить *n* и *e* в 10 раз, и время работы возрастет в 100 раз, начиная доставлять неприятности. Ценные данные о времени, затраченном на те или иные операции (и влияющие на принятие решения об использовании линейных списков), может дать так называемый профиль программы, т.е. информация о времени выполнения тех или иных фрагментов кода, получаемая с помощью специальных программ — профайлеров.

Хеш-таблицы

Во многих компиляторах реализованы варианты технологии поиска, известной под названием *хеширование*¹⁵. Здесь будет рассмотрен простой вариант, именуемый *открытым хешированием*. Открытость в данном случае означает отсутствие ограничения на количество элементов в таблице. Даже такая схема дает нам возможность выполнить *e* запросов на *n* имен за время $n(n+e)/m$, где константа *m* может быть выбрана нами и сделана сколь угодно большой — вплоть до *n*. В связи с этим данный метод в целом более эффективен, чем линейные списки, и используется для таблиц символов в большинстве случаев. Как можно ожидать, при этом методе память, требуемая структурой данных, растет с увеличением *m*, так что “выигрываем в скорости — проигрываем в количестве памяти”.

Базовая схема открытого хеширования приведена на рис. 7.34. Структура данных состоит из двух частей — хеш-таблицы и блоков.

1. Хеш-таблица представляет собой фиксированный массив из *m* указателей на записи таблицы.

¹⁵ Подробнее вопросы хеширования рассмотрены в книге Кнут Д. Э. *Искусство программирования. Т. 3. Сортировка и поиск*, 2-е изд. — М.: Издательский дом “Вильямс”, 2000; в разделе 6.4, к которому рекомендуется обратиться заинтересованному читателю. — Прим. перев.

2. Записи таблицы организованы в виде m отдельных связанных списков, именуемых **блоками**¹⁶ (некоторые блоки могут быть пустыми). Каждая запись в таблице символов встречается только в одном из этих списков. Память для этих записей может быть представлена в виде массива записей, который обсуждался в предыдущем разделе. Можно использовать и динамическое выделение памяти для записей, что зачастую приводит к определенной потере эффективности.

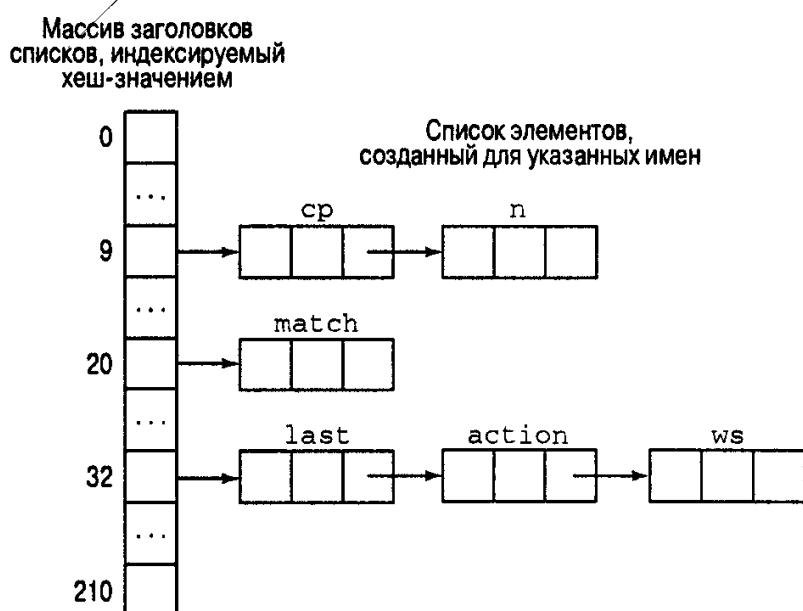


Рис. 7.34. Хеш-таблица размера 211

Для определения, существует ли в таблице символов запись для строки s , мы вычисляем хеш-функцию h от строки s , возвращающую целое число от 0 до $m-1$. Если строка s имеется в таблице символов, то она находится в списке с номером $h(s)$. Если s в таблице символов еще отсутствует, то она вносится в таблицу путем создания записи в начале списка с номером $h(s)$.

В качестве грубой оценки примем, что средняя длина списка при n записях в таблице символов и хеш-таблице размера m — n/m . Выбирая m настолько большим, чтобы n/m было ограничено небольшой величиной, например 2, мы, по сути, делаем время обращения к элементу таблицы константой.

Пространство, занимаемое таблицей символов, состоит из m слов для хеш-таблицы и $c n$ слов для записей таблицы символов, где c — количество слов в одной записи таблицы символов. Таким образом, размер хеш-таблицы зависит только от m , а память для записей таблицы символов — только от количества записей¹⁷.

Выбор m зависит от предполагаемого использования таблицы символов. Если m порядка нескольких сотен, то на просмотр таблицы тратится незначительная часть общего времени работы компилятора даже для программ среднего размера. Однако если вход-

¹⁶ В оригинале — *buckets*, т.е. ведро. Здесь мы воспользовались переводом этого термина в издании Кнут Д. Э. *Искусство программирования. Т. 3. Сортировка и поиск*, 2-е изд. — М.: Издательский дом “Вильямс”, 2000. — Прим. перев.

¹⁷ Это справедливо при динамическом выделении памяти для элементов таблицы символов. При выделении памяти в виде массива для последующего размещения в нем записей таблицы символов (метод, описанный в предыдущем разделе) нельзя говорить о строгой пропорциональности памяти, необходимой для записей таблицы символов. — Прим. ред.

ной поток компилятора генерируется другой программой, то количество имен может резко вырасти по сравнению с большинством программ того же размера, создаваемых человеком. В таком случае желательны большие размеры хеш-таблицы.

Большое внимание следует уделить вопросу создания хеш-функции, которая должна легко вычисляться для строк символов и при этом равномерно распределять строки среди m списков.

Один из пригодных подходов к вычислению хеш-функции состоит в следующем.

1. Определим положительное целое число h по символам c_1, c_2, \dots, c_k строки s . Преобразование отдельного символа в целое число обычно поддерживается языками программирования. Например, Pascal предоставляет для этого функцию *ord*, а С автоматически преобразовывает символ в целое число при выполнении арифметических операций.
2. Преобразуем полученное целое число h в номер списка, т.е. в целое число между 0 и $m-1$. Вполне разумно сделать это путем деления по модулю m . Взятие остатка лучше работает в случае простого m , так что выбор 211 на рис. 7.34 более удачен, чем, например, 200.

Хеш-функция, учитывающая все символы строки, выглядит более разумной, чем функция, учитывающая только несколько символов в начале или середине строки. Учтите, что входной поток компилятора может генерироваться другой программой и содержать имена специализированного вида во избежание конфликтов с именами, используемыми другими программами или программистом; люди также склонны к "кластеризации" имен (например, *baz*, *newbaz*, *baz1* и т.д.).

Простейшая идея состоит в сложении всех целых чисел, представляющих символы строки. Несколько лучший результат можно получить, умножая старое значение h на константу α перед прибавлением следующего символа. Таким образом, $h_0 = 0$, $h_i = \alpha h_{i-1} + c_i$, $1 \leq i \leq k$ и $h = h_k$, где k — длина строки (вспомните, что хеш-значение определяется как $h \bmod m$). Простое сложение символов представляет собой частный случай при $\alpha=1$. Аналогичная стратегия может использовать вместо сложения операцию исключающего или: $c_i \text{ xor } \alpha h_{i-1}$.

Если для 32-битовых целых чисел принять $\alpha=65599$, простое число, близкое к 2^{16} , то вскоре при вычислении αh_{i-1} будет получено переполнение. Поскольку α — простое число, игнорирование переполнения и работа с 32 младшими битами произведения дают неплохие результаты.

В ряде экспериментов неплохие результаты для всех размеров таблиц показала хеш-функция *hashpjw* (рис. 7.35) из компилятора С Вайнбергера (рис. 7.36). Размеры таблицы включали первые простые числа, большие 100, 200, ..., 1500. Близкие результаты дала хеш-функция, построенная путем умножения старого хеш-значения на 65599 с игнорированием переполнения. Функция *hashpjw* вычисляется с использованием стартового значения $h=0$. Для каждого символа c производится сдвиг h на 4 позиции влево и суммирование с c . Если любой из 4 старших битов равен 1, сдвигаем их вправо на 24 позиции и выполняем операцию исключающего или со значением h и сбрасываем значения 4 старших битов в 0.

```
(1) #define PRIME 211
(2) #define EOS    '\0'
(3) int hashpjw(s)
(4) char * s;
```

```

(5)  {
(6)      char * p;
(7)      unsigned h = 0, g;
(8)      for( p = s; *p != EOS; p++ )
(9)          if (g = (h = (h << 4) + *p) & 0xFFFFFFFF)
(10)             h ^= g >> 24 ^ g;
(11)      return h % PRIME;
(12)  }

```

Рис. 7.35. Текст хеш-функции *hashpjw* на языке С

Пример 7.10

Для получения наилучших результатов при разработке хеш-функции во внимание должны приниматься размер хеш-таблицы и ожидаемый входной поток компиляции. Например, желательно, чтобы хеш-значения для наиболее часто встречающихся в языке имен были различны. Если ключевые слова вносятся в таблицу символов, то их можно рассматривать в качестве часто встречающихся имен, хотя в одном примере программы на языке С имя *i* встречалось в три раза чаще, чем *while*.

Один из путей тестирования хеш-функции — проверка числа строк, попадающих в один и тот же список. Пусть для данного файла F, состоящего из n строк, в список j ($0 \leq j \leq m-1$) попадает b_j строк. Меру равномерности распределения строк по спискам можно получить вычислением

$$\sum_{j=0}^{m-1} b_j(b_j+1)/2 \quad (7.2)$$

Интуитивное пояснение этого выражения состоит в следующем. Нам требуется просмотреть 1 элемент списка, чтобы найти первую запись в списке j ; 2 элемента — для поиска второй записи и так до b_j элементов для поиска последней записи. Сумма 1, 2, ..., b_j равна $b_j(b_j+1)/2$.

Из упр. 7.14 следует, что значение (7.2) для хеш-функции, случайным образом распределяющей строки по блокам, равно

$$(n/2m)(n+2m-1). \quad (7.3)$$

Значение отношения $\sum_{j=0}^{m-1} \frac{b_j(b_j+1)/2}{(n/2m)(n+2m-1)}$ для разных хеш-функций, примененных к

9 файлам, показано на рис. 7.36. Эти файлы представляют собой следующее.

1. 50 наиболее часто встречающихся имен и ключевых слов в наборе С-программ.
2. То же, что и (1), но выборка насчитывает 100 имен.
3. То же, что и (1), но выборка насчитывает 500 имен.
4. 952 внешних имени в ядре операционной системы UNIX.
5. 627 имен в С-программе, получаемой в результате работы транслятора¹⁸ C++ ([415]).
6. 915 сгенерированных случайным образом строк.

¹⁸ Имеется в виду ранний транслятор C++ Бьерна Страуструпа, генерировавший из исходного текста на языке C++ промежуточный код на языке С, в дальнейшем компилируемый в исполняемую программу. — Прим. ред.

7. 614 слов из раздела 3.1 оригинального издания этой книги.
8. 1201 английское слово, к которым добавлены суффиксы и префиксы xxx .
9. 300 имен $v100, v101, \dots, v399$.

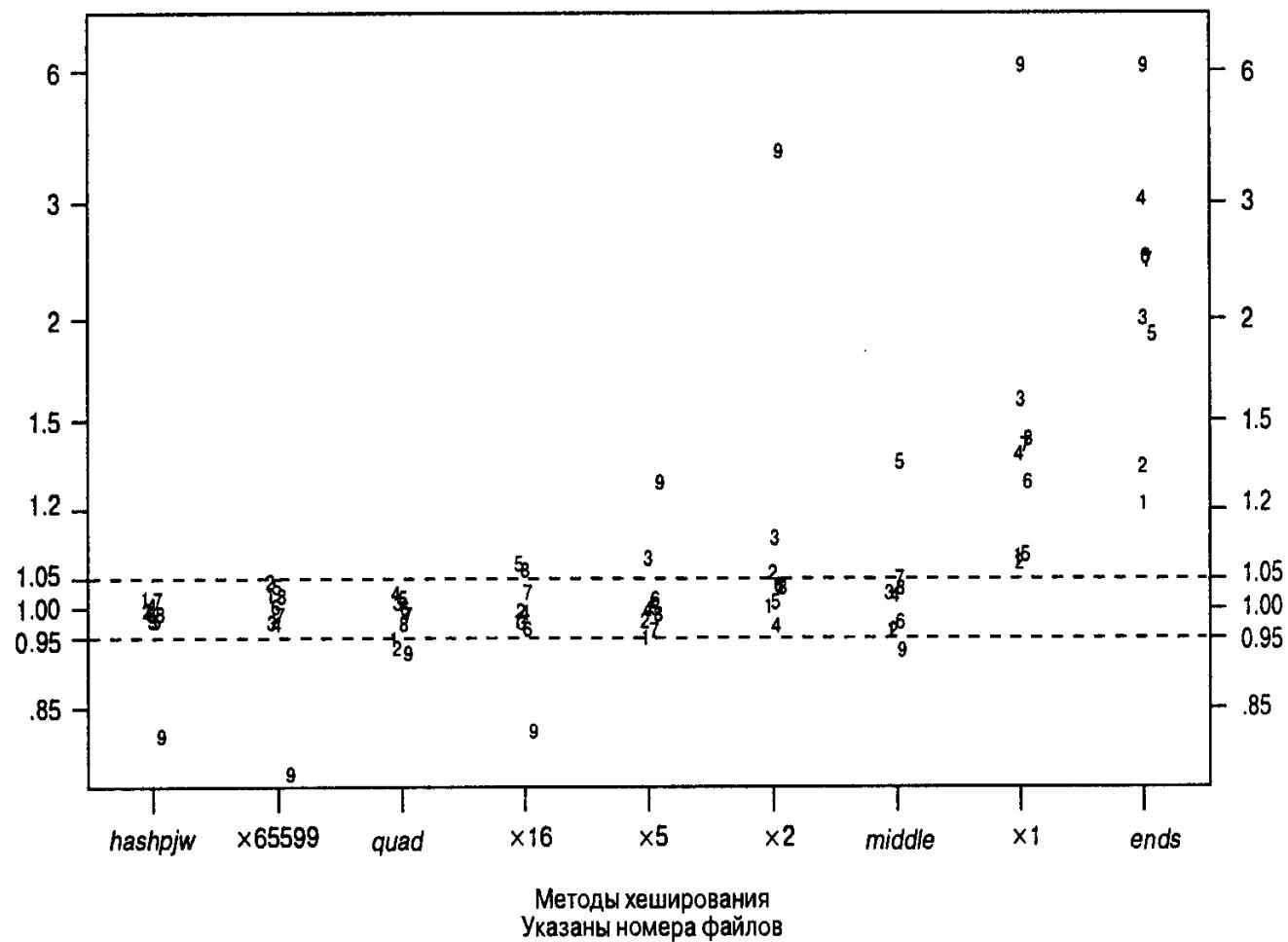


Рис. 7.36. Относительное качество хеш-функций для таблицы размера 211

Функция *hashpjw* приведена на рис. 7.35. Функции, обозначаемые $\times\alpha$, где α — целая константа, вычисляют значение $h \bmod m$, где h вычисляется итеративно, начиная со значения 0, путем умножения на α и суммирования со следующим символом в строке. Функция *middle* строит h по средним четырем символам строки; *ends* суммирует три первых и три последних символа с длиной строки. И наконец, *quad* группирует каждые четыре последовательных символа строки в целое число, а затем суммирует эти числа. \square

Представление информации об области видимости

Записи таблицы символов соответствуют объявлениям имен. Когда в исходном тексте встречается некоторое имя, поиск в таблице символов должен вернуть запись для соответствующего объявления. Правила области видимости исходного языка определяют, какое объявление должно использоваться.

Простейший подход состоит в поддержании отдельных таблиц символов для каждой области видимости. По существу, таблица символов для процедуры или области видимости представляет собой эквивалент записи активации времени компиляции. Информация о нелокальных по отношению к процедуре именах может быть найдена сканированием таблицы символов охватывающей процедуры в соответствии с правилами области видимости.

ности языка. Аналогично информация о локальных по отношению к процедуре именах может быть приложена к узлу синтаксического дерева программы, соответствующему процедуре. При таком подходе таблица символов интегрируется в промежуточное представление входного потока.

Правило ближайшей вложенной области видимости может быть реализовано посредством адаптации структур данных, представленных ранее в этом разделе. Мы отслеживаем локальные имена процедуры, давая каждой процедуре уникальный номер. В случае языка с блочной структурой дополнительно должны быть пронумерованы составляющие программу блоки. Номер каждой процедуры может быть вычислен синтаксически управляемым способом на основе семантических правил, распознающих начало и конец каждой процедуры. Номер процедуры становится частью всех локальных имен, объявленных в этой процедуре; представление локального имени в таблице символов является парой значений, состоящей из имени и номера процедуры (при некоторых организациях таблицы символов, например при описанном ранее размещении записей в строгом порядке в линейном списке, можно обойтись без номера процедуры, поскольку он может быть вычислен по положению записи в таблице символов).

Когда мы осуществляем поиск прочитанного имени, оно соответствует записи только в том случае, когда все символы имен в точности соответствуют друг другу и номер процедуры в записи таблицы символов равен номеру текущей процедуры. Большинство правил ближайшей вложенной области видимости реализуется с помощью следующих операций с именами.

lookup

Поиск наиболее поздней записи (с данным именем)

insert

Создание новой записи

delete

Удаление наиболее поздней записи

“Удаленные” записи должны быть сохранены; они всего лишь убираются из активной таблицы символов. В однопроходном компиляторе информация в таблице символов об области видимости, состоящей, например, из тела процедуры, после обработки тела процедуры перестает быть нужной. Однако она может быть нужна в процессе работы программы, в особенности при реализации диагностической системы времени исполнения. В этом случае информация из таблицы символов должна быть добавлена к генерированному коду для использования компоновщиком или диагностической системой времени исполнения (см. также обсуждение вопроса о полях имен в записях в разделах 8.2 и 8.3).

Обсуждаемые в этом разделе структуры данных — списки и хеш-таблицы — могут быть организованы так, чтобы поддерживать указанные операции.

При описании линейного списка, состоящего из массива записей, мы упоминали, что операция *lookup* может быть реализована путем вставки записей в конец массива, чтобы обеспечить тот же порядок записей в массиве, что и порядок вставки. Сканирование начинается с конца списка и выполняется по направлению к началу массива, находя для данного имени запись, созданную последней. Эта ситуация схожа со связным списком, показанным на рис. 7.37. Здесь указатель *front* определяет последнюю созданную запись в списке. Операция *insert* выполняется за фиксированное время, поскольку новая запись размещается в начале списка. Реализация *lookup* выполняется сканированием списка, начиная с записи, на которую указывает *front*, и следуя связям списка до тех пор, пока не будет найдено требуемое имя или не будет достигнут конец списка. На рис. 7.37 запись для а, объявленного в блоке B_2 , вложенном в блок B_0 , появляется в списке ближе к его началу, чем запись для а, объявленного в блоке B_0 .

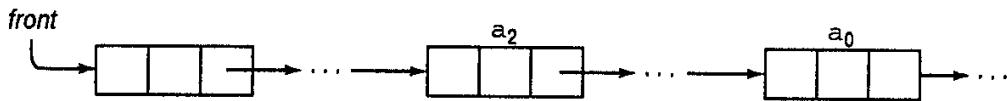


Рис. 7.37. Последняя запись для a находится ближе к началу списка

Рассматривая операцию *delete*, заметим, что записи, соответствующие объявлению в наиболее глубоко вложенной процедуре, располагаются наиболее близко к началу списка. Следовательно, нет необходимости хранить в каждой записи номер процедуры. Если такая информация имеется в первой записи для процедуры, то по окончании работы с областью видимости данной процедуры могут быть удалены все записи от начала списка до указанной записи включительно.

Хеш-таблица состоит из m списков, обращение к которым осуществляется посредством массива. Поскольку одно и то же имя всегда хешируется в один и тот же список, отдельные списки обрабатываются точно так же, как и список, показанный на рис. 7.37. Однако для реализации операции *delete* мы можем обойтись и без сканирования всей хеш-таблицы в поисках списков, содержащих удаляемые записи, используя, например, следующий подход. Предположим, что каждая запись имеет две связи.

1. Хеш-связь, которая связывает запись с другими, имена которых хешируются в одно и то же значение.
2. Связь области видимости, связывающая все записи одной области видимости.

Если связь области видимости оставить нетронутой при удалении записи из хеш-таблицы (по сути, осуществив удаление только соответствующими изменениями хеш-связей), то цепочка, образованная связями области видимости, будет составлять отдельную (неактивную) таблицу символов для рассматриваемой области видимости.

Удаление записей из хеш-таблицы должно выполняться с осторожностью, поскольку это влияет на предыдущую запись в списке. Вспомним, что удаление i -й записи выполняется путем настройки указателей таким образом, что $(i-1)$ -я запись указывает на $(i+1)$ -ю, так что просто использовать связи области видимости для поиска i -й записи недостаточно. $(i-1)$ -я запись может быть найдена, если хеш-связи образуют циклический связный список, в котором последняя запись указывает на первую. Иной подход, которым можно воспользоваться для отслеживания списков, содержащих записи, подлежащие удалению, — это использование стека. При сканировании новой процедуры в стек помещается соответствующий маркер. Выше этого маркера содержатся номера списков, содержащих записи для объявленных в этой процедуре имен. По окончании обработки процедуры список номеров снимается со стека до тех пор, пока не будет обнаружен маркер процедуры. Еще одна схема рассматривается в упр. 7.11.

7.7. Возможности языков по динамическому выделению памяти

В этом разделе мы вкратце опишем возможности, обеспечиваемые некоторыми языками для динамического выделения памяти для данных под управлением программы. Память для таких данных обычно берется из кучи. Выделенные данные часто сохраняются до тех пор, пока память не будет освобождена явным образом. Распределение памяти может выполняться явно или неявно. В Pascal, например, явное выделение памяти выполняется с помощью стандартной процедуры `new`. При выполнении

`new(p)`, где `p` — указатель на некоторый тип данных, выделяется память для объекта этого типа; после выполнения `p` указывает на вновь созданный объект. В большинстве реализаций Pascal для явного освобождения выделенной памяти используется процедура `dispose`.

Неявное распределение памяти выполняется при вычислении выражения, приводящего к выделению памяти для хранения полученного результата. Например, при использовании `cons` Lisp выделяет ячейку в списке; ячейки, обращений к которым больше не будет, автоматически освобождаются. Snobol допускает изменения длины строки в процессе работы, управляя при этом памятью в куче, необходимой для хранения строки.

Пример 7.11

Pascal-программа, приведенная на рис. 7.38, строит связанный список, показанный на рис. 7.39, и выводит целые числа, хранящиеся в ячейках списка. Выход программы таков.

```
76      3  
 4      2  
 7      1
```

Когда программа начинает работу со строки 15, память для указателя `head` располагается в записи активации программы. Всякий раз, когда управление попадает в строку

```
(11)      new(p); p↑.key := k; p↑.info := i;
```

вызов `new(p)` приводит к выделению памяти в некотором месте кучи; `p↑` указывает на эту память в присвоениях в строке 11.

```
(1) program table(input, output);  
(2) type link = ^cell;  
(3)     cell = record  
(4)         key, info : integer;  
(5)         next    : link  
(6)     end;  
(7) var head : link;  
(8) procedure insert(k, i : integer);  
(9)     var p : link;  
(10)    begin  
(11)        new(p); p↑.key := k; p↑.info := i;  
(12)        p↑.next := head; head := p;  
(13)    end;  
(14) begin  
(15)     head := nil;  
(16)     insert(7,1); insert(4,2); insert(76,3);  
(17)     writeln(head↑.key, head↑.info);  
(18)     writeln(head↑.next↑.key, head↑.next↑.info);  
(19)     writeln(head↑.next↑.next↑.key,  
                  head↑.next↑.next↑.info);  
(20) end.
```

Рис. 7.38. Динамическое размещение ячеек с использованием `new` в Pascal

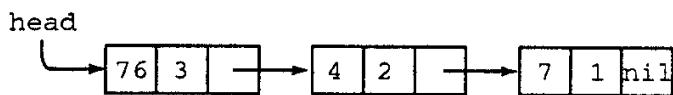


Рис. 7.39. Связанный список, построенный программой, приведенной на рис. 7.38

Обратите внимание, что вывод программы производится после того, как управление возвращается в основную программу из процедуры `insert`. Другими словами, выделенная в активации `insert` с помощью `new` память сохраняется при возврате управления из активации в основную программу. □

Мусор

Динамически выделенная память может стать недоступной. Выделенная программой память, к которой невозможно обратиться, называется *мусором* (garbage). Предположим, что на рис. 7.38 между строками 16 и 17 `head^.next` получает значение `nil`.

```
(16)      insert(7,1); insert(4,2); insert(76,3);
          head^.next := nil;
(17)      writeln(head^.key, head^.info);
```

Теперь крайняя слева ячейка списка на рис. 7.39 вместо указателя на среднюю ячейку содержит указатель “в никуда” `nil`. Поскольку указатель на среднюю ячейку потерян, средняя и правая ячейки становятся мусором.

Lisp выполняет *сборку мусора* (обсуждаемую в следующем разделе), которая восстанавливает недоступную память. В Pascal и C сборка мусора не предусмотрена, и программа должна явным образом освободить неиспользуемую память. Освобожденная память в этих языках может использоваться повторно, но мусор остается мусором до завершения программы.

Висячие ссылки

Явное освобождение памяти может вызвать определенные сложности, выражавшиеся в висячих (оборванных, dangling) ссылках. Как упоминалось в разделе 7.3, висячие ссылки возникают при обращении к освобожденной памяти. В качестве примера рассмотрим влияние вызова `dispose(head^.next)` между 16 и 17 строками листинга, приведенного на рис. 7.38.

```
(16)      insert(7,1); insert(4,2); insert(76,3);
          dispose(head^.next)
(17)      writeln(head^.key, head^.info);
```

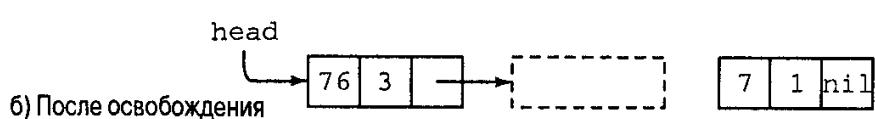
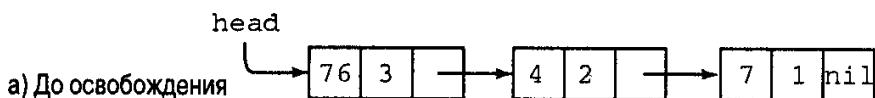


Рис. 7.40. Создание висячей ссылки и мусора

Вызов `dispose` освобождает ячейку, следующую за той, на которую указывает `head`, как показано на рис. 7.40. Однако указатель `head.next` не изменяется, так что он становится “висячим указателем” на освобожденную память.

Висячие ссылки и мусор являются тесно связанными понятиями; висячие указатели возникают, если освобождение происходит до последнего обращения, а мусор существует после последнего обращения и до освобождения памяти.

7.8. Технологии динамического распределения памяти

Технологии, необходимые для реализации динамического распределения памяти, зависят от способа освобождения выделенной памяти. Если освобождение производится неявно, за определение того, что некоторый блок памяти более не используется, отвечает пакет поддержки времени исполнения. При явном освобождении памяти программистом задача компилятора упрощается. Вначале мы рассмотрим явное освобождение выделенной памяти.

Явное выделение блоков фиксированного размера

Простейший вид динамического распределения памяти предполагает работу с блоками фиксированного размера. Связывая блоки в список, как показано на рис. 7.41, мы легко выполняем выделение и освобождение блоков с минимальными накладными расходами памяти (или вовсе без таковых).

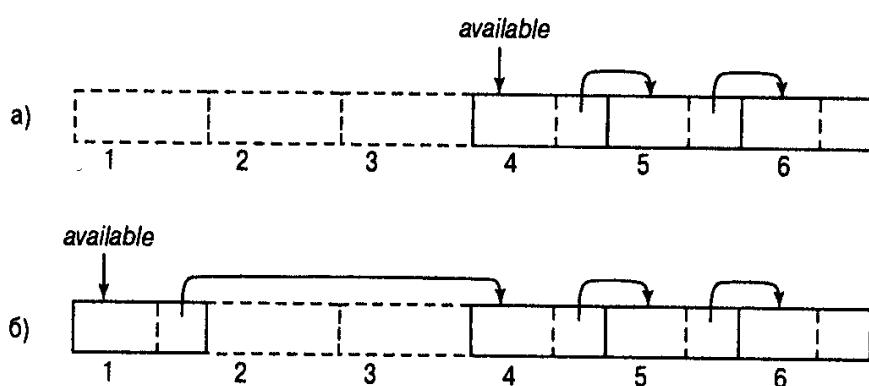


Рис. 7.41. Освобожденный блок добавляется в список свободных блоков

Предположим, что блоки берутся из непрерывной области памяти. Инициализация этой области выполняется путем использования части каждого блока для хранения указателя на следующий блок. Первый свободный блок определяется указателем `available`. При таком подходе выделение блока состоит в его удалении из списка, а освобождение — во внесении в список.

Подпрограммы компилятора, управляющие блоками, не обязаны знать тип объекта, хранимого программой в данном блоке. Каждый блок можно рассматривать как варианту запись, которую компилятор трактует как связь со следующим свободным блоком, а пользовательскую программу — как запись некоторого другого типа (используемого в программе). Следовательно, накладные расходы памяти отсутствуют, так как для своих целей пользовательская программа может использовать весь блок целиком. При освобождении блока подпрограммы компилятора используют часть блока для связи его с другими блоками в списке свободных блоков, как показано на рис. 7.41.

Явное выделение блоков переменного размера

При выделении и освобождении блоков память может стать *фрагментированной*; это означает, что куча состоит из чередующихся свободных и используемых блоков, как показано на рис. 7.42.



Рис. 7.42. Свободные и используемые блоки в куче

Ситуация, показанная на рис. 7.42, может возникнуть, например, при выделении программой пяти блоков с последующим освобождением второго и четвертого. Фрагментация не представляет никакой сложности при использовании блоков фиксированного размера, но при работе с блоками произвольного размера превращается в проблему, поскольку мы не можем выделить блок, больший имеющихся свободных блоков — даже если суммарная свободная память превосходит затребованную.

Один из методов выделения блоков переменного размера называется методом *первого подходящего*. При выделении блока размера s мы находим первый свободный блок размера $f \geq s$. Далее этот блок разбивается на два блока — с размерами s и $f-s$ ¹⁹. Заметим, что такое выделение приводит к определенным накладным расходам времени, связанным с поиском достаточно большого блока.

При освобождении блока мы проверяем, не соседствует ли он со свободным блоком. Если это так, освобождаемый блок объединяется со свободным для получения большего свободного блока, что позволяет снизить остроту проблемы фрагментации. Существует множество методов выделения и освобождения блоков и работы со списками свободных блоков, каждый из которых имеет свои достоинства и недостатки и обеспечивает некоторый компромисс между необходимыми для данного метода временем, памятью и доступностью блоков большого размера. Детально ознакомиться с этими вопросами читатель может в книгах [8, 257].

Неявное освобождение

Неявное освобождение требует сотрудничества между пользовательской программой и пакетом поддержки времени исполнения, поскольку последний должен знать о том, когда блок более не используется. Такое сотрудничество реализуется путем определения формата блоков памяти. Предположим, что формат блока памяти такой, как показано на рис. 7.43.

Первая проблема заключается в распознавании границ блока. При фиксированном размере блока для этого достаточно информации о положениях блоков (например, если каждый блок занимает 20 слов, то новые блоки начинаются через каждые 20 слов). В противном случае мы должны хранить размеры блоков, чтобы иметь возможность определить, где начинается следующий блок.

Вторая проблема состоит в распознавании используемых блоков. Мы считаем, что блок используется, если пользовательская программа может обратиться к нему. Такое обращение может произойти посредством указателя (или последовательности указате-

¹⁹ На самом деле при использовании блоков переменного размера, как правило, имеются определенные накладные расходы памяти, так что при наличии одного разделяемого блока размера f сумма выделяемых программе частей этого блока обычно меньше f . — Прим. ред.

лей), так что компилятор должен знать положение всех указателей программы в памяти. При использовании формата блока, приведенного на рис. 7.43, указатели содержатся в определенной позиции в блоке. Можно, кстати, предполагать, что в области информации пользователя в блоке нет ни одного указателя.

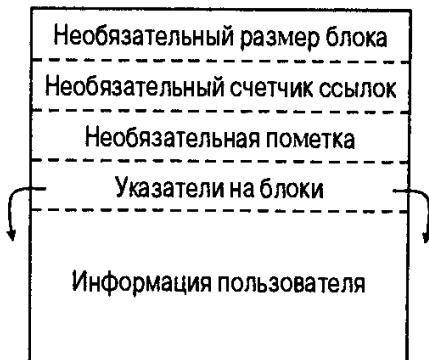


Рис. 7.43. Формат блока памяти

Для неявного освобождения могут использоваться два подхода, которые мы вкратце опишем здесь (более подробно этот вопрос изложен в [8]).

1. **Счетчики ссылок.** Мы отслеживаем количество блоков, непосредственно указывающих на данный блок. Если число таких блоков упадет до 0, блок может быть удален, поскольку на него никто не ссылается. Другими словами, блок становится мусором, который можно убрать. Поддержка счетчиков ссылок может занимать много времени; так, обычное присвоение указателей $p := q$ приводит к изменению счетчика ссылок блоков, на которые указывают p и q . Счетчик блока, на который до присвоения указывал p , уменьшается на единицу, а счетчик блока, на который указывает q , увеличивается. Счетчики ссылок лучше всего использовать, когда указатели между блоками не образуют замкнутых циклов. Например, на рис. 7.44 ни один из блоков недоступен из других, так что они представляют собой мусор, но при этом счетчик ссылок у каждого из них равен 1.
2. **Технология пометок.** Альтернативный подход состоит во временном приостановлении выполнения пользовательской программы и использовании "замороженных" указателей для определения, какие из блоков используются в настоящий момент. Этот подход требует, чтобы были известны все указатели в куче. Образно говоря, мы льем в кучу краску через сито из указателей, и после этого все используемые блоки оказываются окрашенными, а неокрашенные блоки могут быть освобождены. Говоря более подробно, мы проходим по куче и помечаем все блоки как неиспользуемые. После этого, следуя указателям программы, мы помечаем достигнутые таким образом блоки как используемые. При последнем сканировании кучи производится сборка всех блоков, оставшихся непомеченными.



Рис. 7.44. Неиспользуемые блоки с ненулевыми счетчиками ссылок

При использовании блоков переменного размера мы получаем дополнительную возможность перемещения блоков памяти из их текущих положений (то же можно делать и

с блоками фиксированного размера, но в этом случае мы не получим никакого преимущества). Этот процесс перемещения блоков памяти, именуемый *уплотнением*, позволяет переместить все используемые блоки к одному концу кучи, тем самым объединяя всю свободную память в один большой блок. При уплотнении также требуется информация о всех указателях на данный блок, поскольку они должны быть корректно обновлены, чтобы верно отражать новое положение блока. Уплотнение избавляет от проблемы фрагментации свободной памяти.

7.9. Распределение памяти в Fortran

Fortran разрабатывался как язык со статическим распределением памяти, описанным в разделе 7.3. Однако имеется ряд специфичных для Fortran вопросов распределения памяти, таких как обработка объявлений COMMON или EQUIVALENCE. Компилятор Fortran может создавать ряд *областей данных* (data areas), т.е. блоков памяти, в которых могут храниться значения объектов. В Fortran существует по одной области данных для каждой процедуры и по одной — для каждого именованного блока COMMON и для непомеченног COMMON, если таковой используется. Таблица символов должна записывать для каждого имени область данных, частью которой является имя, и его смещение в этой области (т.е. положение относительно ее начала). И наконец, компилятор должен принять решение о размещении областей данных относительно исполнимого кода и друг друга (однако взаимное расположение областей произвольно, поскольку области не зависят друг от друга).

Компилятор должен вычислить размер каждой области данных. Для областей данных процедур достаточно одного счетчика, поскольку их размеры известны после обработки каждой из процедур. Запись для каждого блока COMMON должна сохраняться в процессе обработки всех процедур, поскольку каждая процедура, использующая блок, может иметь собственные представления о его размере, и реальный размер блока определяется как максимальное значение его размера у разных процедур. Если процедуры компилируются раздельно, выбирать размер блока COMMON должен редактор связей, используя для этого максимальный из размеров блоков с данным именем среди всех связываемых частей скомпилированного кода.

Для каждой области данных компилятор создает *карту памяти*, которая представляет собой описание содержимого области. Эта карта памяти может в простейшем случае состоять из указания в записях таблицы символов смещений каждого имени в данной области. От нас не требуется легкий способ ответа на вопрос “Чем являются имена в данной области данных?”, однако в Fortran для областей данных процедур ответ на этот вопрос нам известен, поскольку все имена, объявленные в процедуре и не являющиеся общими (именами COMMON) или эквивалентными общим, располагаются в области данных процедуры. Записи таблицы символов для имен COMMON могут быть связаны (по одной цепочке для каждого блока COMMON) в порядке их появления в блоке. В действительности, так как смещения имен в области данных не всегда могут быть определены до окончания обработки процедуры (так, массивы Fortran могут быть объявлены до того, как будут объявлены их размерности), создание цепочек имен COMMON становится необходимостью.

Программа на языке Fortran состоит из основной программы, подпрограмм и функций (все они называются *процедурами*). Каждое появление имени имеет область видимости, состоящую только из одной процедуры. Мы можем генерировать объектный код каждой процедуры по достижении конца этой процедуры. В этом случае можно удалить

большую часть информации из таблицы символов. Мы должны сохранить только имена, внешние по отношению к только что обработанной процедуре, т.е. имена других процедур и общих блоков. Эти имена могут и не быть истинно внешними по отношению ко всей компилируемой программе, но должны быть сохранены до тех пор, пока не будет обработано все множество процедур.

Данные в областях COMMON

Для каждого блока мы создаем запись, которая задает первое и последнее имена, принадлежащие текущей процедуре и объявленные в блоке COMMON. При обработке объявлений типа

COMMON /BLOCK1/ NAME1, NAME2

компилятор должен выполнить следующее.

1. В таблице имен блоков COMMON создать запись для BLOCK1 (если таковая еще не существует).
2. В записях таблицы символов для NAME1 и NAME2 установить указатель на запись таблицы символов для BLOCK1, определяющий, что они описаны как COMMON и являются членами BLOCK1.
3.
 - a) Если запись для BLOCK1 только что создана, установить в ней указатель на запись таблицы символов для NAME1, показывающий, что это имя является первым в данном блоке COMMON. Затем связать запись таблицы символов для NAME1 с записью таблицы символов для NAME2, используя поле таблицы символов, зарезервированное для связи членов одного и того же блока COMMON. И наконец, следует установить указатель в записи для BLOCK1 на запись таблицы символов для NAME2, определяя последний найденный член этого блока.
 - b) Если же это не первое объявление BLOCK1, просто связать NAME1 и NAME2 с концом списка имен для BLOCK1. Естественно, при этом должен быть обновлен содержащийся в записи указатель на конец списка для BLOCK1.

После обработки процедуры мы применяем алгоритм эквивалентности, вкратце рассматриваемый далее. Мы можем обнаружить, что блоку COMMON принадлежат некоторые дополнительные имена, поскольку они эквивалентны именам, имеющимся в COMMON. Нам не обязательно связывать такое имя XYZ с цепочкой для его блока COMMON. Достаточно установить бит в записи таблицы символов для XYZ, указывающий, что это имя определено как эквивалентное где-то в другом месте. Структура данных, которая будет рассмотрена далее, дает положение XYZ относительно некоторого имени, на самом деле объявленного как COMMON.

После выполнения операций эквивалентности мы можем создать карту памяти для каждого блока COMMON путем сканирования списка имен для этого блока. Инициализируем счетчик нулевым значением, и для каждого имени в списке делаем его смещение равным текущему значению счетчика, после чего добавляем к значению счетчика количество единиц памяти, занимаемой объектом, обозначенным данным именем. Записи блока COMMON после этого могут быть удалены, а занимаемая ими память использована при работе со следующей процедурой.

Если имя XYZ в COMMON эквивалентно имени вне блока COMMON, то мы должны определить максимальное смещение от начала XYZ для любого слова памяти, необходимой

каждому из эквивалентных имен. Например, если XYZ — действительное число, эквивалентное A(5, 5), где A — массив целых чисел размером 10×10, то A(1, 1) располагается за 44 слова до XYZ, а A(10, 10) — через 55 слов после XYZ, как показано на рис. 7.45. Существование A не влияет на счетчик блока COMMON; его значение будет только увеличено на одно слово при рассмотрении XYZ, независимо от того, чему эквивалентно это имя. Однако конец области данных для блока COMMON должен располагаться достаточно далеко, чтобы область данных вместила массив A. Таким образом, мы записываем максимальное смещение от начала блока COMMON для каждого из слов, используемых именами, эквивалентными членам данного блока. На рис. 7.45 эта величина должна быть как минимум равна смещению XYZ плюс 55. Кроме того, мы должны убедиться, что массив A не выходит за начало области данных, так что смещение XYZ от начала блока должно быть не менее 44. В противном случае мы получаем ошибку и должны вывести соответствующее диагностическое сообщение.

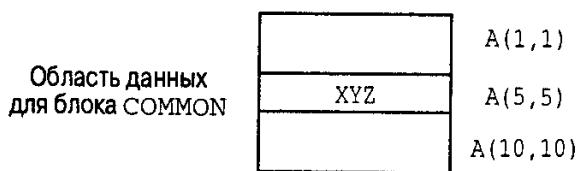


Рис. 7.45. Соотношение между инструкциями COMMON и EQUIVALENCE

Простой алгоритм эквивалентности

Первый алгоритм для обработки инструкций эквивалентности появился не в компиляторах, а в ассемблерах. Поскольку такие алгоритмы могут быть весьма сложными, в особенности при взаимодействии рассматриваемых инструкций COMMON и EQUIVALENCE, пока ограничимся типичной для ассемблера ситуацией, когда имеются только инструкции EQUIVALENCE вида

EQUIVALENCE A, B+*offset*

где A и B — имена ячеек памяти. Эта инструкция заставляет A обозначать память на расстоянии *offset* ячеек памяти от B.

Последовательность инструкций EQUIVALENCE группирует имена в *множества эквивалентности*, в которых положения одного имени по отношению к другому определяются инструкциями EQUIVALENCE. Например, последовательность инструкций

EQUIVALENCE A, B+100
EQUIVALENCE C, D-40
EQUIVALENCE A, C+30
EQUIVALENCE E, F

группирует имена в два множества {A, B, C, D} и {E, F}, где E и F означают одно и то же положение в памяти. С расположено через 70 ячеек после B, A — через 30 ячеек после C, а D — через 10 ячеек после A.

Чтобы вычислить множества эквивалентности, мы создаем для каждого из них дерево. Каждый узел дерева представляет имя и содержит смещение этого имени по отношению к имени в родительском узле. Имя в корне дерева мы будем называть *лидером* (leader). Положение любого имени по отношению к лидеру может быть вычислено при прохождении пути от узла для этого имени и суммирований встречающихся смещений.

Пример 7.12

Множество эквивалентности $\{A, B, C, D\}$, упомянутое выше, может быть представлено деревом, показанным на рис. 7.46. D является лидером, и можно вычислить, что A находится на 10 позиций ранее D, поскольку сумма смещений на пути от A к D равна $100 + (-110) = -10$. \square

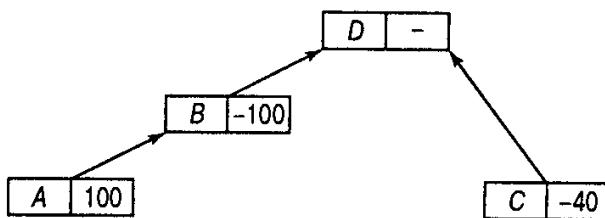


Рис. 7.46. Дерево, представляющее множество эквивалентности

Теперь мы приведем алгоритм построения деревьев для множеств эквивалентности. Подходящими полями записей в таблице символов являются следующие.

1. *Parent*, указывающее на запись в таблице символов для родительского узла либо имеющее нулевое значение, если имя представляется корнем (либо не связано отношением эквивалентности ни с каким другим именем).
2. *Offset*, определяющее смещение имени относительно имени в родительском узле.

Приведенный далее алгоритм предполагает, что лидером множества эквивалентности может быть любое имя. На практике в языке ассемблера реальное положение в памяти, определенное псевдооперацией, может иметь одно и только одно имя из множества, и именно это имя должно быть лидером. Мы считаем, что читатель может самостоятельно изменить алгоритм так, чтобы лидером было некоторое выделенное имя.

Алгоритм 7.1. Построение деревьев эквивалентности

Вход. Список инструкций определения эквивалентности вида

EQUIVALENCE A, B+dist

Выход. Множество деревьев, таких, что для любого упомянутого во входном списке эквивалентностей имени мы можем получить его положение в памяти относительно лидера, следуя по пути от этого имени к корню дерева и суммируя значения полей *offset*, встречающихся на этом пути.

Метод. Для каждой инструкции эквивалентности EQUIVALENCE A, B+dist выполняем действия, приведенные в листинге на рис. 7.47. Пояснение формулы в строке (12) для смещения лидера A относительно лидера B следующее. Положение A, скажем I_A , равно с плюс положение лидера A, скажем, m_A . Положение B, обозначенное I_B , равно d плюс положение m_B лидера B. Однако $I_A = I_B + dist$, так что $c + m_A = d + m_B + dist$. Следовательно, $m_A - m_B = d - c + dist$. \square

begin

- (1) Пусть p и q указывают соответственно на узлы A и B
- (2) $c := 0; d := 0; /* c и d представляют значения смещений A и B от лидеров в своих множествах */$
- (3) **while** parent(p) ≠ null **do begin**
- (4) $c := c + offset(p);$
- (5) $p := parent(p)$

```

end; /* Перемещаем p к лидеру a,
      по пути суммируя смещения */
(6) while parent(q) ≠ null do begin
(7)   d := d + offset(q);
(8)   q := parent(q)
end; /* Делаем то же для B */
(9) if p = q then /* A и B уже были обработаны */
(10)   if c - d ≠ dist then error;
        /* Для A и B заданы два разных
           относительных положения */
     else begin /* Объединяем множества A и B */
(11)       parent(p) := q /* Делаем лидера A
                           потомком лидера B */
(12)       offset(p) := d - c + dist
end
end

```

Рис. 7.47. Алгоритм эквивалентности

Пример 7.13

Если мы обработаем список

EQUIVALENCE A, B + 100
EQUIVALENCE C, D - 40

получим показанную на рис. 7.46 конфигурацию, но без смещения -100 в узле для B и без связи от B к D. При обработке

EQUIVALENCE A, C + 30

мы найдем, что p указывает на B после цикла `while` в строке (3), а q — на d после выполнения цикла `while` в строке (6). Кроме того, мы также имеем $c = 100$ и $d = -40$. Затем в строке (11) мы делаем D родителем по отношению к B и устанавливаем поле `offset` для B равным $(-40) - (100) + 30 = -110$. \square

Для обработки n инструкций эквивалентности алгоритм 7.1 может потребовать времени, пропорциональное n^2 , поскольку в наихудшем случае пути, проходимые в циклах в строках (3) и (6), могут включать все узлы соответствующих деревьев. Поскольку процесс обработки инструкций эквивалентности занимает лишь малую часть времени компиляции, пропорциональность n^2 не является слишком высокой ценой, так что более сложный по сравнению с представленным на рис. 7.47 алгоритм вряд ли нужен. Однако, как оказывается, мы можем заставить алгоритм 7.1 выполнять свою работу за время, практически линейно зависящее от количества обрабатываемых инструкций эквивалентности. Хотя маловероятно, чтобы множества эквивалентности были в среднем столь велики, чтобы сделать необходимыми эти изменения, отметим, что эквивалентность служит примером многих важных процессов, включающих “объединение множеств”. Например, многие эффективные алгоритмы анализа потока данных зависят от быстрых алгоритмов эквивалентности; заинтересовавшимся читателям рекомендуем обратиться к библиографическим примечаниям в главе 10, “Оптимизация кода”.

Первое, что мы можем сделать, — это для каждого лидера хранить количество узлов в его дереве. Тогда в строках (11) и (12) вместо произвольного привязывания лидера A к лидеру B привяжем лидера с меньшим количеством узлов к другому лидеру. Это гаран-

тирует, что деревья будут широкими и невысокими, а пути — короткими. Оставим читателю в качестве упражнения доказательство того, что и обработанных таким образом инструкций эквивалентности не могут привести к пути длиной более чем $\log_2 n$ узлов.

Вторая идея известна как сжатие пути. При проходе к корню в циклах в строках (3) и (6) мы делаем все встреченные узлы дочерними по отношению к лидеру (если они еще не являются таковыми), т.е. при проходе по пути мы записываем все встреченные узлы n_1, n_2, \dots, n_k , где n_1 — узел для А или В, а n_k — лидер. После этого мы обновляем смещения и делаем n_1, n_2, \dots, n_{k-2} дочерними по отношению к n_k , как показано на рис. 7.48.

```
begin
    h := offset(n_{k-1});
    for i := k-2 downto 1 do begin
        parent(n_i) := n_k;
        h := h + offset(n_i);
        offset(n_i) := h
    end
end
```

Рис. 7.48. Обновление смещений

Алгоритм эквивалентности для языка программирования Fortran

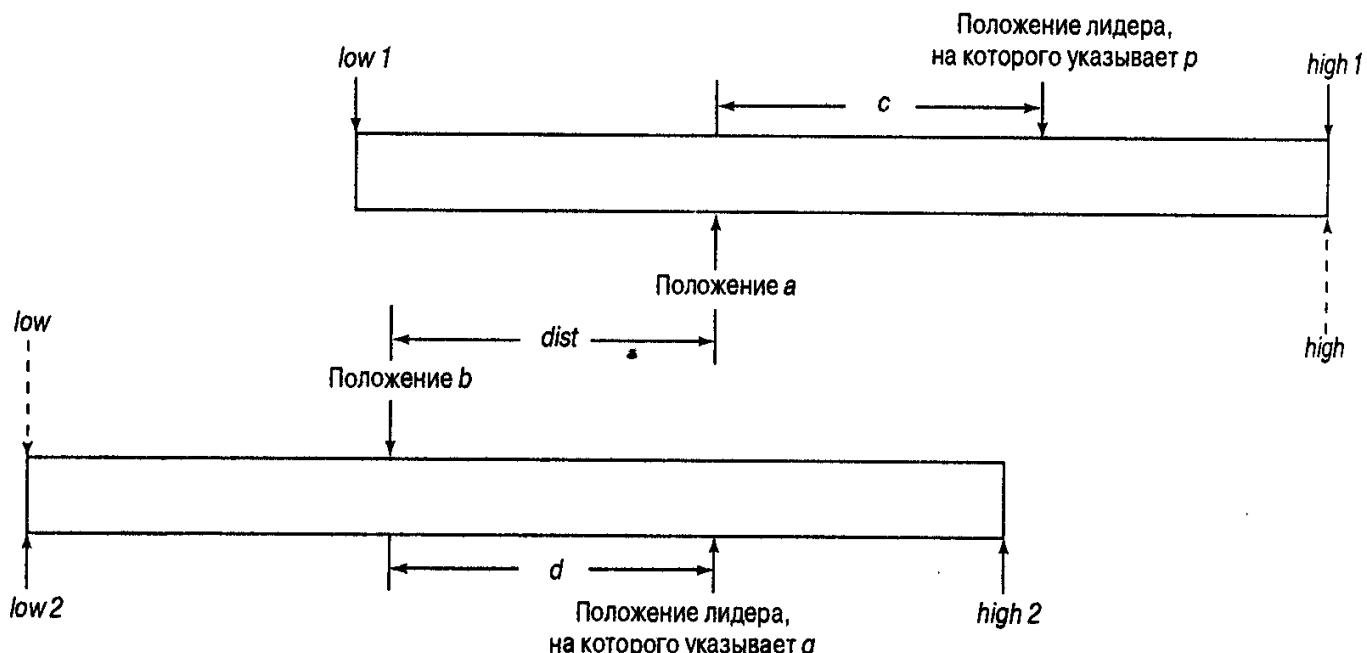
Для того чтобы алгоритм 7.1 работал в компиляторе Fortran, к нему следует добавить ряд дополнительных возможностей. Во-первых, мы должны определить, находится ли множество эквивалентности в блоке COMMON, что можно сделать путем записи для каждого лидера, находится ли имена из его множества в блоке COMMON, и если да, то в каком.

Во-вторых, в языке ассемблера один член множества эквивалентности привязывает все множество к реальным адресам, поскольку может рассматриваться как метка; таким образом, положение каждого имени из множества эквивалентности в реальной памяти определяется относительно этой метки. В случае Fortran положение в памяти определяется компилятором, так что множество эквивалентности вне COMMON может рассматриваться как “плавающее” до тех пор, пока компилятор не определит положение всего множества в соответствующей области данных. Для того чтобы корректно выполнить эту работу, компилятору требуется знать размер множества эквивалентности, т.е. объем памяти, занимаемой всей совокупностью имен множества. Для решения этой проблемы мы добавляем к лидеру два поля — *low* и *high*, указывающие смещение относительно лидера нижней и верхней границ памяти, используемой каким-либо членом множества. В-третьих, имеется небольшая проблема, связанная с тем, что имена могут быть массивами и элементы массива могут быть связаны отношением эквивалентности с элементами в других массивах.

Поскольку имеется три поля (*low*, *high* и указатель на блок COMMON), которые должны быть связаны с каждым лидером, нам не хотелось бы выделять память для них во всех записях таблицы символов. Одним из способов решения этой задачи является использование поля *parent* из алгоритма 7.1, которое в случае лидера указывает на запись в новой таблице с тремя полями — *low*, *high* и *comblk*. Поскольку эта таблица и таблица символов занимают непересекающиеся области, мы можем определить, на запись какой таблицы указывает указатель. Другой способ решения состоит в том, чтобы записи таблицы символов содержали бит, указывающий, является ли в настоящий момент лидером

интересующее нас имя. Если проблема памяти стоит очень остро, можно воспользоваться методом, позволяющим избежать использования дополнительной таблицы ценой чуть больших программистских усилий (такой метод обсуждается в упражнениях к данной главе).

Рассмотрим вычисления, которые должны заменить строки (11) и (12) на рис. 7.47. На рис. 7.49 a изображена ситуация, когда должны быть объединены два множества эквивалентности, лидеры которых определяются указателями p и q . Структура данных, представляющая эти множества, показана на рис. 7.49 b . Сначала мы должны убедиться, что среди двух множеств эквивалентности не имеется двух членов, находящихся в блоке COMMON. Даже если они оба находятся в одном блоке COMMON, стандарт Fortran запрещает объявление их эквивалентности. Если один из блоков COMMON содержит член одного из множеств эквивалентности, то объединенное множество имеет указатель на запись для этого блока в *comblk*. Код, выполняющий описанную проверку (полагая, что лидером объединенного множества становится лидер, на которого указывает указатель q), показан на рис. 7.50. Вместо строк (11) и (12) кода на рис. 7.47 мы должны также вычислить размер объединенного множества эквивалентности. Формулы для новых значений *low* и *high* относительно лидера, определяемого указателем q , приведены на рис. 7.49 a .



$$\begin{aligned} low &= \min(low_2, low_1 - c + dist + d) \\ high &= \max(high_2, high_1 - c + dist + d) \end{aligned}$$

a) Относительное расположение множеств эквивалентности



б) Структура данных

Рис. 7.49. Объединение множеств эквивалентности

```

begin
    comblk1 := comblk(parent(p));
    comblk2 := comblk(parent(q));
    if comblk1 ≠ null and comblk2 ≠ null then
        error; /* Эквивалентны два имени в COMMON */
    else if comblk2 = null then
        comblk(parent(q)) := comblk1
end

```

Рис. 7.50. Вычисление блоков COMMON

Таким образом, мы должны выполнить следующее.

```

begin
    low (parent(q)) := min(low (parent(q)),
                           low (parent(p)) - c + dist + d);
    high(parent(q)) := max(high(parent(q)),
                           high(parent(p)) - c + dist + d)
end

```

Эти инструкции следуют за строками (11) и (12) на рис. 7.47 при слиянии двух множеств эквивалентности.

Остались две детали, чтобы сделать алгоритм 7.1 пригодным для компилятора Fortran. В этом языке программирования мы можем объявить элемент из средины массива эквивалентным элементу другого массива или некоторому имени. Смещение массива A от его лидера означает смещение первой ячейки памяти, занятой A, от первой ячейки памяти лидера. Если, например, A(5, 7) эквивалентно B(20), то мы должны вычислить положение A(5, 7) относительно A(1, 1) и проинициализировать значение *c* в строке (2) на рис. 7.47 отрицательным значением этого смещения. Аналогично *d* должно быть проинициализировано отрицательным значением положения B(20) относительно B(1). Формул из раздела 8.3 вместе со знанием размера элементов массивов A и B достаточно для вычисления начальных значений *c* и *d*.

И наконец, последнее, что следует учесть, — это то, что Fortran позволяет инструкции EQUIVALENCE включать несколько переменных, например,

EQUIVALENCE (A(5, 7), B(20), C, D(4, 5, 6))

Такое объявление можно рассматривать как следующий набор объявлений.

EQUIVALENCE (B(20), A(5, 7))
EQUIVALENCE (C, A(5, 7))
EQUIVALENCE (D(4, 5, 6), A(5, 7))

Заметим, что если мы выполним обработку инструкций эквивалентности в указанном порядке, только A станет лидером множества более чем из одного элемента. Запись с полями *low*, *high*, *comblk* может многократно использоваться для “множеств эквивалентности” одного имени.

Отображение областей данных

Теперь мы можем описать правила, в соответствии с которыми память в различных областях данных назначается каждому имени подпрограммы.

1. Для каждого блока COMMON проходим по всем именам, объявленным находящимися в этом блоке в порядке их объявлений (используя цепочки имён COMMON, созданные

для этой цели в таблице символов). Выделяем необходимое для каждого имени количество слов памяти, используя при этом счетчик количества выделенных слов, так что для каждого имени может быть вычислено его смещение. Если имя A объявлено как эквивалентное другому, размер его множества эквивалентности не имеет значения, но мы должны убедиться, что значение *low* лидера A не выходит за начало блока COMMON. Обращаемся к значению *high* лидера с тем, чтобы установить нижнюю границу для последнего слова блока. Мы оставляем читателю вывод точных формул для этих вычислений.

2. Проходим все имена подпрограммы в любом порядке.
 - a) Если имя находится в блоке COMMON, не делаем ничего. Память для этого имени выделена в (1).
 - b) Если имя находится не в блоке COMMON и не объявлено как эквивалентное, выделяем для него необходимое количество памяти в области данных для подпрограммы.
 - c) Если имя A объявлено как эквивалентное, находим его лидера (скажем, L). Если L уже имеет определенную позицию в области данных для подпрограммы, вычисляем положение A, прибавляя к положению L значения всех полей *offset*, найденных на пути от A к L в дереве, представляющем множество эквивалентности A и L. Если L не имеет определенной позиции, выделяем для множества эквивалентности очередные *high-low* слов в области данных. Положение L среди этих слов — *-low*, а положение A, как и ранее, может быть вычислено суммированием значений полей *offset*.

Упражнения

- 7.1. Используя правила области видимости языка Pascal, определите объявления, применимые к каждому появлению имен a и b на рис. 7.51. Вывод программы состоит из целых чисел от 1 до 4.

```
program a(input, output);
procedure b(u,v,x,y: integer);
  var a: record a, b : integer end;
      b: record b, a : integer end;
begin
  with a do begin a := u; b := v end;
  with b do begin a := x; b := y end;
  writeln(a.a, a.b, b.a, b.b)
end;
begin
  b(1,2,3,4)
end.
```

Рис. 7.51. Pascal-программа с несколькими объявлениями a и b

- 7.2. Рассмотрим язык с блочной структурой, в котором имена могут быть объявлены как целые или действительные числа. Предположим, что выражения представлены термином *expr*, а инструкции языка ограничены присвоением, условными операторами, циклами *while* и последовательностями инструкций. Предполагая, что

целые числа занимают одно слово, а действительные — два, разработайте синтаксически управляемый алгоритм (основанный на пригодной для этого грамматике для объявлений и блоков) для определения связей имен со словами, которые могут использоваться активацией блока. Использует ли ваше распределение минимальное количество слов, достаточное при любом выполнении блока?

- *7.3. В разделе 7.4 мы утверждали, что дисплей может корректно поддерживаться, если каждая процедура глубины i сохраняет $d[i]$ в начале активации и восстанавливает $d[i]$ в конце. Индуктивно по числу вызовов докажите, что каждая процедура имеет дело с корректным дисплеем.
- 7.4. *Макрос* представляет собой определенный вид процедуры, реализуемый буквальной подстановкой его тела при каждом вызове. На рис. 7.52 показана программа на языке Pic и ее выход. Первые две строки определяют макросы *show* и *small*. Тела макросов располагаются между двумя знаками % в строке. Каждая из четырех изображенных на рисунке окружностей выводится макросом *show*; радиус окружности задается нелокальным именем *r*. Блоки в Pic ограничиваются квадратными скобками [и]. Каждая переменная с присвоением ей в блоке неявно объявляется в этом блоке. Что вы можете сказать об областях видимости каждого появления *r*, основываясь на выходе программы?

```
define show  % { circle radius r at Here } %
define small % [ r = 1/12; show ] %
[
    r = 1/6;
    show; small;
    move;
    show; small;
]
```



Рис. 7.52. Окружности, выведенные Pic-программой

- 7.5. Напишите процедуру для вставки элемента в связанный список, которой передается указатель на начало списка. При каком механизме передачи параметров эта процедура будет корректно работать?
- 7.6. Что выведет программа, представленная на рис. 7.53, при использовании следующего метода передачи параметров?
- По значению.
 - По ссылке.
 - Копирование-восстановление.
 - По имени.

```
program main(input, output);
procedure p(x, y, z);
begin
    y := y + 1;
    z := z + x;
end;
```

```

begin
    a := 2;
    b := 3;
    p(a+b, a, a);
    print a
end.

```

Рис. 7.53. Псевдопрограмма, иллюстрирующая методы передачи параметров

- 7.7. Когда процедура передается в качестве параметра в языке с лексической областью видимости, ее нелокальное окружение может быть передано с использованием связи доступа. Приведите алгоритм для определения этой связи.
- 7.8. На рис. 7.54 проиллюстрированы три типа окружения, которые могут быть связаны с процедурой, передаваемой в качестве параметра. *Лексическое (lexical), передаваемое (passing) и активационное (activation)* окружения такой процедуры состоят из связей идентификаторов в точке, где процедура соответственно определяется, передается в качестве параметра и активируется. Рассмотрим функцию *f*, передаваемую в качестве параметра в строке (11).

При использовании лексического, передаваемого и активационного окружений *f*, нелокальное *m* в строке (8) находится соответственно в области видимости объявления *m* в строках (6), (10) и (3).

- Изобразите дерево активации для данной программы.
- Каким будет выход программы при использовании лексического, передаваемого и активационного окружений *f*?
- * Модифицируйте реализацию дисплея для языка с лексической областью видимости, чтобы лексическое окружение корректно устанавливалось при активации процедуры, передаваемой в качестве параметра.

```

(1) program param(input, output);
(2)     procedure b(function h(n: integer) : integer);
(3)         var m: integer;
(4)         begin m := 3; writeln(h(2)) end { b };
(5)     procedure c;
(6)         var m: integer;
(7)             function f(n: integer) : integer;
(8)                 begin f := m + n end { f };
(9)             procedure r;
(10)                var m: integer;
(11)                    begin m := 7; b(f) end { r };
(12)                begin m := 0; r end { c };
(13)            begin
(14)                c
(15)            end.

```

Рис. 7.54. Пример лексического, передаваемого и активационного окружений

- *7.9. Инструкция *f := a* в строке (11) псевдопрограммы на рис. 7.55 вызывает функцию *a*, возвращающую в качестве результата функцию *addm*.
- Изобразите дерево активации выполнения данной программы.

- b) Предположим, что для нелокальных имен используется лексическая область видимости. Почему эта программа не будет корректно работать при выделении памяти в стеке?
- c) Каким будет выход программы при выделении памяти в куче?

```

(1)  program ret(input, output);
(2)  var f: function (integer): integer;
(3)      function a: function (integer): integer;
(4)          var m: integer;
(5)          function addm(n: integer): integer;
(6)              begin return m + n end;
(7)              begin m := 0; return addm end;
(8)      procedure b(g: function(integer): integer);
(9)          begin writeln(g(2)) end;
(10)     begin
(11)         f := a; b(f)
(12)     end.

```

Рис. 7.55. Псевдопрограмма, в которой функция addm возвращается в качестве результата

- *7.10. Некоторые языки, наподобие Lisp, могут возвращать вновь созданную в процессе работы программы процедуру. На рис. 7.56 все функции, как определенные в тексте, так и созданные в процессе работы, получают не более одного аргумента и возвращают одно значение — либо функцию, либо действительное число. Оператор \circ обозначает композицию функций, т.е. $(f \circ g)(x) = f(g(x))$.

- a) Какое значение будет выведено функцией *main*?
- *b) Предположим, что когда создается и возвращается процедура *p*, ее запись активации становится дочерней по отношению к записи активации функции, возвращающей *p*. Передача окружения *p* может в таком случае поддерживаться с помощью дерева записей активации, а не стека. Что из себя представляет дерево записей активации, когда *main* на рис. 7.56 вычисляет *a*?
- *c) Предположим, что запись активации для *p* создается, когда *p* активируется, и становится потомком записи активации для процедуры, вызывающей *p*. Такой подход может использоваться для поддержки среды активации *p*. Изобразите снимки записей активации и их родительско-дочерних отношений при выполнении инструкций из *main*. Достаточно ли стека для хранения записей активации при использовании такого подхода?

```

function f(x: function);
var y: function;
    y := x ∘ h; /* Создание у при выполнении */
    return y
end { f };
function h();
    return sin
end { h };
function g(z: function);
var w: function;
    w := arctan ∘ z; /* Создание w при выполнении */

```

```

    return w
end { g };
function main();
var a: real;
    u, v: function;
    v := f(g);
    u := v();
    a := u(π/2);
    print a
end { main }.

```

Рис. 7.56. Псевдопрограмма создания функций в процессе работы

- 7.11. Один из способов удаления из хеш-таблиц имен, область видимости которых завершилась (как описано в разделе 7.6), состоит в том, чтобы оставить ненужные записи в списке до очередного сканирования списка. Считая, что записи включают имя процедуры, в которой было сделано объявление, мы в принципе в состоянии определить, является ли запись устаревшей, и удалить ее, если это так. Разработайте схему индексации для процедур, которая позволит нам за время $O(1)$ определить, является ли процедура “устаревшей”, т.е. завершилась ли ее область видимости.
- 7.12. Многие хеш-функции могут быть охарактеризованы последовательностью целых констант $\alpha_0, \alpha_1, \dots$. Если c_i , $1 \leq i \leq n$, представляет собой целое значение i -го символа в строке s , то строка хешируется в

$$\text{hash}(s) = (\alpha_0 + \sum_{i=1}^n \alpha_i c_i) \bmod m$$

где m — размер хеш-таблицы. В каждом из описанных далее случаев по строке s определяется некоторое целое число; хеш-значение получается из него путем деления этого числа по модулю m . Определите соответствующие каждому случаю константы $\alpha_0, \alpha_1, \dots$ (или покажите, что искомой последовательности не существует).

- a) Получить сумму символов.
- b) Получить сумму первого и последнего символов.
- c) Получить значение h_n , где $h_0 = 0$ и $h_i = 2h_{i-1} + c_i$.
- d) Рассмотреть биты четырех символов в середине строки как 32-битовое целое число.
- e) 32-битовое целое число можно рассматривать, как состоящее из 4 байтов, где каждый байт представляет собой разряд, способный принимать 256 возможных значений. Начиная с 0000, для $1 \leq i \leq n$ добавляем c_i к байту с номером $i \bmod 4$ (при необходимости — с переносом в следующий разряд). Таким образом, c_1 и c_5 добавляются к байту 1, c_2 и c_6 — к байту 2 и т.д. Получить окончательное значение.

- *7.13. Почему хеш-функции, характеризуемые последовательностями целых чисел $\alpha_0, \alpha_1, \dots$ (см. упр. 7.12), иногда плохо работают для входного потока, состоящего из последовательных строк, например v000, v001, ...? Признак плохой работы заключается в том, что их поведение отклоняется от случайного и может быть предсказано.

- *7.14. Когда n строк хешируются в m списков, среднее количество строк на список составляет n/m безотносительно к неравномерности распределения строк. Предположим, что d представляет собой “распределение”, т.е. случайная строка помещается в i -й список с вероятностью $d(i)$. Предположим, что хеш-функция с распределением d разместила b_j случайным образом выбранных строк в список j ($0 \leq j \leq m-1$). Покажите, что ожидаемое значение $W = \sum_{j=0}^{m-1} (b_j)(b_j + 1)/2$ линейно связано с дисперсией распределения d . Покажите, что для равномерного распределения ожидаемым значением W будет $(n/2m)(n+2m-1)$.

- 7.15. Предположим, что в программе на языке Fortran есть следующая последовательность объявлений.

```
SUBROUTINE SUB(X, Y)
INTEGER A, B(20), C(10,15), D, E
COMPLEX F, G
COMMON /CBLK/ D, E
EQUIVALENCE (G, B(2))
EQUIVALENCE (D, F, B(1))
```

Покажите содержимое областей данных для SUB и CBLK (по крайней мере часть области CBLK, доступную из SUB). Почему в них нет места для X и Y?

- *7.16. При вычислении эквивалентности одной из полезных структур данных является *кольцевая структура* (ring structure). В каждой записи таблицы символов для связи членов множества эквивалентности используется один указатель и поле смещения. Такая структура представлена на рис. 7.57, где эквивалентны A, B, C и D, а также E и F, причем B расположено после A на расстоянии 20 слов и т.д.

- Разработайте алгоритм для вычисления смещения X относительно Y, полагая, что X и Y — члены одного множества эквивалентности.
- Разработайте алгоритм вычисления *low* и *high*, определенных в разделе 7.9, относительно положения некоторого имени Z.
- Разработайте алгоритм для обработки
EQUIVALENCE U, V

При этом U и V не обязательно находятся в разных множествах эквивалентности.

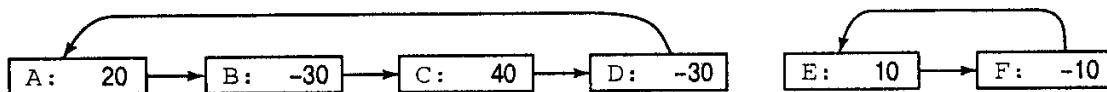


Рис. 7.57. Кольцевая структура

- *7.17. Алгоритм отображения областей данных, приведенный в разделе 7.9, требует проверки, что значение *low* лидера множества эквивалентности A не вызывает выхода пространства для множества эквивалентности A за начало блока COMMON и что при необходимости для увеличения верхней границы блока COMMON мы вычисляем значение *high* лидера A. Разработайте соответствующие формулы, использующие смещение A в блоке COMMON *next* и текущее последнее слово блока *last*, для проверки и обновления (при необходимости) значения *last*.

Библиографические примечания

В реализации рекурсивных функций стеки играли существенную роль. В [305], посвященной истории Lisp, можно найти воспоминания о том, как в 1958 году при реализации рекурсивных подпрограмм было принято решение об использовании непрерывного общедоступного стекового массива для хранения значений переменных и адресов возврата из подпрограмм. Включение блоков и использование рекурсивных процедур в Algol 60 [330] также стимулировали разработку стекового распределения памяти. Идея использования дисплея для доступа к нелокальным именам в языке с лексическими областями видимости разработана Дейкстрой [113, 114]. Хотя Lisp и использует динамические области видимости, можно добиться эффекта лексических областей видимости, используя специальные аргументы, состоящие из функции и связи доступа; в [305] описана разработка такого механизма. Наследники Lisp, такие как Common Lisp [411], отошли от идеи динамических областей видимости.

Связывание имен рассматривается в книгах, посвященных языкам программирования [2, 352, 430]. Альтернативный подход, предложенный в главе 2, “Простой однопроходный компилятор”, состоит в чтении описания компилятора. В [247] описана пошаговая разработка, которая начинается с калькулятора для арифметических выражений и приводит к построению интерпретатора простого языка с рекурсивными процедурами. Можно также ознакомиться с кодом для Pascal-S в [457]. Детальное описание стекового распределения памяти, использование дисплеев и динамическое распределение массивов имеется в [363].

В [220] обсуждается разработка вызывающей последовательности, позволяющая использовать переменное количество аргументов процедуры. Общий метод установки глобального дисплея состоит в следовании по цепочке связей доступа с попутной установкой элементов дисплея. Подход из раздела 7.4, касающийся только одного элемента, описан в [377]. В [324] обсуждаются отличия между средами, применяемыми при передаче функции в качестве параметра, и рассматриваются проблемы, возникающие при реализации таких сред с использованием мелкого и глубокого доступов. Стековое распределение памяти не может использоваться для языков с сопрограммами или множественными процессами. В [276] рассматриваются быстрые реализации с использованием распределения памяти в куче.

В математической логике связанные с ограниченной областью видимости и подстановка появились в работе [145]. Подстановка и передача параметров были темой многочисленных дискуссий как в математической логике, так и в области языков программирования. В [83] отмечается сложность корректной формулировки правила подстановки функциональных переменных. Лямбда-исчисление [82] было применено к окружениям в языках программирования, например, в [277]. В соответствии с этой работой пары, состоящие из функции и связи доступа, часто именуются *замыканиями* (closure).

Структуры данных и алгоритмы для представления таблиц символов и поиска в них детально рассматриваются в [7, 8, 258]. Глубокое рассмотрение вопросов хеширования можно найти в [258, 323]; исходная статья о хешировании представлена в [348]. Дополнительную информацию о технологиях организации таблиц символов можно найти в [310]. Пример 7.10 взят из [56]. Генератор таблиц символов описан в [366].

Алгоритмы эквивалентности описаны в [36, 151]; мы заимствовали подход из [151]. Эффективность алгоритмов эквивалентности рассматривается в [132, 193, 426].

ГЛАВА 8

Генерация промежуточного кода

На начальной стадии компиляции исходная программа транслируется в промежуточное представление, из которого на заключительной стадии генерируется целевой код. Детали целевого языка по возможности сосредоточены на заключительной стадии компилятора. Хотя исходная программа и может транслироваться непосредственно в целевой язык, использование промежуточного, не зависящего от конкретной машины представления имеет определенные преимущества.

1. Облегчается перенос на другую целевую машину; компилятор для другой целевой машины может быть создан подключением заключительной стадии для новой машины к имеющейся начальной.
2. К промежуточному представлению можно применить машинно-независимый оптимизатор кода (такие оптимизаторы подробно рассматриваются в главе 10, “Оптимизация кода”).

В этой главе показано, каким образом можно использовать синтаксически управляемые методы из глав 2, “Простой однопроходный компилятор”, и 5, “Синтаксически управляемая трансляция”, для трансляции в промежуточное представление таких конструкций языка программирования, как объявления, присвоения или инструкции управления потоком. Для простоты мы считаем, что исходная программа уже разобрана и все статические проверки выполнены, как показано на рис. 8.1. Большинство синтаксически управляемых определений в этой главе может быть реализовано как в процессе восходящего, так и нисходящего разбора с использованием технологий, описанных в главе 5, “Синтаксически управляемая трансляция”; так что генерация промежуточного кода при желании может быть включена в синтаксический анализ.



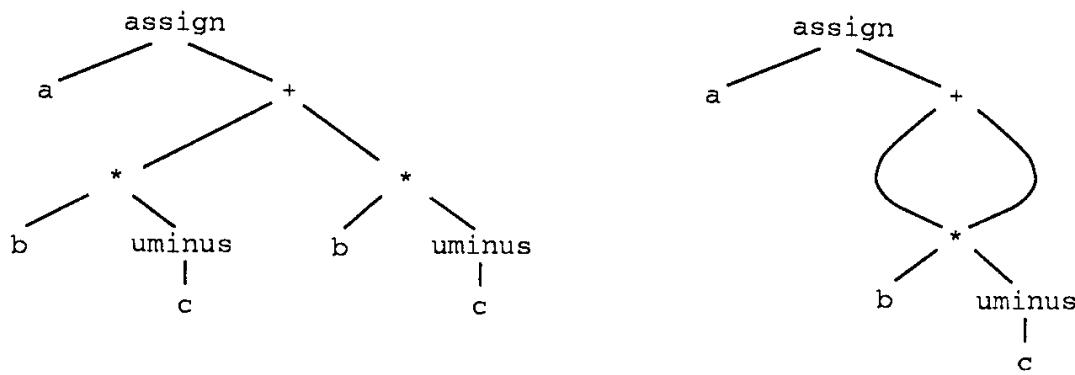
Рис. 8.1. Положение генератора промежуточного кода

8.1. Языки промежуточных представлений

Синтаксические деревья и постфиксная запись, введенные в разделах 5.2 и 2.3, представляют собой два типа промежуточного представления. В этой главе будет использоваться третий тип — трехадресный код. Семантические правила генерации трехадресного кода для основных конструкций языков программирования схожи с правилами построения синтаксических деревьев или генерации постфиксной записи.

Графическое представление

Синтаксическое дерево изображает естественную иерархическую структуру исходной программы. Даг дает ту же информацию, но в более компактном виде (одинаковые подвыражения в нем объединены). На рис. 8.2 приведены синтаксическое дерево и даг для инструкции присвоения $a := b^*-c+b^*-c$.



а) Синтаксическое дерево

б) Даг

Рис. 8.2. Графическое представление присвоения $a := b^*-c+b^*-c$

Постфиксная запись представляет собой линеаризованное представление синтаксического дерева; такая запись — не что иное, как список узлов дерева, в котором узлы располагаются сразу после своих дочерних узлов. Постфиксная запись для синтаксического дерева (рис. 8.2а) представляет собой

$a\ b\ c\ uminus\ *\ b\ c\ uminus\ *\ +\ assign \quad (8.1)$

Дуги синтаксического дерева явным образом в постфиксной записи не участвуют, но могут быть восстановлены в соответствии с порядком, в котором узлы появляются в постфиксной записи, и числом operandов оператора в узле. Восстановление дуг подобно вычислениям постфиксной записи с использованием стека. Более подробно об этом и связи постфиксной записи с кодом стековой машины рассказывается в разделе 2.8.

Синтаксические деревья для инструкций присвоения строятся с помощью синтаксически управляемого определения (рис. 8.3), представляющего собой расширение определения из раздела 5.2. Нетерминал S порождает инструкцию присвоения. Два бинарных оператора $+$ и $*$ представляют собой примеры полного множества операторов типичного языка программирования. Ассоциативность и приоритет операторов — обычные, несмотря на то что они не внесены в грамматику. Приведенное определение строит дерево на рис. 8.2а для входного потока $a := b^*-c+b^*-c$.

ПРОДУКЦИЯ	СЕМАНТИЧЕСКОЕ ПРАВИЛО
$S \rightarrow id := E$	$S.nptr := mknnode('assign', mkleaf(id, id.place), E.nptr)$
$E \rightarrow E_1 + E_2$	$E.nptr := mknnode('+', E_1.nptr, E_2.nptr)$
$E \rightarrow E_1 * E_2$	$E.nptr := mknnode('*', E_1.nptr, E_2.nptr)$
$E \rightarrow -E_1$	$E.nptr := mknnode('uminus', E_1.nptr)$
$E \rightarrow (E_1)$	$E.nptr := E_1.nptr$
$E \rightarrow id$	$E.nptr := mkleaf(id, id.place)$

Рис. 8.3. Синтаксически управляемое определение, позволяющее построить синтаксические деревья для инструкций присвоения

Это же синтаксически управляемое определение позволит построить даг, показанный на рис. 8.2б, если функции *mkunode(op, child)* и *mknnode(op, left, right)* будут, если возможно, возвращать указатель на уже существующий узел, а не строить новый. Токен *id* имеет атрибут *place*, указывающий на запись в таблице символов для данного идентификатора. В разделе 8.3 мы покажем, каким образом может быть найдена запись в таблице символов по атрибуту *id.name*, представляющему связанный с данным появлением *id* лексему. Если лексический анализатор хранит все лексемы в едином массиве символов, атрибут *name* может представлять собой индекс первого символа лексемы.

На рис. 8.4 показаны два представления синтаксического дерева, приведенного на рис. 8.2а. Каждый узел представлен записью с полем для оператора и дополнительными полями для указателей на дочерние узлы. На рис. 8.4б узлы выделяются из массива записей, и индекс (или позиция) узла служит указателем на него. Все узлы синтаксического дерева можно обойти, следуя указателям и начав с корня в позиции 10.

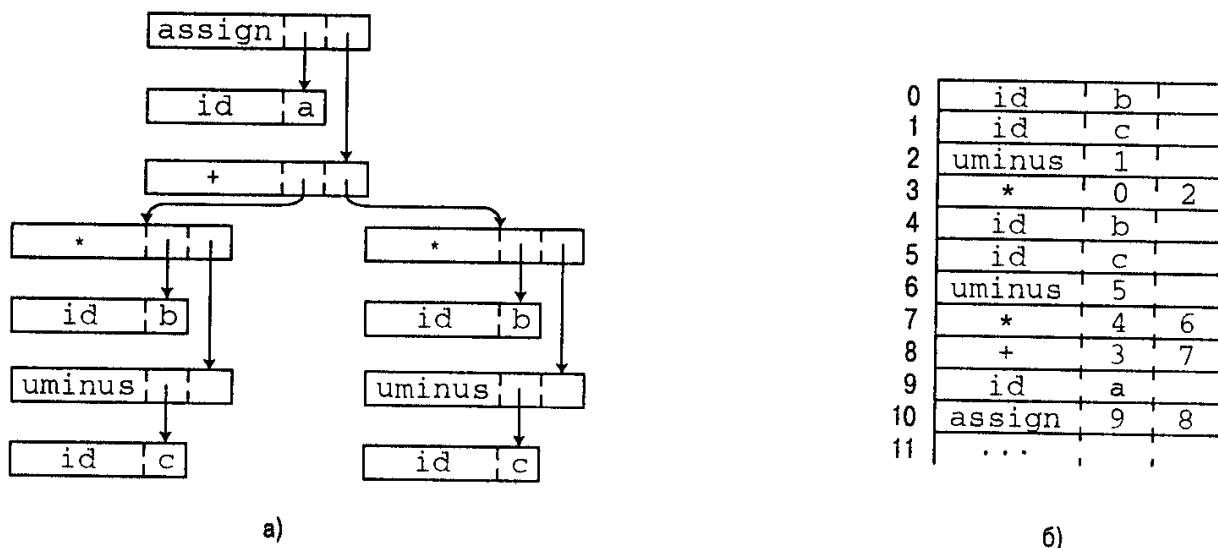


Рис. 8.4. Два представления синтаксического дерева на рис. 8.2а

Трехадресный код

Трехадресный код представляет собой последовательность инструкций вида
 $x := y \ op \ z$

где x , y и z — имена, константы или временные переменные, генерируемые компилятором; op означает некоторый оператор, например арифметический оператор для работы с числами с фиксированной или плавающей точкой или оператор для работы с логическими значениями. Заметим, что не разрешены никакие встроенные арифметические выражения, и в правой части инструкции имеется только один оператор. Следовательно, выражение исходного языка наподобие $x+y^*z$ может быть транслировано в следующую последовательность.

```
t1 := y * z
t2 := x + t1
```

Здесь t_1 и t_2 — сгенерированные компилятором временные имена. Такое разложение сложных арифметических выражений и вложенных инструкций потока управления делает трехадресный код подходящим для генерации целевого кода и оптимизации (см. главы 10, “Оптимизация кода” и 12, “Некоторые компиляторы”). Использование имен для

вычисленных программой промежуточных значений обеспечивает трехадресному коду, в отличие от постфиксной записи, возможность легкого переупорядочения.

Трехадресный код является линеаризованным представлением синтаксического дерева или дага, в котором внутренним узлам графа соответствуют явные имена. Синтаксическое дерево и даг, показанные на рис. 8.2, представлены последовательностью трехадресных кодов на рис. 8.5. Имена переменных могут непосредственно участвовать в трехадресных инструкциях, так что на рис. 8.5а нет инструкций, соответствующих листьям на рис. 8.4.

```
t1 := - c  
t2 := b * t1  
t3 := - c  
t4 := b * t3  
t5 := t2 + t4  
a := t5
```

а) Код для синтаксического дерева

```
t1 := - c  
t2 := b * t1  
t5 := t2 + t2  
a := t5
```

б) Код для дага

Рис. 8.5. Трехадресный код, соответствующий дереву и дагу на рис. 8.2

Причина использования термина “трехадресный код” кроется в том, что каждая инструкция обычно содержит три адреса — два для операндов и один для результата. В реализации трехадресного кода, приведенной далее в этой главе, определенные в программе имена заменяются указателями на записи в таблице символов для этих имен.

Типы трехадресных инструкций

Трехадресные инструкции похожи на ассемблерный код. Инструкции могут иметь символьные метки; имеются инструкции работы с потоком управления. Символьная метка представляет индекс трехадресной инструкции в массиве, содержащем промежуточный код. Замена меток индексами может быть выполнена в процессе отдельного прохода либо с использованием технологии обратных поправок, обсуждаемой в разделе 8.6.

Ниже приведен список основных трехадресных инструкций, используемых в оставшейся части книги.

1. Инструкции присвоения вида $x := y \text{ op } z$, где op — бинарная арифметическая или логическая операция.
2. Инструкция присвоения вида $x := \text{op } y$, где op — унарная операция. Основные унарные операции включают унарный минус, логическое отрицание, операторы сдвига и операторы преобразования, которые, например, преобразуют число с фиксированной точкой в число с плавающей точкой.
3. Инструкции копирования вида $x := y$, в которых значение y присваивается x .
4. Безусловный переход `goto L`. После этой инструкции будет выполнена трехадресная инструкция с меткой L .
5. Условный переход типа `if x relop y goto L`. Эта инструкция применяет оператор отношения $\text{relop} (<, \geq \text{ и т.п.})$ к x и y , и следующей выполняется инструкция с меткой L , если соотношение $x \text{ relop } y$ верно. В противном случае выполняется следующая за условным переходом инструкция.

6. Инструкции `param x` и `call p, n` для вызова процедур и `return y` для возврата из них, где `y` обозначает необязательное возвращаемое значение. Обычно они используются в виде следующей последовательности трехадресных инструкций.

```
param x1
param x2
...
param xn
call p, n
```

Данная последовательность генерируется в качестве части вызова процедуры `p(x1, x2, ..., xn)`. В инструкции `call p, n` целое число `n`, указывающее количество действительных параметров, не является излишним в силу того, что вызовы могут быть вложенными. Реализация вызовов процедур описана в разделе 8.7.

7. Индексированные присвоения типа `x := y[i]` и `x[i] := y`. Первая инструкция присваивает `x` значение, находящееся в `i`-й ячейке памяти по отношению к `y`. Инструкция `x[i] := y` заносит в `i`-ю ячейку памяти по отношению к `x` значение `y`. В обеих инструкциях `x`, `y` и `i` ссылаются на объекты данных.
8. Присвоение адресов и указателей вида `x := &y`, `x := *y` и `*x := y`. Первая инструкция устанавливает значение `x` равным расположению `y` в памяти. Предположительно, `y` представляет собой имя, возможно временное, обозначающее выражение с *l*-значением типа `A[i, j]`, а `x` — имя указателя или временное имя. Таким образом, *r*-значение `x` представляет собой *l*-значение некоторого объекта. Во второй инструкции под `y` подразумевается указатель или временная переменная, *r*-значение которой представляет собой местоположение ячейки памяти. В результате *r*-значение `x` становится равным содержимому этой ячейки. И наконец, инструкция `*x := y` устанавливает *r*-значение объекта, указываемого `x`, равным *r*-значению `y`.

Выбор приемлемых операторов представляет собой важный вопрос в создании промежуточного представления. Очевидно, что множество операторов должно быть достаточно богатым, чтобы позволить реализовать все операции исходного языка. Небольшое множество операторов легче реализуется на новой целевой машине, однако ограниченное множество инструкций может привести к генерации длинных последовательностей инструкций промежуточного представления для некоторых конструкций исходного языка и добавить работы оптимизатору и генератору целевого кода.

Синтаксически управляемая трансляция в трехадресный код

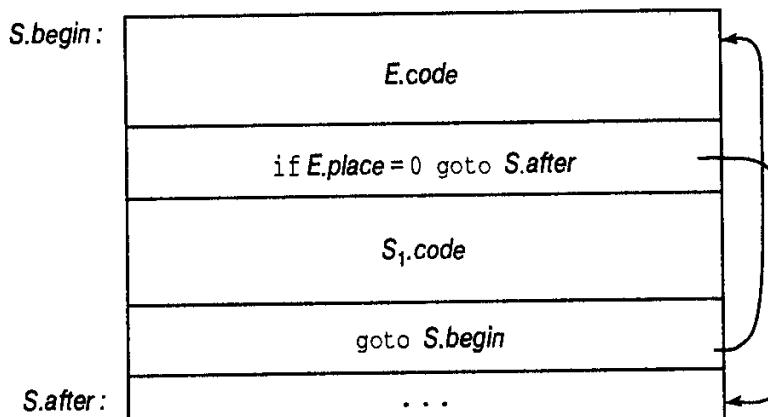
При генерации трехадресного кода для внутренних узлов синтаксического дерева создаются временные имена. Значение нетерминала `E` в левой части $E \rightarrow E_1 + E_2$ будет сохранено в новой временной переменной `t`. В целом трехадресный код для `id := E` состоит из кода для вычисления значения `E` в некоторой временной переменной `t`, за которым следует присвоение `id.place := t`. Если выражение представляет собой отдельный идентификатор, например `y`, то этот идентификатор сам по себе хранит значение выражения. В настоящий момент мы создаем новое временное имя всякий раз, когда необходима временная переменная; технология повторного использования временных переменных изложена в разделе 8.3.

S-атрибутное определение (рис. 8.6) генерирует трехадресный код для инструкций присвоения. Для входной строки `a := b* -c + b* -c` это определение дает код, приведенный на рис. 8.5а. Синтезируемый атрибут `S.code` представляет трехадресный код для присвоения `S`. Нетерминал `E` имеет два атрибута.

- . $E.place$ — имя, которое хранит значение E .
 - . $E.code$ — последовательность трехадресных инструкций, вычисляющих E .
- Функция $newtemp$ последовательно возвращает различные имена временных переменных t_1, t_2, \dots .

ПРОДУКЦИЯ	СЕМАНТИЧЕСКИЕ ПРАВИЛА
$S \rightarrow id := E$	$S.code := E.code \parallel gen(id.place := E.place)$
$E \rightarrow E_1 + E_2$	$E.place := newtemp;$ $E.code := E_1.code \parallel E_2.code \parallel$ $gen(E.place := E_1.place + E_2.place)$
$E \rightarrow E_1 * E_2$	$E.place := newtemp;$ $E.code := E_1.code \parallel E_2.code \parallel$ $gen(E.place := E_1.place * E_2.place)$
$E \rightarrow -E_1$	$E.place := newtemp;$ $E.code := E_1.code \parallel gen(E.place := 'uminus' E_1.place)$
$E \rightarrow (E_1)$	$E.place := E_1.place;$ $E.code := E_1.code$
$E \rightarrow id$	$E.place := id.place;$ $E.code := "$

Рис. 8.6. Синтаксически управляемое определение, позволяющее генерировать трехадресный код для присвоений



ПРОДУКЦИЯ	СЕМАНТИЧЕСКИЕ ПРАВИЛА
$S \rightarrow \text{while } E \text{ do } S_1$	$S.begin := newlabel;$ $S.after := newlabel;$ $S.code := gen(S.begin':') \parallel$ $E.code \parallel$ $gen('if' E.place ='0' 'goto' S.after) \parallel$ $S_1.code \parallel$ $gen('goto' S.begin) \parallel$ $gen(S.after':')$

Рис. 8.7. Семантические правила, генерирующие код для цикла `while`

Для удобства на рис. 8.6 мы используем запись $gen(x := y + z)$ для представления трехадресной инструкции $x := y + z$. Выражения, используемые вместо переменных x, y

и z , вычисляются при передаче их в gen , а операторы в кавычках, например '+', передаются буквально. На практике трехадресные инструкции вместо хранения в атрибутах $code$ могут тут же записываться в выходной файл.

Инструкции потока управления могут быть добавлены к языку присвоений (рис. 8.6) с помощью продукции и семантических правил, наподобие представленных на рис. 8.7 для цикла **while**. Код для $S \rightarrow \text{while } E \text{ do } S$, генерируется с использованием новых атрибутов $S.begin$ и $S.after$ для пометки первой инструкции в коде для E и инструкции, следующей за кодом S , соответственно. Эти атрибуты представляют собой метки, созданные функцией *newlabel*; при каждом вызове возвращающей новую метку. Заметим, что $S.after$ помечает инструкцию, которая располагается после кода цикла **while**. При генерации кода предполагается, что ненулевое выражение представляет значение **true**; так что когда значение E становится равным нулю, управление покидает цикл **while**.

Выражения, которые руководят потоком управления, в целом могут представлять собой логические выражения, содержащие операторы отношения и логические операторы. Семантические правила для циклов **while** в разделе 8.6 отличаются от семантических правил, приведенных на рис. 8.7, тем, что допускают поток управления в логическом выражении.

Постфиксная запись может быть получена путем адаптации семантических правил, представленных на рис. 8.6 (см. также рис. 2.5). Постфиксная запись для идентификатора представляет собой сам идентификатор. Правила для других продукции дописывают оператор к коду для operandов. Например, связанное с продукцией правило $E \rightarrow -E_1$ гласит

$E.code := E_1.code \parallel 'uminus'$

Вообще говоря, промежуточное представление, генерируемое в этой главе синтаксически управляемой трансляцией, может быть изменено путем применения подобных модификаций к семантическим правилам.

Реализация трехадресных инструкций

Трехадресные инструкции представляют собой абстрактный вид промежуточного кода. В компиляторе эти инструкции могут быть реализованы как записи с полями для операторов и operandов. Три возможных представления — это четверки, тройки и косвенные тройки.

Четверки

Четверка (quadruple) представляет собой запись с четырьмя полями, которые мы назовем op , $arg1$, $arg2$ и $result$. Поле op содержит внутренний код оператора. Трехадресная инструкция $x := y \circ z$ представляется размещением y в $arg1$, z — в $arg2$ и x — в $result$. Инструкции с унарным оператором наподобие $x := -y$ или $x := y$ не используют $arg2$. Операторы типа $param$ не используют ни $arg2$, ни $result$. Условные и безусловные переходы помещают в $result$ целевую метку. Четверки на рис. 8.8a представляют присвоение $a := b^* - c + b^* - c$ и получены из трехадресного кода на рис. 8.5a.

Содержимое полей $arg1$, $arg2$ и $result$ обычно представляет собой указатели на записи в таблице символов для имен, представленных этими полями. В этом случае временные имена должны быть внесены в таблицу символов при их создании.

	<i>op</i>	<i>arg1</i>	<i>arg2</i>	<i>result</i>
(0)	uminus	c		t_1
(1)	*	b	t_1	t_2
(2)	uminus	c		t_3
(3)	*	b	t_3	t_4
(4)	+	t_2	t_4	t_5
(5)	$:=$	t_5		a

а) Четверки

	<i>op</i>	<i>arg1</i>	<i>arg2</i>
(0)	uminus	c	
(1)	*	b	(0)
(2)	uminus	c	
(3)	*	b	(2)
(4)	+	(1)	(3)
(5)	assign	a	(4)

б) Тройки

Рис. 8.8. Представление трехадресных инструкций четверками и тройками

Тройки

Для того чтобы избежать вставки временных имен в таблицу символов, мы можем ссылаться на временные значения по номеру инструкции, которая вычисляет значение, соответствующее этому имени. Если мы поступим таким образом, трехадресные инструкции можно будет представить записями только с тремя полями: *op*, *arg1* и *arg2*, как показано на рис. 8.8б. Поля *arg1* и *arg2* для аргументов *op* представляют собой либо указатели в таблицу символов (для определенных программистом имен или констант), либо указатели на тройки (для временных значений). Поскольку здесь используется три поля, этот вид промежуточного кода известен как тройки (triple)¹. За исключением представления имен, определенных в программе, тройки соответствуют представлению синтаксического дерева или графа в виде массива вершин (рис. 8.4).

В таблице числа в скобках представляют указатели на тройки; указатели на таблицу символов представлены соответствующими именами. На практике информация, необходимая для интерпретации различных типов записей в полях *arg1* и *arg2*, может быть закодирована в поле *op* или дополнительных полях. Тройки на рис. 8.8б соответствуют четверкам на рис. 8.8а. Заметим, что инструкция копирования $a := t_5$ закодирована тройкой, в которой a размещено в поле *arg1* и использован оператор *assign*.

Тернарные операции типа $x[i] := y$ требуют двух троек, как показано на рис. 8.9а (присвоение $x := y[i]$ показано на рис. 8.9б).

	<i>op</i>	<i>arg1</i>	<i>arg2</i>
(0)	$[] =$	x	i
(1)	assign	(0)	y

а) $x[i] := y$

	<i>op</i>	<i>arg1</i>	<i>arg2</i>
(0)	$=[]$	y	i
(1)	assign	x	(0)

б) $x := y[i]$

Рис. 8.9. Тройки для операций с массивами

Косвенные тройки

Еще одно представление трехадресного кода состоит в использовании списка указателей на тройки вместо списка самих троек. Естественно, что такая реализация названа косвенными тройками (indirect triples).

¹ Иногда о тройках говорят как о “двуадресном коде”, а о четверках — как о “трехадресном коде”. Мы, тем не менее, трактуем трехадресный код как абстрактную запись с различными реализациями, каковыми являются тройки и четверки.

В качестве примера воспользуемся массивом *statement* для перечисления указателей на тройки в требуемом порядке. Тройки на рис. 8.8б могут быть представлены, как на рис. 8.10.

	<i>statement</i>		<i>op</i>	<i>arg1</i>	<i>arg2</i>
(0)	(14)	(14)	uminus	c	
(1)	(15)	(15)	*	b	(14)
(2)	(16)	(16)	uminus	c	
(3)	(17)	(17)	*	b	(16)
(4)	(18)	(18)	+	(15)	(17)
(5)	(19)	(19)	assign	a	(18)

Рис. 8.10. Косвенные тройки, представляющие трехадресные инструкции

Сравнение представлений: использование косвенного обращения

Разницу между тройками и четверками можно рассматривать как вопрос о наличии в представлении определенной степени косвенности. Когда мы окончательно генерируем целевой код, каждое имя — временное или определенное программистом — получает некоторый адрес в памяти. Этот адрес помещается в запись таблицы символов для данного имени. При использовании четверок трехадресные инструкции, определяющие или использующие временные переменные, могут непосредственно обращаться к памяти для этих переменных с помощью таблицы символов.

Более важное преимущество четверок проявляется в оптимизирующем компиляторе, где инструкции зачастую перемещаются. При использовании четверок таблица символов добавляет дополнительную степень косвенности между вычислением значения и его использованием. При перемещении инструкции, вычисляющей *x*, инструкция, использующая это значение, не требует внесения никаких изменений. В случае же использования троек перемещение инструкции, определяющей временное значение, требует от нас изменения всех ссылок на эту инструкцию в массивах *arg1* и *arg2*. Эта проблема затрудняет использование троек в оптимизирующих компиляторах.

При использовании косвенных троек такой проблемы не возникает — перемещение инструкций осуществляется простым переупорядочением списка *statement*. Поскольку указатели на временные значения ссылаются на массив значений *op-arg1-arg2*, который остается неизменным, ни один из этих указателей не нуждается в изменениях при перемещении троек. Таким образом, косвенные тройки очень похожи на четверки с точки зрения их использования. Обе системы записи требуют примерно одинакового количества памяти, и, к тому же, они одинаково эффективны при переупорядочении кода. Как и в случае с обычными тройками, выделение памяти для временных переменных должно быть отложено до фазы генерации целевого кода. Однако косвенные тройки могут сэкономить определенный объем памяти по сравнению с четверками, если некоторое временное значение используется несколько раз. Причина этого в том, что две или более записей таблицы *statement* могут указывать на одну и ту же строку в структуре *op-arg1-arg2*. Так, например, строки (14) и (16) на рис. 8.10 можно объединить так же, как и (после первого объединения, когда (14) и (16) будут представлять одно значение²) строки (15) и (17).

² По сути, такое объединение соответствует превращению дерева в даг на рис. 8.2. — Прим. ред.

8.2. Объявления

Просматривая последовательность объявлений в процедуре или блоке, мы можем распланировать память для локальных имен процедуры. Для каждого локального имени мы создаем запись в таблице символов с информацией о типе и относительном адресе области памяти, выделенной этому имени. Относительный адрес состоит из смещения относительно базового адреса области статических данных или поля для локальных данных в записи активации.

При генерации адресов на начальной стадии компиляции могут учитываться особенности целевой машины. Предположим, что адреса последовательных целых чисел отличаются на 4 на машине с адресацией байтов. Вычисления адресов, генерируемых на начальной стадии, могут при этом включать умножение на 4. Множество инструкций целевой машины также может оказывать влияние на расположение объектов данных в памяти, а следовательно, и на их адреса. Здесь мы не рассматриваем выравнивание объектов данных; в примере 7.3 было показано выравнивание объектов двумя различными компиляторами.

Объявления в процедуре

Синтаксис таких языков программирования, как C, Pascal и Fortran, позволяет обработать все объявления одной процедуры как группу. В этом случае отслеживать очередной свободный относительный адрес можно с помощью глобальной переменной (скажем, *offset*).

В схеме трансляции, приведенной на рис. 8.11, нетерминал *P* порождает последовательность объявлений вида *id:T*. Перед рассмотрением первого объявления *offset* устанавливается равным 0. При работе с очередным именем оно вносится в таблицу символов с значением смещения, равным текущему значению *offset*, а значение *offset* увеличивается на размер объекта данных, обозначаемого данным именем.

Процедура *enter(name, type, offset)* создает запись в таблице символов для *name* и заносит в нее тип *type* и относительный адрес *offset*. Для нетерминала *T* мы используем синтезируемые атрибуты *type* и *width*, указывающие тип и размер, т.е. количество единиц памяти, занимаемой объектами данного типа. Атрибут *type* представляет выражение типа, построенное из базовых типов *integer* и *real* путем применения конструкторов типа *pointer* и *array* (как описывалось в разделе 6.1). Если выражение типа представлено графиком, то атрибут *type* может быть указателем на узел, представляющий выражение типа.

$P \rightarrow$	$\{ \text{offset} := 0; \}$
D	
$D \rightarrow D ; D$	
$D \rightarrow id : T$	$\{ \text{enter}(id.name, T.type, offset);$ $\text{offset} := \text{offset} + T.width \}$
$T \rightarrow integer$	$\{ T.type := \text{integer};$ $T.width := 4 \}$
$T \rightarrow real$	$\{ T.type := \text{real};$ $T.width := 8 \}$
$T \rightarrow array [num] of T_1$	$\{ T.type := \text{array}(num.val, T_1.type);$ $T.width := num.val \times T_1.width \}$
$T \rightarrow \uparrow T_1$	$\{ T.type := \text{pointer}(T_1.type);$ $T.width := 4 \}$

Рис. 8.11. Вычисление типов и относительных адресов объявленных имен

На рис. 8.11 целые числа имеют размер 4, а действительные — 8. Размер массива получаем путем умножения размера каждого элемента массива на количество элементов в массиве³. Размер каждого указателя мы полагаем равным 4. В Pascal и С указатель может появиться до того, как будет объявлен тип объекта, на который он указывает (вспомните обсуждение рекурсивных типов в разделе 6.3). Выделение памяти для таких типов упрощается, если все указатели имеют одинаковый размер.

Инициализация $offset$ в схеме трансляции на рис. 8.11 будет более очевидна, если записать первую продукцию в одну строку.

$$P \rightarrow \{ offset := 0 \} D \quad (8.2)$$

Можно переписать продукцию так, чтобы все действия оказались в концах правых частей, если использовать нетерминалы, порождающие ϵ (в разделе 5.6 они были названы нетерминалами-маркерами). С помощью такого нетерминала M продукцию (8.2) можно переписать следующим образом.

$$P \rightarrow M D$$

$$M \rightarrow \epsilon \{ offset := 0 \}$$

Отслеживание информации об области видимости

В языках с вложенными процедурами относительные адреса могут быть назначены локальным именам с использованием приведенного на рис. 8.11 подхода. При появлении вложенной процедуры обработка объявлений в охватывающей процедуре временно приостанавливается. Этот подход будет проиллюстрирован путем добавления семантических правил к следующему языку.

$$\begin{aligned} P &\rightarrow D \\ D &\rightarrow D ; D \mid id : T \mid proc id ; D ; S \end{aligned} \quad (8.3)$$

Продукции для нетерминалов S (инструкции) и T (типы) здесь не показаны, поскольку в настоящий момент нас интересуют только объявления. Нетерминал T имеет синтезируемые атрибуты *type* и *width*, как и в схеме трансляции, показанной на рис. 8.11.

Для простоты полагаем, что для каждой процедуры языка (8.3) имеется отдельная таблица символов. Одной из возможных реализаций таблицы символов является связанный список записей для имен. При необходимости реализация может быть заменена более подходящей.

Когда встречается объявление процедуры $D \rightarrow proc id D_1 ; S$, создается новая таблица символов и записи для объявлений в D_1 вносятся в нее. Новая таблица указывает на таблицу символов охватывающей процедуры; имя, представленное самим id , является локальным по отношению к охватывающей процедуре. Единственное изменение в трактовке объявлений переменных по сравнению с рис. 8.11 заключается в том, что процедура *enter* получает информацию о том, в какой таблице символов должна создаваться запись.

Например, на рис. 8.12 показаны таблицы символов для пяти процедур. Структура вложенности процедур может быть выведена из связей между таблицами символов. Текст программы можно найти на рис. 7.22. Таблицы символов для процедур *readarray*, *exchange* и *quicksort* указывают на содержащую их процедуру *sort*, которая представляет собой программу целиком. Поскольку *partition* объявлена внутри *quicksort*, ее таблица указывает на таблицу *quicksort*.

³ Для массивов, нижняя граница которых не равна 0, вычисление адреса элементов массива упрощается, если смещение, занесенное в таблицу символов, исправлено так, как рассказывается в разделе 8.3.

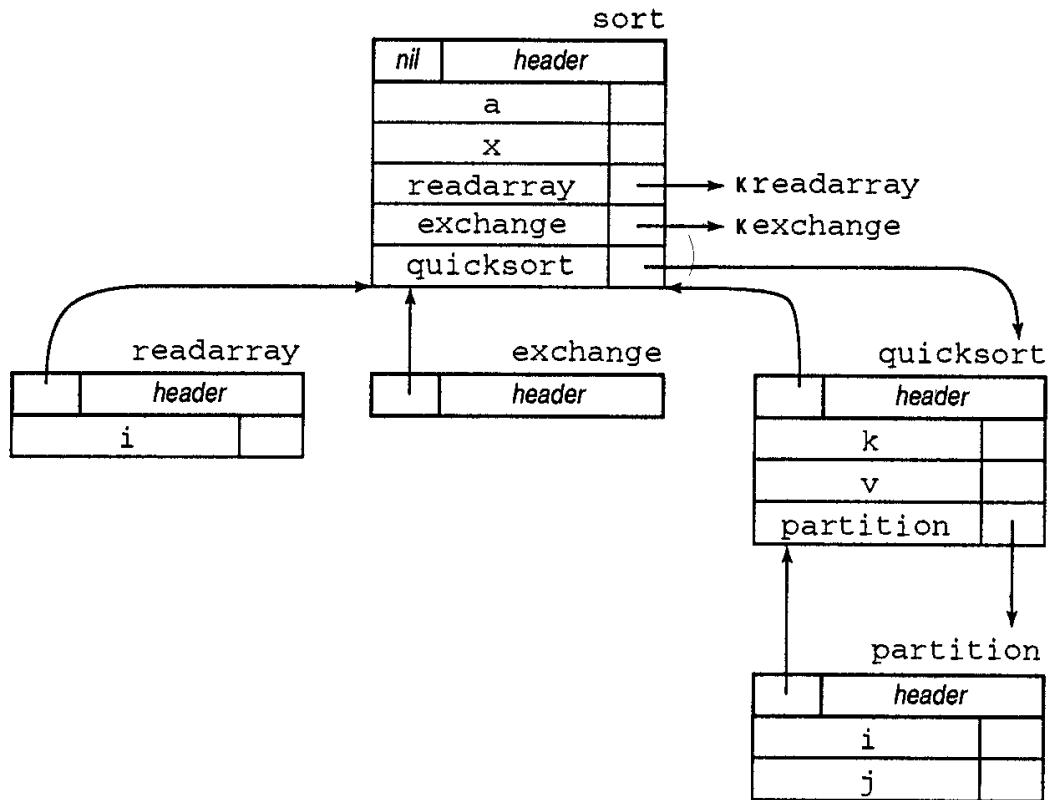


Рис. 8.12. Таблицы символов для вложенных процедур

Семантические правила определяются с использованием следующих операций.

1. *mkttable(previous)* создает новую таблицу символов и возвращает указатель на нее. Аргумент *previous* указывает на предшествующую таблицу, т.е. (предположительно) на таблицу охватывающей процедуры. Указатель *previous* размещается в заголовке новой таблицы символов вместе с дополнительной информацией, такой как глубина вложенности процедуры. Мы также можем пронумеровать процедуры в порядке появления их объявлений и хранить этот номер в заголовке таблицы символов.
2. *enter(table, name, type, offset)* создает новую запись для имени *name* в таблице символов, на которую указывает *table*. Операция *enter* также размещает в полях записи тип *type* и относительный адрес *offset* имени.
3. *addwidth(table, width)* записывает совокупный размер всех элементов таблицы в заголовок, связанный с этой таблицей символов.
4. *enterproc (table, name, newtable)* создает новую запись для процедуры *name* в таблице символов, на которую указывает *table*. Аргумент *newtable* указывает на таблицу символов для этой процедуры.

Схема трансляции, приведенная на рис. 8.13, показывает, каким образом данные могут быть размещены в процессе одного прохода с использованием стека *tblptr* для хранения указателей на таблицы символов охватывающих процедур. Когда с использованием таблиц символов, показанных на рис. 8.12, обрабатываются объявления в процедуре *partition*, стек *tblptr* содержит указатели на таблицы для *sort*, *quicksort* и *partition*. На вершине стека находится указатель на текущую таблицу символов. Другой стек — *offset* — представляет собой естественное обобщение атрибута *offset* из рис. 8.11 для вложенных процедур. Верхний элемент стека *offset* является следующим свободным относительным адресом для локальных имен текущей процедуры.

$P \rightarrow M D$	{ addwidth($\text{top}(\text{tblptr})$, $\text{top}(\text{offset})$); pop(tblptr); pop(offset) }
$M \rightarrow \epsilon$	{ $t := \text{mkttable}(\text{nil})$; $\text{push}(t, \text{tblptr})$; push(0, offset) }
$D \rightarrow D_1 ; D_2$	
$D \rightarrow \text{proc id} ; N D_1 ; S$	{ $t := \text{top}(\text{tblptr})$; addwidth(t , $\text{top}(\text{offset})$); pop(tblptr); pop(offset); $\text{enterproc}(\text{top}(\text{tblptr}), \text{id.name}, t)$ } { $\text{enter}(\text{top}(\text{tblptr}), \text{id.name}, T.\text{type}, \text{top}(\text{offset}))$ }; $\text{top}(\text{offset}) := \text{top}(\text{offset}) + T.\text{width}$ } { $t := \text{mkttable}(\text{top}(\text{tblptr}))$; $\text{push}(t, \text{tblptr})$; push(0, offset) }
$D \rightarrow \text{id} : T$	
$N \rightarrow \epsilon$	

Рис. 8.13. Обработка объявлений во вложенных процедурах

Все семантические действия в поддеревьях для B и C в

$$A \rightarrow B C \{ \text{action}_A \}$$

выполняются до action_A в конце продукции. Следовательно, первым выполняется действие, связанное с маркером M на рис. 8.13.

Действие для нетерминала M инициализирует стек tblptr таблицей символов, созданной операцией $\text{mkttable}(\text{nil})$ для самой внешней области видимости. Кроме того, указанное действие помещает в стек offset относительный адрес 0. Нетерминал N играет аналогичную роль для объявлений в процедуре. Его действие использует операцию $\text{mkttable}(\text{top}(\text{tblptr}))$ для создания новой таблицы символов. Здесь аргумент $\text{top}(\text{tblptr})$ задает охватывающую область видимости для новой таблицы. Указатель на новую таблицу помещается в стек над указателем для охватывающей области видимости. Кроме того, в стек offset вновь вносится значение 0.

Для каждого объявления переменной $\text{id} : T$ в текущей таблице символов создается запись для id . Это объявление оставляет стек tblptr неизмененным; верхний элемент стека offset увеличивается на величину $T.\text{width}$. Когда встречается действие из правой части $D \rightarrow \text{proc id} ; N D_1 ; S$, размер всех объявлений, порождаемых D_1 , находится на вершине стека offset (он записывается сюда с помощью операции addwidth). Затем со стеков tblptr и offset снимаются верхние элементы, и мы возвращаемся к рассмотрению объявлений в охватывающей процедуре. В этот момент имя охватывающей процедуры вносится в таблицу символов охватывающей ее процедуры.

Имена полей в записях

Следующая продукция позволяет нетерминалу T генерировать записи в дополнение к базовым типам, указателям и массивам.

$$T \rightarrow \text{record } D \text{ end}$$

Действия в схеме трансляции, приведенной на рис. 8.14, подчеркивают схожесть организации записей как конструкций языка и записей активации. Поскольку определения про-

цедур не влияют на вычисления размеров (рис. 8.13), мы опустим тот факт, что приведенные выше продукции позволяют определять процедуры внутри записей⁴.

$T \rightarrow \text{record } L D \text{ end}$	{ $T.type := \text{record}(\text{top}(tblptr));$ $T.width := \text{top}(\text{offset});$ $\text{pop}(tblptr); \text{pop}(\text{offset}) \}$
$L \rightarrow \epsilon$	{ $t := \text{mktable}(\text{nil});$ $\text{push}(t, tblptr); \text{push}(0, offset) \}$

Рис. 8.14. Настройка таблицы символов для имен полей в записи

После того как во входном потоке встречается ключевое слово **record**, действие, связанное с маркером L , создает новую таблицу символов для имен полей. Указатель на эту таблицу символов помещается в стек $tblptr$, а относительный адрес 0 — в стек $offset$. Действие для $D \rightarrow \text{id} : T$ (рис. 8.13) таким образом помещает информацию об имени поля **id** в таблицу символов записи. Кроме того, после обработки всей записи вершина стека $offset$ хранит суммарный размер всех объектов данных в записи. Действие, следующее за **end** на рис. 8.14, возвращает этот размер как синтезируемый атрибут $T.width$. Тип $T.type$ получается путем применения конструктора **record** к указателю на таблицу символов для этой записи. Этот указатель будет использоваться в следующем разделе для восстановления имен, типов и размеров полей записи по $T.type$.

8.3. Инструкции присвоения

Выражения в этом разделе могут быть целыми, действительными, массивами и записями. Мы покажем, каким образом в процессе трансляции присвоений в трехадресный код может осуществляться поиск имен в таблице символов и как может производиться доступ к элементам массивов и записей.

Имена в таблице символов

В разделе 8.1 мы создавали трехадресные инструкции с использованием имен, подразумевая, что имена представляют указатели на их записи в таблице символов. Приведенная на рис. 8.15 схема трансляции показывает, каким образом могут быть найдены эти записи таблицы символов. Лексема для имени, представленного **id**, задается атрибутом **id.name**. Операция $\text{lookup}(\text{id}.name)$ проверяет, имеется ли в таблице символов запись для данного имени, и если это так, возвращает указатель на эту запись; в противном случае lookup возвращает указатель **nil**, что означает отсутствие записи в таблице.

$S \rightarrow \text{id} := E$	{ $p := \text{lookup}(\text{id}.name);$ $\text{if } p \neq \text{nil} \text{ then}$ $\quad \text{emit}(p' := E.place)$ $\text{else error} \}$
$E \rightarrow E_1 + E_2$	{ $E.place := \text{newtemp};$ $\text{emit}(E.place' := E_1.place + E_2.place) \}$
$E \rightarrow E_1 * E_2$	{ $E.place := \text{newtemp};$

⁴ Читатели, знакомые с объектно-ориентированным программированием, несомненно, оценят значимость этого факта. — Прим. ред.

$E \rightarrow -E_1$	<i>emit(E.place := E₁.place '*' E₂.place) }</i>
	{ <i>E.place := newtemp;</i>
	<i>emit(E.place := 'uminus' E₁.place) }</i>
$E \rightarrow (E_1)$	{ <i>E.place := E₁.place</i> }
$E \rightarrow \text{id}$	{ <i>p := lookup(id.name);</i>
	<i>if p ≠ nil then</i>
	<i>E.place := p</i>
	<i>else error</i> }

Рис. 8.15. Схема трансляции получения трехадресного кода для присвоений

Семантические действия, показанные на рис. 8.15, используют процедуру *emit* для вывода трехадресного кода в выходной файл, вместо построения атрибута *code* нетерминалов, как на рис. 8.6. Из раздела 2.3 известно, что трансляция может быть выполнена путем вывода в выходной файл, если атрибуты *code* нетерминалов в левой части продукции формируются конкатенацией атрибутов *code* нетерминалов из правой части в том же порядке, в котором нетерминалы встречаются в правой части (возможно, с некоторыми дополнительными строками между ними).

При изменении трактовки операции *lookup* на рис. 8.15 схема трансляции может использоваться даже при применении правила ближайшей вложенной области видимости к нелокальным именам (как в языке Pascal). Предположим, что контекст, в котором появляется присвоение, задается следующей грамматикой.

$$\begin{array}{lcl} P & \rightarrow & MD \\ M & \rightarrow & \epsilon \\ D & \rightarrow & D ; D \mid \text{id} : T \mid \text{proc id} ; ND ; S \\ N & \rightarrow & \epsilon \end{array}$$

При добавлении этих продукции к приведенным на рис. 8.15 нетерминал *P* становится новым стартовым символом.

Для каждой процедуры, порожденной этой грамматикой, схема трансляции на рис. 8.13 создает отдельную таблицу символов. Каждая такая таблица имеет заголовок, содержащий указатель на таблицу для охватывающей процедуры (см., например, рис. 8.12). При рассмотрении инструкции, образующей тело процедуры, на вершине стека *tblptr* появляется указатель на таблицу символов для этой процедуры. Указатель вносится в стек действием, связанным с нетерминалом-маркером *N* в правой части продукции *D → proc id ; ND ; S*.

Пусть рис. 8.15 представляет продукцию для нетерминала *S*. Имена в присвоении, порождаемом *S*, должны быть объявлены либо в процедуре, в которой появляется *S*, либо в некоторой охватывающей процедуре. Применительно к *name*, модифицированная операция *lookup* сначала проверяет, имеется ли *name* в текущей таблице символов, доступной с помощью указателя *top(tblptr)*. Если в текущей таблице символов его нет, *lookup* использует указатель в заголовке таблицы символов для поиска таблицы символов охватывающей процедуры и ищет имя в ней. Если в результате последовательного поиска по всем таблицам символов охватывающих процедур имя не найдено, *lookup* возвращает *nil*.

Предположим, у нас те же таблицы символов, что и на рис. 8.12, и мы рассматриваем присвоение в теле процедуры *partition*. Операция *lookup(i)* найдет запись в таблице символов для *partition*. Поскольку *v* в этой таблице символов нет, *lookup(v)* воспользуется указателем в заголовке этой таблицы символов для продолжения поиска в таблице символов охватывающей процедуры *quicksort*.

Повторное использование временных имен

До сих пор мы полагали, что всякий раз, когда нам требовалась новая временная переменная, она создавалась с помощью *newtemp*. С точки зрения оптимизирующего компилятора при каждом вызове *newtemp* полезно действительно создавать новое уникальное имя; в главе 10, “Оптимизация кода” этому будет дано пояснение. Однако использование временных переменных для хранения промежуточных значений при вычислении выражений быстро переполняет таблицу символов и требует огромного количества памяти для хранения их значений.

Изменив *newtemp*, мы можем использовать временные имена повторно. Другой подход — упаковка различных временных имен в одно и то же место памяти в процессе генерации кода — будет рассмотрен в следующей главе.

Большинство временных имен, обозначающих данные, генерируются в процессе синтаксически управляемой трансляции выражений по правилам наподобие приведенных на рис. 8.15. Код, генерируемый в соответствии с правилами для $E \rightarrow E_1 + E_2$, имеет следующий общий вид.

```
Вычислить  $E_1$  в  $t_1$ 
Вычислить  $E_2$  в  $t_2$ 
 $t := t_1 + t_2$ 
```

Из правил для синтезируемого атрибута $E.place$ следует, что t_1 и t_2 больше нигде в программе не используются. Время жизни этих переменных вложено наподобие пары соответствующих друг другу сбалансированных скобок. Время жизни всех временных переменных, используемых при вычислении E_2 , содержится во времени жизни t_1 . Следовательно, можно модифицировать *newtemp* таким образом, чтобы для хранения временных переменных она использовала небольшой массив в области данных процедуры, как если бы это был стек.

Предположим для простоты, что мы работаем только с целыми числами. Воспользуемся счетчиком c , инициализированным значением 0. При использовании временного имени в качестве операнда счетчик уменьшается на 1, а при генерации нового имени мы используем в качестве имени $\$c$ и увеличиваем c на 1. Заметим, что “стек” временных имен не изменяется в процессе работы программы, хотя иногда компилятор осуществляет сохранение и загрузку временных значений, чтобы оказаться “на вершине стека”.

Пример 8.1

Рассмотрим следующее присвоение.

```
x := a * b + c * d - e * f
```

На рис. 8.16 показана последовательность трехадресных инструкций, генерируемая семантическими правилами (рис. 8.15) с модифицированной процедурой *newtemp*. На рисунке также показаны “текущие” значения c после генерации каждой инструкции. Заметим, что при вычислении $\$0 - \1 значение c уменьшается до нуля, так что $\$0$ вновь становится доступной для хранения результата. \square

Временные переменные, которым может присваиваться значение и/или которые могут использоваться несколько раз (например, в условной инструкции), не могут получать имена описанным стековым способом. Поскольку это достаточно редкое явление, такие временные значения могут получать постоянные имена. Аналогичная проблема временных переменных, определенных или используемых несколько раз, встает и при оптими-

зации кода, такой, например, как объединение общих подвыражений или вынесение вычислений из цикла (см. главу 10, “Оптимизация кода”). Достаточно разумная стратегия состоит в получении нового имени при каждом создании дополнительного определения или использовании временной переменной, а также при перемещении ее вычисления.

ИНСТРУКЦИЯ	ЗНАЧЕНИЕ С
	0
\$0 := a * b	1
\$1 := c * d	2
\$0 := \$0 + \$1	1
\$1 := e * f	2
\$0 := \$0 - \$1	1
x := \$0	0

Рис. 8.16. Трехадресный код для стековых временных имен

Адресация элементов массива

Быстрый доступ к элементам массива легко осуществляется, когда элементы хранятся в блоке из последовательных ячеек памяти. Если размер каждого элемента массива — w , то i -й элемент массива A начинается с адреса

$$base + (i - low) \times w, \quad (8.4)$$

где low — нижняя граница индекса, а $base$ — относительный адрес памяти, выделенной под массив, т.е. относительный адрес $A[low]$.

Выражение (8.4) может быть частично вычислено во время компиляции, если переписать его как $i \times w + (base - low \times w)$. Подвыражение $c = base - low \times w$ можно вычислить в момент обработки объявления массива. Полагая, что величина c хранится в записи таблицы символов для A , относительный адрес $A[i]$ мы получим просто прибавлением $i \times w$ к c .

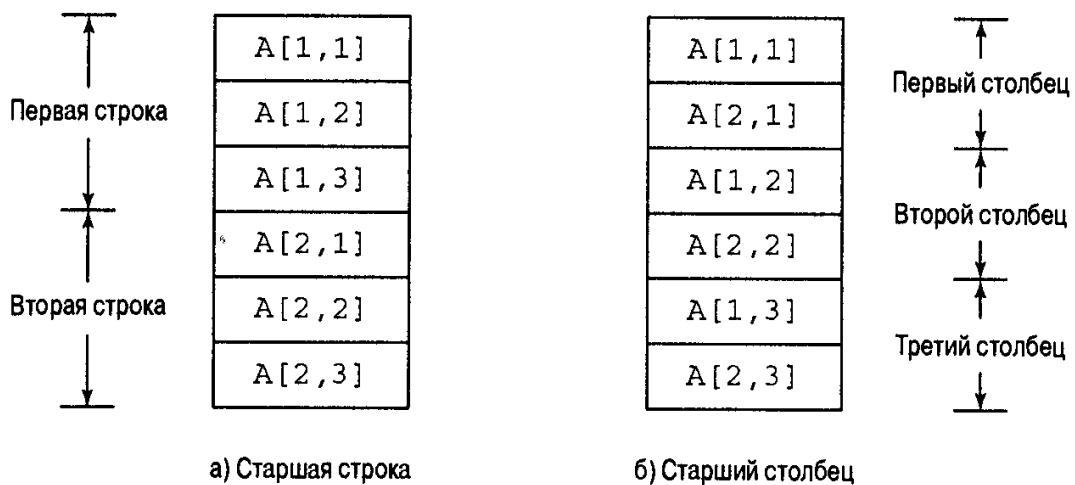


Рис. 8.17. Размещение в памяти элементов двухмерного массива

Предвычисления в процессе компиляции могут быть применены и для вычисления адресов элементов многомерных массивов. Двухмерный массив обычно хранится либо со *старшей строкой* (по строкам), либо со *старшим столбцом* (по столбцам). На

рис. 8.17 показаны описанные способы размещения в памяти элементов массива A размером 2×3 . Fortran использует способ старшего столбца, а Pascal — старшей строки, поскольку $A[i, j]$ эквивалентно $A[i][j]$, а элементы каждого массива $A[i]$ хранятся последовательно⁵.

В случае двухмерного массива, хранимого по строкам, относительный адрес $A[i_1, i_2]$ может быть вычислен по формуле

$$base + ((i_1 - low_1) \times n_2 + i_2 - low_2) \times w,$$

где low_1 и low_2 — нижние границы значений i_1 и i_2 , а n_2 — число значений, которые может принимать i_2 . Таким образом, если $high_2$ — верхняя граница значения i_2 , то $n_2 = high_2 - low_2 + 1$. Полагая, что в процессе компиляции неизвестны только i_1 и i_2 , можно переписать приведенное выше выражение следующим образом.

$$((i_1 \times n_2) + i_2) \times w + (base - ((low_1 \times n_2) + low_2) \times w) \quad (8.5)$$

Второе слагаемое этого выражения можно вычислить в процессе компиляции.

Мы можем обобщить хранение по строкам и столбцам на случай многомерного массива. Обобщение способа старшей строки состоит в хранении элементов массива таким образом, что при последовательном сканировании блока памяти самый правый индекс изменяется наиболее быстро, как цифры на спидометре автомобиля. Выражение (8.5) обобщается в этом случае в следующее выражение для относительного адреса $A[i_1, i_2, \dots, i_k]$.

$$\begin{aligned} & ((\dots((i_1 n_2 + i_2) n_3 + i_3) \dots) n_k + i_k) \times w \\ & + base - ((\dots((low_1 n_2 + low_2) n_3 + low_3) \dots) n_k + low_k) \times w \end{aligned} \quad (8.6)$$

Поскольку для всех j $n_j = high_j - low_j + 1$ считается фиксированным, слагаемое во второй строке (8.6) может быть вычислено компилятором и сохранено в записи таблицы символов для A⁶. Способ хранения по столбцам обобщается в противоположное размещение, при котором быстрее всего изменяется левый индекс.

Некоторые языки допускают динамическое указание размера массивов при вызове процедуры в процессе выполнения программы. Размещение таких массивов в стеке времени исполнения рассматривалось в разделе 7.3. Формулы для доступа к элементам таких массивов аналогичны формулам для массивов фиксированных размеров, с той лишь разницей, что верхние и нижние границы неизвестны в процессе компиляции.

Основная проблема при генерации кода для работы с элементами массива состоит в том, чтобы связать вычисление (8.6) с грамматикой для работы с массивами. Ссылки на элементы массива в присвоениях могут быть разрешены, если вместо *id* на рис. 8.15 использовать нетерминал *L* со следующими продукциями.

$$\begin{aligned} L & \rightarrow id [Elist] | id \\ Elist & \rightarrow Elist , E | E \end{aligned}$$

⁵ После этого совершенно очевидно, что в C, где обращение к элементам массива изначально имеет вид $a[i][j]$, также используется способ старшей строки. — Прим. ред.

⁶ В C многомерный массив определяется как массив, элементы которого являются массивами. Предположим, например, что x — массив массивов целых чисел. Тогда язык допускает как запись $x[i]$, так и $x[i][j]$, и размеры этих выражений различны. Однако нижняя граница любого массива в C равна 0, так что слагаемое во второй строке (8.6) упрощается до *base*.

Для того чтобы при группировании выражений индексы в *Elist* были допустимы различные пределы n_j у разных размерностей массива, следует переписать указанные продукции следующим образом.

$$\begin{array}{l} L \rightarrow Elist] | id \\ Elist \rightarrow Elist, E | id [E \end{array}$$

Таким образом, имя массива в процессе формирования *L* присоединяется к левому выражению индекса, а не к *Elist*. Эти продукции позволяют передать в качестве синтезируемого атрибута *Elist.array* указатель на запись в таблице символов для имени массива⁷.

Мы также используем *Elist.ndim* для записи количества размерностей (индексных выражений) в *Elist*. Функция *limit(array, j)* возвращает n_j , количество элементов j -й размерности массива, на запись которого в таблице символов указывает *array*. И наконец, *Elist.place* обозначает временную переменную, хранящую значение, вычисленное по индексному выражению в *Elist*.

Elist, порождающий первые m индексов ссылки на элемент k -мерного массива $A[i_1, i_2, \dots, i_k]$, будет генерировать трехадресный код для вычисления

$$(\cdots((i_1 n_2 + i_2) n_3 + i_3) \cdots) n_m + i_m, \quad (8.7)$$

используя рекуррентные соотношения

$$\begin{aligned} e_1 &= i_1 \\ e_m &= e_{m-1} \times n_m + i_m \end{aligned} \quad (8.8)$$

Таким образом, когда $m = k$, все, что нужно для вычисления члена в первой строке (8.6), — это умножение на размер w . Заметим, что здесь i_j представляет значение выражения, и код для его вычисления внедрен в код для вычисления (8.7).

l-значение *L* будет иметь два атрибута — *L.place* и *L.offset*. Если *L* представляет собой простое имя, *L.place* является указателем на запись в таблице символов для этого имени, а *L.offset* имеет значение *null*, указывающее, что *l*-значение — простое имя, а не ссылка на элемент массива. Нетерминал *E* имеет такую же трансляцию, связанную с атрибутом *E.place*, что и на рис. 8.15.

Схема трансляции для адресации элементов массива

Семантические действия будут добавляться к следующей грамматике.

- (1) $S \rightarrow L := E$
- (2) $E \rightarrow E + E$
- (3) $E \rightarrow (E)$
- (4) $E \rightarrow L$
- (5) $L \rightarrow Elist]$
- (6) $L \rightarrow id$
- (7) $Elist \rightarrow Elist, E$
- (8) $Elist \rightarrow id [E$

Как и в случае выражений без ссылок на элементы массива, трехадресный код производится процедурой *emit*, вызываемой в семантических действиях.

Мы генерируем обычное присвоение, если *L* представляет собой простое имя, и индексированное присвоение по адресу, обозначенному *L*, в противном случае.

⁷ Это преобразование похоже на упоминавшееся в конце раздела 5.6 для устранения наследуемых атрибутов. Здесь мы также решаем проблему наследуемых атрибутов.

(1) $S \rightarrow L := E$

```
{ if  $L.offset = null$  then /*  $L$  — простое имя */
    emit( $L.place$  ' := '  $E.place$ );
else
    emit( $L.place$  '['  $L.offset$  '] ' := '  $E.place$  ) }
```

Код для арифметических выражений тот же, что и на рис. 8.15.

(2) $E \rightarrow E_1 + E_2$

```
{  $E.place$  := newtemp;
emit( $E.place$  ' := '  $E_1.place$  '+'  $E_2.place$  ) }
```

(3) $E \rightarrow (E_1)$

```
{  $E.place$  :=  $E_1.place$  }
```

Когда ссылка на массив L свертывается в E , нам требуется r -значение L . Таким образом, мы используем индексацию для получения содержимого ячейки памяти $L.place[L.offset]$.

(4) $E \rightarrow L$

```
{ if  $L.offset = null$  then /*  $L$  — простой id */
     $E.place$  :=  $L.place$ 
else begin
     $E.place$  := newtemp;
    emit( $E.place$  ' := '  $L.place$  '['  $L.offset$  '] ')
end }
```

Ниже $L.offset$ является новой временной переменной, представляющей первое слагаемое в (8.6); функция $width(Elist.array)$ возвращает значение w в (8.6). $L.place$ представляет второе слагаемое в (8.6), возвращаемое функцией $c(Elist.array)$.

(5) $L \rightarrow Elist [$

```
{  $L.place$  := newtemp;
 $L.offset$  := newtemp;
emit( $L.place$  ' := '  $c(Elist.array)$ );
emit( $L.offset$  ' := '  $Elist.place$  '*'  $width(Elist.array)$  ) }
```

Нулевое смещение указывает на простое имя.

(6) $L \rightarrow id$

```
{  $L.place$  :=  $id.place$ ;
 $L.offset$  := null }
```

При рассмотрении следующего индексного выражения мы применяем рекуррентное соотношение (8.8). В следующем действии $Elist_1.place$ соответствует e_{m-1} из (8.8), а $Elist.place$ — e_m . Заметим, что если $Elist_1$ имеет $m-1$ компонент, то $Elist$ в левой части продукции имеет m компонентов.

(7) $Elist \rightarrow Elist_1 , E$

```
{  $t$  := newtemp;
 $m$  :=  $Elist_1.ndim$  + 1;
emit( $t$  ' := '  $Elist_1.place$  '*'  $limit(Elist_1.array, m)$ );
emit( $t$  ' := '  $t$  '+'  $E.place$ );
 $Elist.array$  :=  $Elist_1.array$ ;
 $Elist.place$  :=  $t$ ;
 $Elist.ndim$  :=  $m$  }
```

$E.place$ хранит как значение выражения E , так и значение (8.7) для $m = 1$ ⁸.

(8) $Elist \rightarrow id [E$

```
{  $Elist.array$  :=  $id.place$ ;
 $Elist.place$  :=  $E.place$ ;
 $Elist.ndim$  := 1 }
```

⁸ При $m = 1$ эти значения, по сути, одинаковы. — Прим. ред.

Пример 8.2

Пусть A — массив размером 10×20 с $low_1 = low_2 = 1$. Таким образом, $n_1 = 10$, а $n_2 = 20$. Примем w равным 4. Аннотированное дерево разбора для присвоения $x := A[y, z]$ показано на рис. 8.18. Присвоение транслируется в следующую последовательность трехадресных инструкций.

```

 $t_1 := y * 20$ 
 $t_1 := t_1 + z$ 
 $t_2 := c$            /* Константа  $c = base_A - 84$  */
 $t_3 := 4 * t_1$ 
 $t_4 := t_2[t_3]$ 
 $x := t_4$ 

```

Для каждой переменной вместо $id.place$ мы использовали ее имя. □

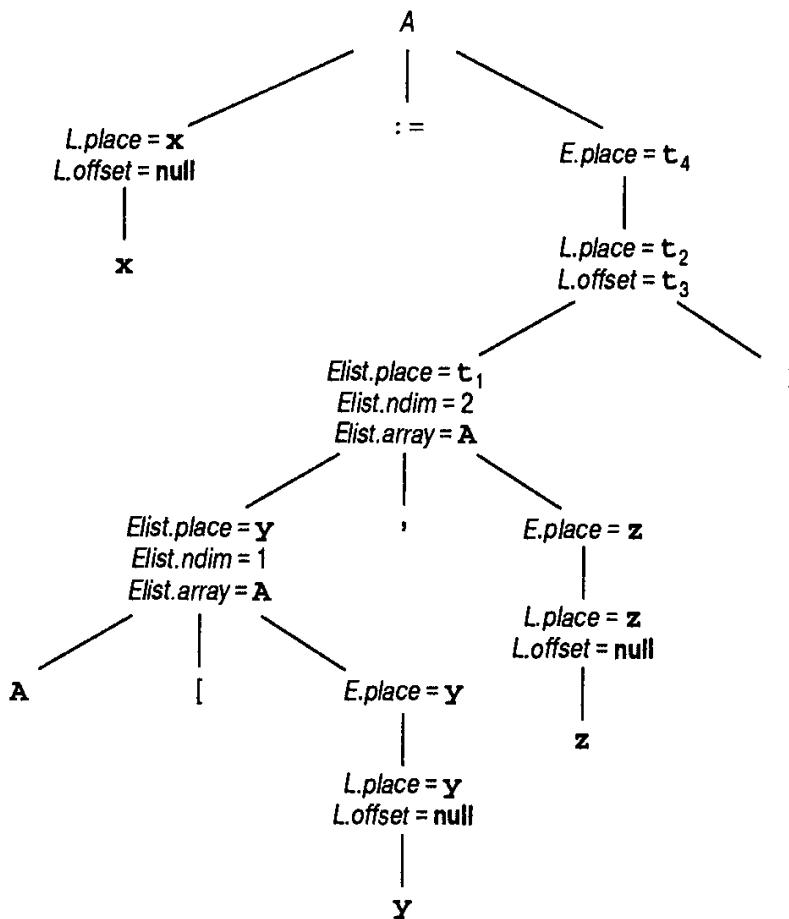


Рис. 8.18. Аннотированное дерево разбора для $x := A[y, z]$

Преобразования типов в присвоениях

На практике имеется множество различных типов переменных и констант, так что компилятор должен либо отвергать некоторые операции с разными типами, либо генерировать соответствующие инструкции преобразования типов.

Рассмотрим грамматику для присвоений, представленную ранее, но предположим, что у нас имеется два типа — действительные и целые, причем при необходимости целые числа преобразуются в действительные. Мы введем еще один атрибут $E.type$, значе-

ние которого либо *real*, либо *integer*. Семантическое правило для $E.type$, связанное с продукцией $E \rightarrow E_1 + E_2$, будет следующим.

$E \rightarrow E_1 + E_2 \quad \{ E.type :=$
 if $E_1.type = integer$ and
 $E_2.type = integer$ then *integer*
 else *real* }

Это правило — в духе раздела 6.4. Однако здесь и далее во всей главе мы опускаем проверки ошибок типов; этому посвящена глава 6, “Проверка типов”.

Семантические правила для $E \rightarrow E + E$ и большинства других продукции должны быть модифицированы при необходимости генерации трехадресных инструкций вида $x := inttoreal y$, действие которых состоит в преобразовании целого y в эквивалентное действительное значение x . Мы должны также включить вместе с кодом оператора указание, какой арифметический оператор подразумевается — для чисел с фиксированной или плавающей точкой. Полностью семантическое действие для продукции вида $E \rightarrow E_1 + E_2$ приведено на рис. 8.19.

```
E.place := newtemp;
if E1.type = integer and E2.type = integer then begin
    emit(E.place ':= E1.place 'int+' E2.place);
    E.type := integer
end
else if E1.type = real and E2.type = real then begin
    emit(E.place ':= E1.place 'real+' E2.place);
    E.type := real
end
else if E1.type = integer and E2.type = real then begin
    u := newtemp;
    emit(u ':= 'inttoreal' E1.place);
    emit(E.place ':= u 'real+' E2.place);
    E.type := real
end
else if E1.type = real and E2.type = integer then begin
    u := newtemp;
    emit(u ':= 'inttoreal' E2.place);
    emit(E.place ':= E1.place 'real+' u);
    E.type := real
end
else
    E.type := type_error;
```

Рис. 8.19. Семантическое действие для $E \rightarrow E_1 + E_2$

Например, пусть во входном потоке

$x := y + i * j$

x и y имеют тип *real*, а i и j — тип *integer*. Тогда вывод будет выглядеть следующим образом.

$t_1 := i \ int* j$
 $t_3 := inttoreal t_1$

```
t2 := y real+ t3
x := t2
```

Семантическое действие на рис. 8.19 использует два атрибута нетерминала E — $E.place$ и $E.type$. При увеличении числа преобразуемых типов количество возникающих при этом вариантов увеличивается квадратично (или еще в большей степени при операторах с более чем двумя аргументами). Соответственно, при большом количестве используемых типов правильная организация семантических действий становится особо важной.

Доступ к полям записей

Компилятор должен отслеживать как типы, так и относительные адреса полей записи. Преимущество хранения этой информации в отдельных записях таблицы символов заключается в том, что подпрограмма, используемая для поиска имен в таблице символов, может быть использована и для имен полей. С учетом этого в предыдущем разделе семантические действия на рис. 8.14 создавали для каждого типа записи отдельную таблицу символов. Если t — указатель на таблицу символов для типа записи, то тип $record(t)$ создается путем применения конструктора $record$ к этому указателю, который возвращается в виде атрибута $T.type$.

Для иллюстрации того, как указатель на таблицу символов извлекается из атрибута $E.type$, используем выражение $p^{\uparrow}.info + 1$. Из операций этого выражения следует, что p должно быть указателем на запись, в которой есть поле $info$ арифметического типа. Если типы строятся так, как на рис. 8.13 и 8.14, тип p должен быть задан с помощью выражения типа $pointer(record(t))$. Тогда тип p^{\uparrow} представляет собой $record(t)$, откуда можно извлечь t . Поиск имени поля $info$ осуществляется в таблице символов, на которую указывает t .

8.4. Логические выражения

В языках программирования логические выражения предназначены для вычисления логических значений. Чаще всего они используются в качестве условных выражений в инструкциях, изменяющих обычное течение потока управления, таких как **if-then-else** или **while-do**.

Логические выражения состоят из логических операторов (**and**, **or** и **not**), применяемых к элементам, которыми являются логические переменные или выражения отношения. Выражения отношения, в свою очередь, имеют вид $E_1 \text{ relop } E_2$, где E_1 и E_2 — арифметические выражения. Некоторые языки, такие как PL/I или C, позволяют использовать более общие выражения, в которых логические и арифметические операторы, а также операторы отношения могут применяться к выражениям любого типа, без различия между логическими и арифметическими значениями; при необходимости выполняются соответствующие преобразования типов. В этом разделе мы рассмотрим логические выражения, порождаемые следующей грамматикой.

$$E \rightarrow E \text{ or } E \text{ and } E \text{ | not } E \text{ | (} E \text{) | id relop id | true | false}$$

Для определения того, какой именно из операторов сравнения ($<$, \leq , $=$, \neq , $>$ или \geq) представляет **relop**, используется атрибут **or**. Как принято, мы полагаем, что **and** и **or** левоассоциативны и приоритеты операторов **or**, **and** и **not** возрастают в указанном здесь порядке.

Методы трансляции логических выражений

Существует два принципиально разных метода представления значений логических выражений. Первый метод представляет значения `true` и `false` в виде чисел, и логическое выражение вычисляется аналогично арифметическому. Часто для представления `true` используется значение 1, а для `false` — 0 (хотя возможны и другие способы представления). Например, мы можем считать, что любое ненулевое значение представляет `true`, а нулевое — `false`; или же, что `true` представлено неотрицательными значениями, а отрицательные числа представляют `false`.

Второй метод реализации логических выражений использует поток управления, т.е. представляет значение логического выражения положением, достигнутым в программе. Этот метод удобен при реализации логических выражений в инструкциях изменения потока управления, таких как `if-then` или `while-do`. Например, если в выражении $E_1 \text{ or } E_2$ мы определили, что E_1 имеет значение `true`, мы можем заключить, что выражение в целом истинно, не вычисляя E_2 .

Должны ли вычисляться все части логического выражения, — определяется семантикой языка программирования. Если определение языка позволяет (или требует), чтобы части логического выражения оставались не вычисленными, компилятор может оптимизировать вычисления логических выражений и производить только необходимые для определения значения выражения вычисления. Таким образом, в выражении типа $E_1 \text{ or } E_2$ не гарантируется полное вычисление ни выражения E_1 , ни E_2 . Если либо E_1 , либо E_2 представляют собой выражения с побочным эффектом (например, содержат функции, изменяющие величину глобальной переменной), в результате работы такой программы можно получить неожиданный (и неверный) ответ.

Ни один из этих методов явным образом не превосходит другой. Например, оптимизирующий компилятор BLISS/11 [460], как и некоторые другие, выбирает подходящий метод вычислений для каждого логического выражения отдельно. В этом разделе мы рассмотрим оба метода с точки зрения трансляции логических выражений в трехадресный код.

Числовое представление

Вначале рассмотрим реализацию логических выражений, в которой 1 отображает `true`, а 0 — `false`. Выражения будут вычисляться полностью, слева направо, как и арифметические выражения. Например, трансляция

`a or b and not c`

представляет следующую трехадресную последовательность.

```
t1 := not c
t2 := b and t1
t3 := a or t2
```

Выражение отношения, такое как `a < b`, эквивалентно условной инструкции `if a < b then 1 else 0`, которая может быть транслирована в следующую трехадресную последовательность (полагая, что мы начинаем нумерацию инструкций со 100).

```
100:    if a < b goto 103
101:    t := 0
102:    goto 104
103:    t := 1
104:
```

Схема трансляции для получения трехадресного кода логических выражений показана на рис. 8.20. Здесь мы полагаем, что *emit* выводит трехадресный код в выходной файл в требуемом виде, *nextstat* дает индекс очередной трехадресной инструкции в выходной последовательности, а *emit* увеличивает *nextstat* после вывода каждой трехадресной инструкции.

$E \rightarrow E_1 \text{ or } E_2$	{ <i>E.place</i> := newtemp; emit(<i>E.place</i> ' := 'E ₁ .place' or' E ₂ .place) }
$E \rightarrow E_1 \text{ and } E_2$	{ <i>E.place</i> := newtemp; emit(<i>E.place</i> ' := 'E ₁ .place' and' E ₂ .place) }
$E \rightarrow \text{not } E_1$	{ <i>E.place</i> := newtemp; emit(<i>E.place</i> ' := 'not' E ₁ .place) }
$E \rightarrow (E)$	{ <i>E.place</i> := <i>E.place</i> }
$E \rightarrow \text{id}_1 \text{ relop id}_2$	{ <i>E.place</i> := newtemp; emit('if' <i>id</i> ₁ .place <i>relop</i> .op <i>id</i> ₂ .place 'goto' <i>nextstat</i> +3); emit(<i>E.place</i> ' := '0'); emit('goto' <i>nextstat</i> +2); emit(<i>E.place</i> ' := '1') }
$E \rightarrow \text{true}$	{ <i>E.place</i> := newtemp; emit(<i>E.place</i> ' := '1') }
$E \rightarrow \text{false}$	{ <i>E.place</i> := newtemp; emit(<i>E.place</i> ' := '0') }

Рис. 8.20. Схема трансляции, использующая числовое представление логических величин

Пример 8.3

Схема трансляции (рис. 8.20) для выражения $a < b \text{ or } c < d \text{ and } e < f$ производит трехадресный код, показанный на рис. 8.21. \square

100: if <i>a</i> < <i>b</i> goto 103	107: <i>t2</i> := 1
101: <i>t1</i> := 0	108: if <i>e</i> < <i>f</i> goto 111
102: goto 104	109: <i>t3</i> := 0
103: <i>t1</i> := 1	110: goto 112
104: if <i>c</i> < <i>d</i> goto 107	111: <i>t3</i> := 1
105: <i>t2</i> := 0	112: <i>t4</i> := <i>t2</i> and <i>t3</i>
106: goto 108	113: <i>t5</i> := <i>t1</i> or <i>t4</i>

Рис. 8.21. Трансляция $a < b \text{ or } c < d \text{ and } e < f$

Сокращенные вычисления

Логическое выражение можно транслировать в трехадресный код без генерации кода для каждого из логических операторов и без обязательного наличия кода для вычисления всего логического выражения. Такой способ вычисления иногда называется сокращенным (short-circuit), или “перепрыгивающим” (jumping) кодом. Можно вычислить логическое выражение без генерации кода для логических операторов *and*, *or* или *not*, если мы представим значение выражения положением в последовательности кода. Например, на рис. 8.21 мы можем сказать, что значение *t1* определяется тем, достигаем ли мы инструкции 101 или 103, так что само значение *t1* оказывается избыточным. Для многих логических выражений можно определить значение выражения, не вычисляя его полностью.

Инструкции потока управления

Теперь мы рассмотрим трансляцию логических выражений в трехадресный код в контексте инструкций **if-then**, **if-then-else** и **while-do**, порождаемых следующей грамматикой.

$$S \rightarrow \begin{array}{l} \text{if } E \text{ then } S_1 \\ | \quad \text{if } E \text{ then } S_1 \text{ else } S_2 \\ | \quad \text{while } E \text{ do } S_1 \end{array}$$

В каждой из этих продукции E является транслируемым логическим выражением. При трансляции мы полагаем, что трехадресные выражения могут иметь символьные метки и что функция *newlabel* при каждом вызове возвращает новое символьное имя метки.

С логическим выражением E мы связываем две метки: $E.true$ (для передачи потока управления, если E истинно) и $E.false$ (если E ложно). Семантические правила для трансляции инструкций управления потоком S обеспечивают переход от трансляции $S.code$ к трехадресной инструкции, непосредственно следующей за $S.code$. В некоторых случаях инструкция, следующая непосредственно за S , представляет собой безусловный переход к некоторой метке L . Перехода к переходу к L в коде $S.code$ можно избежать, используя наследуемый атрибут $S.next$. Значение $S.next$ представляет собой метку, назначенную первой трехадресной инструкции, выполняемой после кода S^9 . Инициализация $S.next$ не показана.

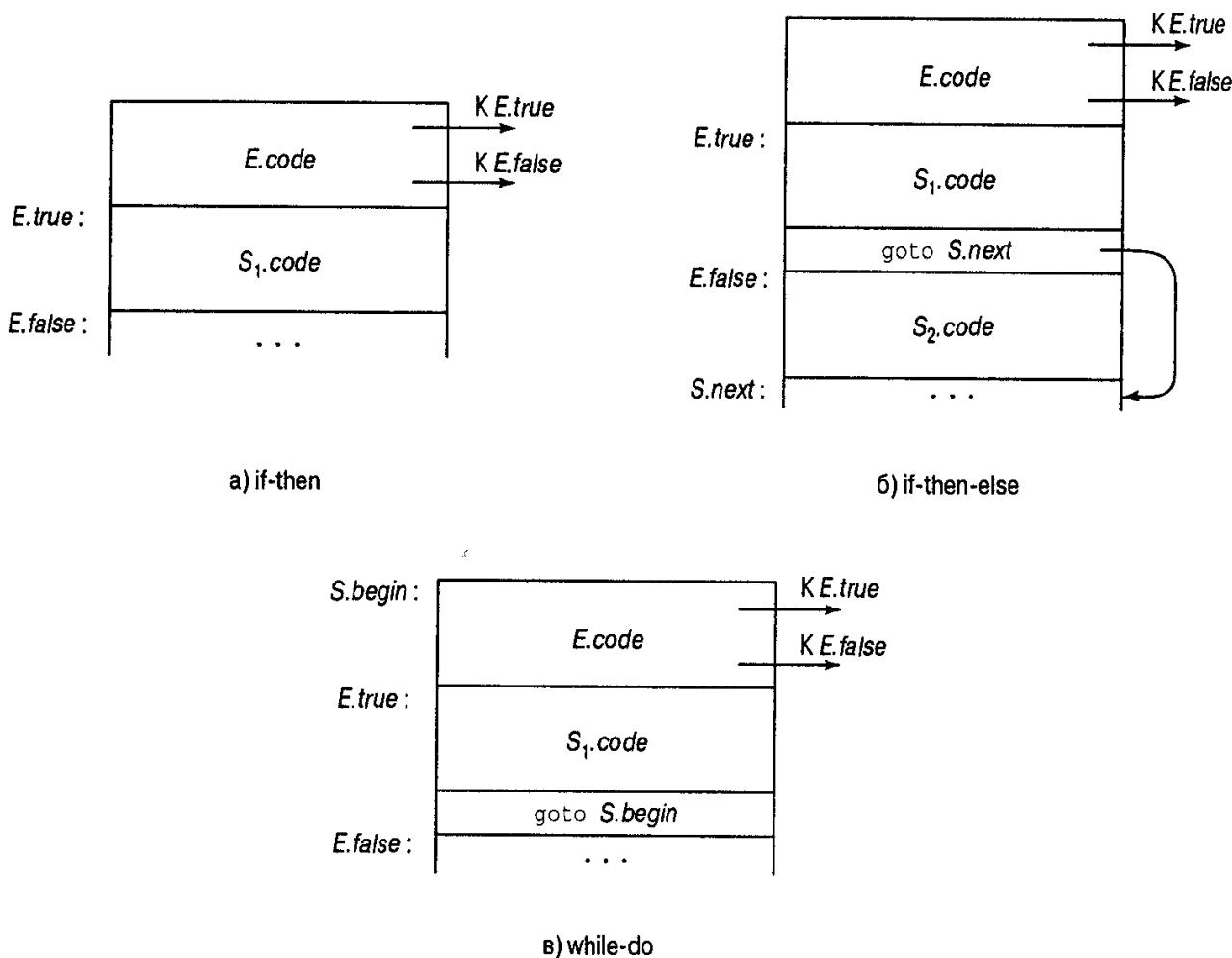


Рис. 8.22. Код инструкций *if-then*, *if-then-else* и *while-do*

⁹ При буквальной реализации подход с использованием наследуемой метки $S.next$ может привести к огромному росту количества меток. Подход с применением обратных поправок из раздела 8.6 создает метки только при необходимости.

При трансляции инструкции $S \rightarrow \text{if } E \text{ then } S_1$ создается новая метка $E.\text{true}$ для первой трехадресной инструкции, генерируемой для S_1 , как показано на рис. 8.22а. Синтаксически управляемое определение приведено на рис. 8.23. Код для E генерирует переход к $E.\text{true}$, если E истинно, и к $S.\text{next}$, если E ложно. Таким образом, мы устанавливаем $E.\text{false}$ равным $S.\text{next}$.

ПРОДУКЦИЯ	СЕМАНТИЧЕСКИЕ ПРАВИЛА
$S \rightarrow \text{if } E \text{ then } S_1$	$E.\text{true} := \text{newlabel};$ $E.\text{false} := S.\text{next};$ $S_1.\text{next} := S.\text{next};$ $S.\text{code} := E.\text{code} $ $\quad \text{gen}(E.\text{true} ':') S_1.\text{code}$
$S \rightarrow \text{if } E \text{ then } S_1 \text{ else } S_2$	$E.\text{true} := \text{newlabel};$ $E.\text{false} := \text{newlabel};$ $S_1.\text{next} := S.\text{next};$ $S_2.\text{next} := S.\text{next};$ $S.\text{code} := E.\text{code} $ $\quad \text{gen}(E.\text{true} ':') S_1.\text{code} $ $\quad \text{gen}(\text{goto}' S.\text{next}) $ $\quad \text{gen}(E.\text{false} ':') S_2.\text{code}$
$S \rightarrow \text{while } E \text{ do } S_1$	$S.\text{begin} := \text{newlabel};$ $E.\text{true} := \text{newlabel};$ $E.\text{false} := S.\text{next};$ $S_1.\text{next} := S.\text{begin}$ $S.\text{code} := \text{gen}(S.\text{begin} ':') E.\text{code} $ $\quad \text{gen}(E.\text{true} ':') S_1.\text{code} $ $\quad \text{gen}(\text{goto}' S.\text{begin})$

Рис. 8.23. Синтаксически управляемое определение для инструкций потока управления

При трансляции инструкции $S \rightarrow \text{if } E \text{ then } S_1 \text{ else } S_2$ код для логического выражения E содержит переход к первой инструкции S_1 для истинного E и к первой инструкции S_2 — для ложного E , как показано на рис. 8.22б. Как и при использовании инструкции if-then, наследуемый атрибут $S.\text{next}$ определяет метку для трехадресной инструкции, выполняемой непосредственно после кода S . Явное goto $S.\text{next}$ добавляется после кода S_1 , но не после S_2 . Читателю в качестве упражнения предлагается доказать, что если при данных семантических правилах $S.\text{next}$ не является меткой инструкции, следующей непосредственно за $S.\text{code}$, то охватывающая инструкция обеспечит переход к метке $S.\text{next}$ после кода для S_2 .

Код для $S \rightarrow \text{while } E \text{ do } S_1$ строится таким образом, как показано на рис. 8.22в. Создается новая метка $S.\text{begin}$, которая назначается первой инструкции, генерируемой для E . Еще одна новая метка $E.\text{true}$ назначается первой инструкции кода для S_1 . Код для E генерирует переход к этой метке, если E истинно, и к $S.\text{next}$ — если E ложно (так что мы опять назначаем для $E.\text{false}$ значение $S.\text{next}$). После кода S_1 мы размещаем инструкцию goto $S.\text{begin}$, которая задает переход к началу вычисления логического выражения. Заметим, что $S_1.\text{next}$ устанавливается равным метке $S.\text{begin}$, так что переход из кода $S_1.\text{code}$ осуществляется непосредственно к $S.\text{begin}$ ¹⁰.

¹⁰ По сути, все сказанное вытекает из того, что инструкция $\text{while } E \text{ do } S_1$ эквивалентна следующей:
 $\text{Loop: if } E \text{ then begin } S_1; \text{ goto Loop end. — Прим. ред.}$

Более подробно мы обсудим трансляцию инструкций потока в разделе 8.6, где познакомимся с методом обратных поправок, генерирующих код для таких инструкций за один проход.

Трансляция логических выражений с помощью потока управления

Теперь обсудим $E.\text{code}$ — код, генерируемый для логического выражения E на рис. 8.23. Как мы указывали, E транслируется в последовательность трехадресных инструкций, которые вычисляют E как последовательность условных и безусловных переходов к одному из двух положений: $E.\text{true}$ (при истинности E) и $E.\text{false}$ (при ложности E).

Основная идея такой трансляции состоит в следующем. Предположим, что E имеет вид $a < b$. Тогда генерируемый код выглядит следующим образом.

```
if a < b goto E.true
goto E.false
```

Предположим, что E имеет вид $E_1 \text{ or } E_2$. Если E_1 истинно, значит, истинно и E , так что $E.\text{true}$ равно $E_1.\text{true}$. Если E_1 ложно, должно быть вычислено E_2 , так что $E.\text{false}$ должно быть меткой первой инструкции кода для E_2 . В этом случае переходы из кода для E_2 должны быть такими же, как и переходы для кода E в целом.

Аналогично рассматривается и трансляция $E_1 \text{ and } E_2$. В случае вычисления $\text{not } E_1$ никакой код для всего выражения не нужен: мы просто меняем местами метки переходов для истинного и ложного значений выражения. Синтаксически управляемое определение, генерирующее соответствующий трехадресный код для логических выражений, представлено на рис. 8.24. Заметьте, что атрибуты *true* и *false* — наследуемые.

ПРОДУКЦИЯ	СЕМАНТИЧЕСКИЕ ПРАВИЛА
$E \rightarrow E_1 \text{ or } E_2$	$E_1.\text{true} := E.\text{true};$ $E_1.\text{false} := \text{newlabel};$ $E_2.\text{true} := E.\text{true};$ $E_2.\text{false} := E.\text{false};$ $E.\text{code} := E_1.\text{code} \parallel \text{gen}(E_1.\text{false} ':') \parallel E_2.\text{code}$
$E \rightarrow E_1 \text{ and } E_2$	$E_1.\text{true} := \text{newlabel};$ $E_1.\text{false} := E.\text{false};$ $E_2.\text{true} := E.\text{true};$ $E_2.\text{false} := E.\text{false};$ $E.\text{code} := E_1.\text{code} \parallel \text{gen}(E_1.\text{true} ':') \parallel E_2.\text{code}$
$E \rightarrow \text{not } E_1$	$E_1.\text{true} := E.\text{false};$ $E_1.\text{false} := E.\text{true};$ $E.\text{code} := E_1.\text{code}$
$E \rightarrow (E_1)$	$E_1.\text{true} := E.\text{true};$ $E_1.\text{false} := E.\text{false};$ $E.\text{code} := E_1.\text{code}$
$E \rightarrow \text{id}_1 \text{ relop } \text{id}_2$	$E.\text{code} := \text{gen}('if' \text{ id}_1.\text{place} \text{ relop.} \text{op id}_2.\text{place}$ $\quad \quad \quad \text{'goto' } E.\text{true}) \parallel \text{gen}('goto' E.\text{false})$
$E \rightarrow \text{true}$	$E.\text{code} := \text{gen}('goto' E.\text{true})$
$E \rightarrow \text{false}$	$E.\text{code} := \text{gen}('goto' E.\text{false})$

Рис. 8.24. Синтаксически управляемое определение для генерации трехадресного кода логических выражений

Пример 8.4

Вновь обратимся к выражению

$a < b \text{ or } c < d \text{ and } e < f$

Предположим, что метки перехода в случае истинности или ложности всего выражения — Ltrue и Lfalse соответственно. Тогда, используя определение, представленное на рис. 8.24, мы получим следующий код.

```
if a < b then goto Ltrue
goto L1
L1: if c < d goto L2
    goto Lfalse
L2: if e < f goto Ltrue
    goto Lfalse
```

Заметим, что сгенерированный таким образом код не оптимален — вторая инструкция может быть удалена без какого-либо изменения результата вычислений. Такие инструкции удаляются путем простейшей оптимизации кода (об этом будет рассказано в главе 9, “Генерация кода”). Другой подход создания кода без излишних переходов состоит в преобразовании выражения отношения вида $\text{id}_1 < \text{id}_2$ в инструкцию $\text{if } \text{id}_1 \geq \text{id}_2 \text{ goto Efalse}$ в соответствии с тем, что при истинности этого отношения мы прекращаем вычисление. \square

Пример 8.5

Рассмотрим исходный текст

```
while a < b do
    if c < d then
        x := y + z
    else
        x := y - z
```

Синтаксически управляемое определение, приведенное выше, вместе со схемами для инструкций присвоения и логических выражений приведут к следующему коду.

```
L1: if a < b goto L2
    goto Lnext
L2: if c < d goto L3
    goto L4
L3: t1 := y + z
    x := t1
    goto L1
L4: t2 := y - z
    x := t2
    goto L1
Lnext:
```

Заметим, что первых двух `goto` можно избежать, изменив условное выражение на обратное. Такой тип локальных преобразований может быть выполнен при оптимизации в процессе генерации кода (подробнее об этом будет сказано в главе 9, “Генерация кода”). \square

Смешанные логические выражения

Важно понимать, что в нашем рассмотрении мы предельно упростили грамматику логических выражений. На практике логические выражения часто содержат арифметические подвыражения, например $(a+b) < c$. В языках, где “ложь” представлена значением 0, а “истина” —

1, выражение типа $(a < b) + (b < a)$ может рассматриваться как арифметическое, дающее значение 0, если a и b имеют одинаковые значения, и 1 — в противном случае.

Метод представления логических выражений кодом с использованием переходов применим, даже если арифметические выражения представлены кодом для вычисления их значений. Рассмотрим, например, типичную грамматику

$$E \rightarrow E + E \mid E \text{ and } E \mid E \text{ relop } E \mid \text{id}$$

Мы можем считать, что $E + E$ дает целый арифметический результат (включение действительных или других арифметических типов лишь усложнит рассмотрение, не добавив ничего к его сути), а $E \text{ and } E$ и $E \text{ relop } E$ — логические значения, представляемые потоком управления. Выражение $E \text{ and } E$ требует, чтобы оба его аргумента были логическими, однако операции $+$ и relop могут принимать аргументы любого типа, включая смешанные аргументы. Идентификатор id также считается арифметическим, хотя наш пример легко расширить, допустив логические идентификаторы.

Для генерации кода в этом случае воспользуемся синтезируемым атрибутом $E.type$, который может принимать значения $arith$ или $bool$ в зависимости от типа E . Нетерминал E имеет также наследуемые атрибуты $true$ и $false$ для логических выражений и синтезируемый атрибут $place$ для арифметических выражений. Часть семантического правила для $E \rightarrow E_1 + E_2$ показана на рис. 8.25.

```
E.type := arith;
if E1.type = arith and E2.type = arith then begin
    /* Обычное арифметическое сложение */
    E.place = newtemp;
    E.code := E1.code || E2.code ||
        gen(E.place ':=' E1.place '+' E2.place)
end
else if E1.type = arith and E2.type = bool then begin
    E.place := newtemp;
    E2.true := newlabel;
    E2.false := newlabel;
    E.code := E1.code || E2.code ||
        gen(E2.true ':' E.place ':=' E1.place + 1) ||
        gen('goto' nextstat + 1) ||
        gen(E2.false ':' E.place ':=' E1.place)
end
else if ...
```

Рис. 8.25. Семантическое правило для продукции $E \rightarrow E_1 + E_2$

В случае смешанного представления выражений мы генерируем код для E_1 , затем E_2 , за которым следует три инструкции.

```
E2.true: E.place := E1.place + 1
            goto nextstat + 1
E2.false: E.place := E1.place
```

Первая инструкция вычисляет значение E как $E_1 + 1$, если E_2 истинно, третья — значение E , равное E_1 , если E_2 ложно. Вторая инструкция представляет собой переход через третью. Семантические правила для остальных случаев и других производств схожи с данным, и мы оставляем их читателю в качестве упражнения.

8.5. Инструкции case

Во многих языках имеются инструкции “switch” или “case” (даже вычислимый оператор GOTO в Fortran можно рассматривать как вариант такой инструкции). Рассматриваемый нами синтаксис показан на рис. 8.26.

```
switch выражение
begin
    case значение: инструкция
    case значение: инструкция
    .
    .
    case значение: инструкция
    default:      инструкция
end
```

Рис. 8.26. Синтаксис инструкции *switch*

В инструкции имеется вычисляемое выражение, за ним следует *n* константных значений, которые может принимать это выражение, и, возможно, “значение” default, с которым совпадает значение выражения, если не подошло ни одно из указанных. Предполагаемая трансляция данной инструкции представляет собой код, который выполняет следующее.

1. Вычисляет выражение.
2. Находит значение из списка, которое совпадает с полученным значением выражения (напомним, что если ни одно из значений не совпало со значением выражения, принимается “значение” по умолчанию default).
3. Выполняет инструкцию, связанную с найденным значением.

Шаг (2) является разветвлением на *n* путей, которое может быть реализовано несколькими способами. Если число вариантов не очень велико, скажем не превышает 10, вполне можно использовать последовательность условных переходов, каждый из которых выполняет проверку на соответствие конкретному значению и передает управление коду соответствующих инструкций.

Более компактный способ реализации такой последовательности условных переходов состоит в том, что создается таблица пар, состоящих из значения и метки кода соответствующей инструкции. Генерируется код для размещения в конце таблицы значения самого выражения и метки для инструкции по умолчанию. Далее простой цикл, генерируемый компилятором, проходит по таблице, сравнивая вычисленное значение со значениями из таблицы (причем гарантированно находит соответствие, поскольку последним значением в таблице является значение самого выражения).

Если число значений превышает 10 или около того, более эффективным способом является построение хеш-таблицы (см. раздел 7.6) значений с метками для разных инструкций в качестве записей этой таблицы. Если для вычисленного значения соответствующая запись в таблице не находится, генерируется переход к инструкции по умолчанию.

Существует специальный случай с еще более эффективной реализацией. Если все значения находятся в небольшом диапазоне, скажем от i_{min} до i_{max} , и число различных значений составляет значительную часть от $i_{max} - i_{min}$, то мы можем построить массив ме-

ток, с меткой для значения j , находящейся в записи таблицы со смещением $j - i_{min}$ от ее начала (все неиспользованные места в таблице заполняются метками кода по умолчанию). Тогда выполнение инструкции `switch` сводится к вычислению значения j , проверке допустимости $i_{min} \leq j \leq i_{max}$ и косвенному переходу по метке из записи таблицы со смещением $j - i_{min}$. Например, если выражение имеет символьный тип, может быть создана таблица из 128 записей (или иного размера, зависящего от используемого набора символов), по которой осуществляется переход без дополнительных проверок на соответствие диапазону.

Синтаксически управляемая трансляция инструкции `case`

Рассмотрим следующую инструкцию `switch`.

```
switch E
  begin
    case V1 : S1
    case V2 : S2
    .
    .
    .
    case Vn-1: Sn-1
    default : Sn
  end
```

При использовании схемы синтаксически управляемой трансляции удобно транслировать эту конструкцию в промежуточный код, представленный на рис. 8.27.

	Код вычисления E в t
	<code>goto test</code>
L ₁ :	Код для S_1
	<code>goto next</code>
L ₂ :	Код для S_2
	<code>goto next</code>
.	.
L _{n-1} :	Код для S_{n-1}
	<code>goto next</code>
L _n :	Код для S_n
	<code>goto next</code>
test:	<code>if t = V₁ goto L₁</code>
	<code>if t = V₂ goto L₂</code>
.	.
	<code>if t = V_{n-1} goto L_{n-1}</code>
	<code>goto L_n</code>
next:	

Рис. 8.27. Трансляция инструкции `case`

Все проверки оказываются в конце, так что генератор кода может распознать многопутевое разветвление программы и сгенерировать для него эффективный код, основанный на одной из описанных в начале этого раздела технологий. Если мы создадим более простой промежуточный код, показанный на рис. 8.28, компилятор должен выполнить обширный анализ для поиска наиболее эффективной реализации. Заметим, что размещение кода ветвления в начале неудобно, поскольку тогда компилятор не может строить код для каждого S_i в момент его появления.

```

Код для вычисления  $E$  в  $t$ 
if  $t \neq V_1$  goto  $L_1$ 
Код для  $S_1$ 
goto next
 $L_1:$  if  $t \neq V_2$  goto  $L_2$ 
Код для  $S_1$ 
goto next
 $L_2:$ 
    . . .
 $L_{n-2}:$  if  $t \neq V_{n-1}$  goto  $L_{n-1}$ 
Код для  $S_{n-1}$ 
goto next
 $L_{n-1}:$  Код для  $S_n$ 
next:

```

Рис. 8.28. Вариант трансляции инструкции case

Для трансляции (рис. 8.27) ключевого слова **switch** мы создаем две новые метки **test** и **next** и новую временную переменную t . Затем, по мере разбора выражения E , генерируется код для вычисления E в t . После обработки E мы генерируем переход **goto test**.

Затем, когда появляется ключевое слово **case**, мы создаем новую метку L_i и вносим ее в таблицу символов. В стек, используемый исключительно для хранения информации о значениях **case**, мы помещаем указатель на эту запись в таблице символов и значение V_i константы при **case**. (Если данная инструкция **switch** является вложенной в одну из инструкций, внутреннюю по отношению к другой инструкции **switch**, мы помещаем в стек маркер, разделяющий информацию, относящуюся к внешней и внутренней инструкциям **switch**.)

Мы обрабатываем каждую инструкцию **case** $V_i: S_i$ путем генерации новой метки L_i , за которой следует код для S_i с последующим переходом **goto next**. После того как появится завершающее конструкцию ключевое слово **end**, мы готовы сгенерировать код ветвления. Считывая пары указатель-значение из стека от его дна к вершине, мы можем генерировать последовательность трехадресных инструкций вида

```

case    $V_1$     $L_1$ 
case    $V_2$     $L_2$ 

case    $V_{n-1}$   $L_{n-1}$ 
case    $t$       $L_n$ 
Label next

```

где t — имя, хранящее значение селектора E , а L_n — метка инструкции по умолчанию. Трехадресная инструкция **case** $V_i L_i$ является синонимом **if** $t = V_i$ **goto** L_i (рис. 8.27), однако использование **case** облегчает генератору целевого кода задачу нахождения потенциальных кандидатов для специальной обработки. На стадии генерации кода последовательность инструкций **case** может быть транслирована в n -путевое ветвление наиболее эффективного вида, в зависимости от количества значений и их размещения в небольшом диапазоне.

8.6. Технология обратных поправок

Простейший способ реализации синтаксически управляемого определения из раздела 8.4 состоит в использовании двух проходов. Вначале мы строим синтаксическое дерево для входного потока, а затем обходим это дерево вглубь, осуществляя задаваемую определением трансляцию. Основная проблема при однопроходной генерации кода для логических выражений и инструкций потока управления состоит в том, что при одном проходе в момент создания инструкции перехода мы можем не знать метки, к которой должно перейти управление. Эту проблему можно решить путем генерации инструкций ветвления с временно неопределенными переходами. Каждая такая инструкция помещается в список инструкций, целевые метки которых будут указаны, когда они окажутся определены в процессе трансляции. Такое последовательное заполнение инструкций метками называется *технологией обратных поправок* (backpatching).

В этом разделе мы покажем, как использовать данную технологию для однопроходной генерации кода логических выражений и инструкций потока управления. Мы будем применять описанную в разделе 8.4 трансляцию, но с другим способом генерации меток. Для определенности будем генерировать массив четверок. Метки будут представлять собой индексы в этом массиве. Для работы со списками меток будем использовать три функции.

1. *makelist(i)* создает новый список, содержащий только i (индекс в массиве четверок); *makelist* возвращает указатель на созданный список.
2. *merge(p₁, p₂)* соединяет списки, на которые указывают p_1 и p_2 , и возвращает указатель на объединенный список.
3. *backpatch(p, i)* вставляет i в качестве целевой метки для каждой из инструкций в списке, на который указывает p .

Логические выражения

Теперь мы построим схему трансляции, предназначенную для генерации четверок для логических выражений при восходящем разборе. Для выполнения семантических действий по получению индекса генерируемой четверки воспользуемся маркером-нетерминалом M . Используемая нами грамматика выглядит следующим образом.

(1)	$E \rightarrow E_1 \text{ or } M E_2$
(2)	$ E_1 \text{ and } M E_2$
(3)	$ \text{not } E_1$
(4)	$ (E_1)$
(5)	$ \text{id}_1 \text{ relop id}_2$
(6)	$ \text{true}$
(7)	$ \text{false}$
(8)	$M \rightarrow \epsilon$

При генерации кода переходов для логических выражений используются синтезируемые атрибуты *truelist* и *falselist* нетерминала E . При генерации кода для E переходы по условию истинности и ложности остаются незавершенными, с незаполненными полями меток. Эти незавершенные переходы помещаются в списки, на которые указывают соответственно $E.\text{truelist}$ и $E.\text{falselist}$.

Семантические действия отражают приведенные выше рассуждения. Рассмотрим продукцию $E \rightarrow E_1 \text{ and } M E_2$. Если E_1 ложно, то ложно и E , так что инструкции в $E_1.\text{falselist}$ становятся частью $E.\text{falselist}$. Если E_1 истинно, мы должны приступить к проверке E_2 , так что целевой меткой для инструкций $E_1.\text{truelist}$ должно быть начало кода, сгенерированного для E_2 (что можно отследить с помощью маркера M). Атрибут $M.\text{quad}$ записывает номер первой инструкции $E_2.\text{code}$ с помощью связанного с продукцией $M \rightarrow \epsilon$ семантического действия $\{ M.\text{quad} := \text{nextquad} \}$.

Переменная nextquad хранит индекс очередной четверки. Когда будет просмотрена оставшаяся часть продукции $E \rightarrow E_1 \text{ and } M E_2$, это значение может быть внесено в качестве целевой метки в список $E_1.\text{truelist}$. Полностью схема трансляции выглядит следующим образом.

- | | |
|--|---|
| (1) $E \rightarrow E_1 \text{ or } M E_2$ | $\{ \text{backpatch}(E_1.\text{falselist}, M.\text{quad});$
$E.\text{truelist} := \text{merge}(E_1.\text{truelist}, E_2.\text{truelist});$
$E.\text{falselist} := E_2.\text{falselist} \}$ |
| (2) $E \rightarrow E_1 \text{ and } M E_2$ | $\{ \text{backpatch}(E_1.\text{truelist}, M.\text{quad});$
$E.\text{truelist} := E_2.\text{truelist};$
$E.\text{falselist} := \text{merge}(E_1.\text{falselist}, E_2.\text{falselist}) \}$ |
| (3) $E \rightarrow \text{not } E_1$ | $\{ E.\text{truelist} := E_1.\text{falselist};$
$E.\text{falselist} := E_1.\text{truelist} \}$ |
| (4) $E \rightarrow (E_1)$ | $\{ E.\text{truelist} := E_1.\text{truelist};$
$E.\text{falselist} := E_1.\text{falselist} \}$ |
| (5) $E \rightarrow \text{id}_1 \text{ relop } \text{id}_2$ | $\{ E.\text{truelist} := \text{makelist}(\text{nextquad});$
$E.\text{falselist} := \text{makelist}(\text{nextquad} + 1);$
$\text{emit}('if' \text{ id}_1.\text{place} \text{ relop } \text{id}_2.\text{place} 'goto_');$
$\text{emit}('goto_') \}$ |
| (6) $E \rightarrow \text{true}$ | $\{ E.\text{truelist} := \text{makelist}(\text{nextquad});$
$\text{emit}('goto_') \}$ |
| (7) $E \rightarrow \text{false}$ | $\{ E.\text{falselist} := \text{makelist}(\text{nextquad});$
$\text{emit}('goto_') \}$ |
| (8) $M \rightarrow \epsilon$ | $\{ M.\text{quad} := \text{nextquad} \}$ |

Для простоты семантическое действие (5) генерирует две инструкции — условный и безусловный переход; обе остаются незавершенными. Индекс первой генерируемой инструкции вносится в создаваемый список, указатель на который присваивается $E.\text{truelist}$. Вторая генерируемая инструкция — goto_- — также вносится в новый список, указатель на который присваивается $E.\text{falselist}$.

Пример 8.6

Вернемся еще раз к выражению $a < b \text{ or } c < d \text{ and } e < f$. Аннотированное дерево разбора для этого выражения показано на рис. 8.29. Действия выполняются в процессе обхода дерева вглубь. Поскольку все действия расположены в конце правых частей, они могут быть выполнены при осуществлении сверток в процессе восходящего разбора. При свертке $a < b$ в E в соответствии с продукцией (5) генерируются две четверки.

```
100:    if a < b goto _
101:    goto _
```

(Мы вновь начинаем нумерацию инструкций со 100.) Маркер M в продукции $E \rightarrow E_1 \text{ or } M E_2$ сохраняет значение $nextquad$, которое в настоящий момент равно 102. Свертка $c < d$ в E в соответствии с продукцией (5) приводит к генерации следующих четверок.

```
102: if c < d goto _
103: goto _
```

Мы рассмотрели E_1 в продукции $E \rightarrow E_1 \text{ and } M E_2$. Маркер в этой продукции сохраняет текущее значение $nextquad$, равное 104. Свертка $e < f$ в соответствии с (5) генерирует следующие четверки.

```
104: if e < f goto _
105: goto _
```

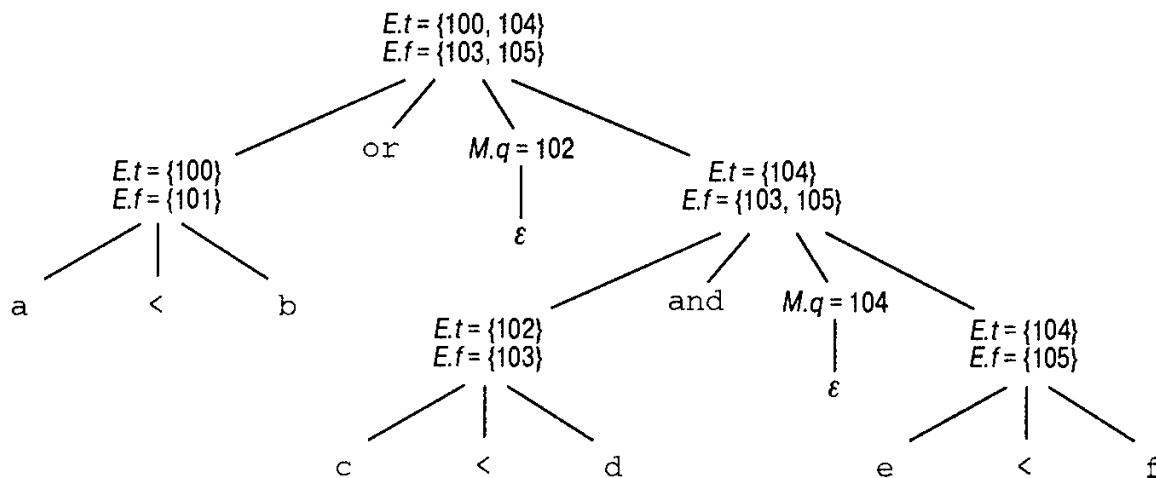


Рис. 8.29. Аннотированное дерево разбора для $a < b \text{ or } c < d \text{ and } e < f$

Теперь мы выполняем свертку по продукции $E \rightarrow E_1 \text{ and } M E_2$. Соответствующее семантическое действие вызывает $backpatch(\{102\}, 104)$, где аргумент $\{102\}$ означает указатель на список, содержащий только 102, — список, на который указывает $E_1.trueList$. Этот вызов $backpatch$ заполняет поле инструкции 102 меткой 104. Теперь шесть сгенерированных инструкций выглядят следующим образом:

```
100: if a < b goto _
101: goto _
102: if c < d goto 104
103: goto _
104: if e < f goto _
105: goto _
```

Семантическое действие, связанное с последней сверткой $E \rightarrow E_1 \text{ or } M E_2$, вызывает $backpatch(\{101\}, 102)$, после чего инструкции приобретают следующий вид.

```
100: if a < b goto _
101: goto 102
102: if c < d goto 104
103: goto _
104: if e < f goto _
105: goto _
```

Все выражение истинно тогда и только тогда, когда достигаются инструкции перехода 100 или 104, и ложно — когда достигаются инструкции `goto` 103 или 105. Целевые метки этих инструкций будут заполнены позже в процессе компиляции, когда станет известно, что необходимо выполнить при истинности или ложности выражения. \square

Инструкции потока управления

Теперь мы покажем, каким образом можно использовать технологию обратных поправок для однопроходной трансляции инструкций потока управления. Как и ранее, мы остановимся на генерации четверок и используем те же имена полей и процедуры работы со списками, что и в предыдущем подразделе. В качестве примера разработаем схему трансляции для инструкций, порождаемых следующей грамматикой.

- (1) $S \rightarrow \text{if } E \text{ then } S$
- (2) | $\text{if } E \text{ then } S \text{ else } S$
- (3) | $\text{while } E \text{ do } S$
- (4) | $\text{begin } L \text{ end}$
- (5) | A
- (6) $L \rightarrow L ; S$
- (7) | S

Здесь S означает инструкцию, L — список инструкций, A — инструкцию присвоения, а E — логическое выражение. Заметим, что в грамматике должны присутствовать и другие продукции, например, для инструкции присвоения. Однако имеющихся продуктов вполне достаточно для иллюстрации технологий, используемых при трансляции инструкций потока управления.

Мы используем ту же структуру кода для инструкций `if-then`, `if-then-else` и `while-do`, что и в разделе 8.4. Неявно предполагается, что код, следующий при выполнении за данной инструкцией, физически следует за ней и в массиве четверок. В противном случае должен использоваться явный оператор перехода.

Наш подход заключается в заполнении операторов перехода по мере определения целевых меток. При этом, кроме двух списков переходов, используемых для истинных и ложных значений логических выражений, для инструкций потребуются списки переходов (определяемые атрибутом `nextlist`) к коду, который следует за ними в последовательности выполнения.

Схема реализации трансляции

Сейчас мы ознакомимся со схемой синтаксически управляемой трансляции для указанных выше инструкций потока управления. Нетерминал E , как и ранее, имеет два атрибута — $E.trueList$ и $E.falseList$. И L , и S также необходимы списки незавершенных четверок, которые в итоге будут заполнены с помощью технологии обратных поправок. На эти списки указывают атрибуты $L.nextList$ и $S.nextList$. Атрибут $S.nextList$ представляет собой указатель на список всех условных и безусловных переходов к четверке, следующей за S в порядке выполнения ($L.nextList$ определяется аналогично).

В схеме кода для $S \rightarrow \text{while } E \text{ do } S_1$ (рис. 8.22б) имеются метки $S.begin$ и $E.true$, маркирующие начало кода инструкции S в целом и тела S_1 . Два маркера, используемые в следующей продукции, обеспечивают запись номеров четверок в этих положениях:

$$S \rightarrow \text{while } M_1 E \text{ do } M_2 S_1$$

Как и ранее, продукция для M представляет собой $M \rightarrow \epsilon$ с действием, устанавливающим значение атрибута $M.quad$ равным номеру следующей четверки. После выполнения тела S_1 управление передается к началу инструкции. Таким образом, при свертке $\text{while } M_1 E \text{ do } M_2 S_1$ в S мы выполняем обратные поправки к списку $S_1.nextlist$, устанавливая все целевые метки в этом списке равными $M_1.quad$. После кода S_1 добавляется явный переход к началу кода E , поскольку управление может полностью пройти по коду S_1 до конца. Все целевые метки в $E.trueclist$ устанавливаются равными $M_2.quad$ для перехода к началу кода S_1 .

Более серьезный аргумент в пользу использования $S.nextlist$ и $L.nextlist$ появляется при генерации кода для условной инструкции $\text{if } E \text{ then } S_1 \text{ else } S_2$. Если управление доходит до конца S_1 (например, если S_1 — присвоение), то в конце кода для S_1 мы должны включить переход через код S_2 . Для этого воспользуемся еще одним маркером N с продукцией $N \rightarrow \epsilon$. N имеет атрибут $N.nextlist$, который будет представлять собой список, содержащий номер четверки с инструкцией `goto_`, генерируемой семантическим правилом, связанным с N . Далее мы приведем семантические правила исправленной грамматики.

(1) $S \rightarrow \text{if } E \text{ then } M_1 S_1 N \text{ else } M_2 S_2$

```
{ backpatch(E.trueclist, M1.quad);
  backpatch(E.falselist, M2.quad);
  S.nextlist := merge(S1.nextlist, merge(N.nextlist, S2.nextlist)) }
```

При истинности E мы заполняем соответствующие переходы меткой $M_1.quad$, которая представляет собой начало кода S_1 . Аналогично переходы при ложном E заполняются меткой начала кода S_2 . Список $S.nextlist$ включает все переходы из S_1 и S_2 , а также переход, генерируемый маркером N .

(2) $N \rightarrow \epsilon$

```
{ N.nextlist := makelist(nextquad);
  emit('goto_') }
```

(3) $M \rightarrow \epsilon$

```
{ M.quad := nextquad }
```

(4) $S \rightarrow \text{if } E \text{ then } M S_1$

```
{ backpatch(E.trueclist, M.quad);
```

```
  S.nextlist := merge(E.falselist, S1.nextlist) }
```

(5) $S \rightarrow \text{while } M_1 E \text{ do } M_2 S_1$

```
{ backpatch(S1.nextlist, M1.quad);
```

```
  backpatch(E.trueclist, M2.quad);
```

```
  S.nextlist := E.falselist;
```

```
  emit('goto' M1.quad) }
```

(6) $S \rightarrow \text{begin } L \text{ end}$

```
{ S.nextlist := L.nextlist }
```

(7) $S \rightarrow A$

```
{ S.nextlist := nil }
```

Присвоение $S.nextlist := \text{nil}$ инициализирует $S.nextlist$ пустым списком.

(8) $L \rightarrow L_1 ; M S$

```
{ backpatch(L1.nextlist, M.quad);
  L.nextlist := S.nextlist }
```

Инструкция, следующая в порядке выполнения за L_1 , является началом S . Таким образом, список $L_1.nextlist$ заполняется меткой начала кода S , которая представляет собой не что иное, как $M.quad$.

(9) $L \rightarrow S$

```
{ L.nextlist := S.nextlist }
```

Заметим, что в семантических правилах нигде, кроме (2) и (5), не создаются новые четверки. Весь остальной код генерируется семантическими действиями, связанными с инструкциями присвоения и выражениями. Задача описанной схемы — правильное внесение обратных поправок с тем, чтобы поток управления корректно связывал присвоения и вычисления логических выражений в единую программу.

Метки и безусловные переходы

Наиболее простыми конструкциями языков программирования для изменения потока управления в программе являются метки и безусловные переходы. Когда компилятор встречает инструкцию вида `goto L`, он должен проверить, что в области видимости этого перехода находится в точности одна инструкция с меткой `L`. Если эта метка уже встречалась компилятору — либо в объявлении метки, либо как метка некоторой инструкции исходного кода, — то таблица символов содержит запись с сгенерированной компилятором меткой, указывающей первую трехадресную инструкцию кода, соответствующего исходной инструкции с меткой `L`. При трансляции мы генерируем трехадресный код `goto` с сгенерированной компилятором меткой в качестве целевой.

Когда метка `L` встречается в исходной программе впервые, в объявлении или в качестве целевой метки инструкции `goto`, мы вносим `L` в таблицу символов и генерируем для нее символьную метку.

8.7. Вызовы процедур

Процедуры¹¹ представляют собой настолько важные и часто используемые программные конструкции, что компилятор просто обязан сгенерировать эффективный код вызова процедуры и возврата из нее. Подпрограммы времени исполнения, обеспечивающие передачу аргументов, вызовы и возвраты, являются частью пакета поддержки времени исполнения. Различные типы механизмов, необходимых для реализации пакета поддержки времени исполнения, обсуждались в главе 7, “Среды времени исполнения”. В этом разделе мы рассмотрим типичный код, генерируемый для вызовов процедур и возвратов из них.

Рассмотрим грамматику простейшей инструкции вызова процедуры.

- (1) $S \rightarrow \text{call id} (\text{Elist})$
- (2) $\text{Elist} \rightarrow \text{Elist}, E$
- (3) $\text{Elist} \rightarrow E$

Вызывающие последовательности

Как говорилось в главе 7, “Среды времени исполнения”, трансляция вызова включаетзывающую последовательность (или последовательность вызова), которая состоит из действий, выполняемых при входе в каждую процедуру и выходе из нее. Посколькузывающие последовательности различны даже в реализациях одного и того же языка, опишем типичные действия, выполняемые этими последовательностями.

При вызове процедуры выделяется память для записи активации данной процедуры. Вычисляются и делаются доступными для вызываемой процедуры аргументы в ее

¹¹ Термин “процедура” в данном случае включает функции (рассматриваемые как процедуры с возвращаемым значением).

вызове (путем их размещения в заранее известном месте). Указатели окружения устанавливаются таким образом, чтобы обеспечить вызываемой процедуре возможность доступа к данным в охватывающих блоках. Состояние вызывающей процедуры сохраняется с тем, чтобы быть восстановленным после возвращения из вызываемой процедуры. Кроме того, в определенном месте сохраняется адрес возврата, по которому передается управление при возврате из процедуры (обычно это адрес инструкции, следующей за вызовом процедуры). И наконец, генерируется переход к началу кода вызываемой процедуры.

При возврате из процедуры также выполняется ряд действий. Если вызываемая процедура представляет собой функцию, ее результат сохраняется в определенном месте, чтобы им могла воспользоваться вызывающая процедура. Восстанавливается запись активации вызывающей процедуры и генерируется переход по сохраненному адресу возврата в вызывающую процедуру.

Не существует точного разделения задач времени исполнения между вызываемой и вызывающей процедурами. Зачастую исходный язык, целевая машина и операционная система налагают свои требования, приводящие к тому или иному решению поставленной задачи.

Простой пример

Рассмотрим простой пример, в котором параметры передаются по ссылке, а память выделяется статически. В этом случае мы можем использовать для хранения аргументов инструкции `param` как таковые. Вызываемой процедуре передается в регистре указатель на первую инструкцию `param`, после чего процедура может получить указатель на любой из своих параметров путем соответствующего смещения относительно полученного базового указателя. При генерации трехадресного кода для такого типа вызова следует создать трехадресные инструкции, необходимые для вычисления тех аргументов, которые представляют собой выражения, а не простые имена, а затем разместить список трехадресных инструкций `param` по одной для каждого аргумента. Если мы не хотим смешивать инструкции вычисления аргументов с инструкциями `param`, придется хранить значения `E.place` для каждого выражения `E` в `id(E, E, ..., E)`¹².

Для хранения этих значений удобно использовать очередь, т.е. список “первым вошел — первым вышел”. Наша семантическая подпрограмма для $Elist \rightarrow Elist$, E будет включать сохранение `E.place` в очереди `queue`. Затем семантическая подпрограмма для $S \rightarrow \text{call id} (Elist)$ сгенерирует инструкции `param` для каждого элемента очереди, размещая их после инструкций вычисления этих аргументов. Инструкции вычисления аргументов генерируются при свертках выражений для аргументов к E . Следующая синтаксически управляемая трансляция объединяет изложенные здесь идеи.

(1) $S \rightarrow \text{call id} (Elist)$ { for каждый элемент p из `queue` do `emit('param' p); emit('call' id.place)` }

Код для S представляет собой код для $Elist$, вычисляющий аргументы, за которым следуют инструкции `param p` для каждого аргумента, а после них — инструкция `call`.

¹² Если параметры передаются вызываемой процедуре путем размещения в стеке (что обычно делается для динамически распределенных данных), нет причин избегать смешивания вычислений и инструкций `param`. В процессе генерации кода последние будут заменены на код, размещающий параметры в стеке.

Подсчет числа параметров не генерируется, но при необходимости это может быть вычислено так же, как и $E.ndim$ в предыдущем разделе.

- (2) $Elist \rightarrow Elist, E$ { Добавляем $E.place$ в конец $queue$ }
(3) $Elist \rightarrow E$ { Инициализируем $queue$ значением $E.place$ }

При выполнении (3) мы получаем очередь $queue$ с единственным указателем на запись в таблице символов для имени, описывающего значение E .

Упражнения

- 8.1. Транслируйте арифметическое выражение $a^* - (b+c)$ в
- синтаксическое дерево;
 - постфиксную запись;
 - трехадресный код.
- 8.2. Транслируйте выражение $- (a+b)^* (c+d) + (a+b+c)$ в
- четверки;
 - тройки;
 - косвенные тройки.
- 8.3. Транслируйте выполняемые инструкции из приведенной ниже программы на языке С в
- синтаксическое дерево;
 - постфиксную запись;
 - трехадресный код.
- ```
main()
{
 int i;
 int a[10];
 i = 0;
 while (i < 10) {
 a[i] = 0; i = i + 1;
 }
}
```
- \*8.4. Докажите, что если все операторы бинарные, то строка операторов и operandов является постфиксным выражением тогда и только тогда, когда (1) операторов в точности на один меньше, чем operandов, и (2) каждый непустой префикс выражения содержит меньше операторов, чем operandов.
- 8.5. Измените схему трансляции (рис. 8.11) для вычисления типов и относительных адресов объявленных имен так, чтобы она позволяла использовать в объявлениях типа  $D \rightarrow id : T$  списки имен вместо одиночных имен.
- 8.6. Префиксная запись выражения, в котором оператор  $\theta$  применяется к выражениям  $e_1, e_2, \dots, e_k$ , имеет вид  $\theta p_1 p_2 \dots p_k$ , где  $p_i$  — префиксная запись  $e_i$ .
- Найдите префиксную запись для  $a^* - (b+c)$ .

- \*\*b) Покажите, что инфиксные выражения не могут быть транслированы в префиксную запись с помощью схемы трансляции, в которой все действия заключаются в выводе информации и находятся в конце правых частей продукции.
- c) Приведите синтаксически управляемое определение для трансляции инфиксных выражений в префиксную запись. Какие из методов главы 5, “Синтаксически управляемая трансляция”, вы можете при этом использовать?

8.7. Напишите программу реализации синтаксически управляемого определения, приведенного на рис. 8.24, для трансляции логических выражений в трехадресный код.

8.8. Измените приведенное на рис. 8.24 синтаксически управляемое определение, чтобы генерировать код для стековой машины из раздела 2.8.

8.9. Синтаксически управляемое определение на рис. 8.24 транслирует  $E \rightarrow \text{id}_1 < \text{id}_2$  в следующую пару инструкций.

```
if id1 < id2 goto ...
goto ...
```

Вместо этого мы можем выполнить трансляцию в одну инструкцию

```
if id1 ≥ id2 goto _
```

и перейти к следующей инструкции при истинности  $E$ . Измените указанное определение для генерации кода такого типа.

8.10. Напишите программу реализации синтаксически управляемого определения для инструкций потока управления, приведенного на рис. 8.23.

8.11. Напишите программу реализации алгоритма обратных поправок, описанного в разделе 8.6.

8.12. Транслируйте следующее присвоение в трехадресный код с использованием схемы трансляции из раздела 8.3.

```
A[i, j] := B[i, j] + C[A[k, l]] + D[i+j]
```

\*8.13. Некоторые языки типа PL/I позволяют списку имен получать список атрибутов, а также позволяют объявлениям быть вложенными одно в другое. Следующая грамматика абстрагирует данную проблему.

$$\begin{array}{lcl}
 D & \rightarrow & \text{namelist attrlist} \\
 & | & ( D ) \text{ attrlist} \\
 \text{namelist} & \rightarrow & \text{id}, \text{namelist} \\
 & | & \text{id} \\
 \text{attrlist} & \rightarrow & A \text{ attrlist} \\
 & | & A \\
 A & \rightarrow & \text{decimal} | \text{fixed} | \text{float} | \text{real}
 \end{array}$$

Смысл  $D \rightarrow ( D ) \text{ attrlist}$  в том, что все имена, упомянутые в объявлении в скобках, получают атрибуты из  $attrlist$ ; уровень вложенности при этом не имеет значения. Заметим, что объявление  $n$  имен и  $m$  атрибутов может привести к внесению в таблицу символов  $nm$  записей. Приведите синтаксически управляемое определение для объявлений, задаваемых этой грамматикой.

8.14. В С цикл **for** имеет следующий вид.

```
for (e1 ; e2 ; e3) stmt
```

Исходя из того, что приведенный цикл **for** эквивалентен следующему

```
e1;
while (e2) {
 stmt
 e3 ;
}
```

постройте синтаксически управляемое определение для трансляции конструкции **for** языка С в трехадресный код.

- 8.15. Стандарт Pascal определяет инструкцию

```
for v := initial to final do stmt
```

которая имеет тот же смысл, что и следующий код.

```
begin
 t1 := initial; t2 := final;
 if t1 ≤ t2 then begin
 v := t1;
 stmt
 while v ≠ t2 do begin
 v := succ(v);
 stmt
 end
 end
end
```

- a) Рассмотрим следующую программу на языке Pascal.

```
program forloop(input, output);
var i, initial, final: integer;
begin
 read(initial, final);
 for i := initial to final do
 writeln(i)
end.
```

Как поведет себя программа при  $initial = \text{MAXINT} - 5$  и  $final = \text{MAXINT}$ , где  $\text{MAXINT}$  — наибольшее целое число на целевой машине.

- \*b) Постройте синтаксически управляемое определение, которое генерирует корректный трехадресный код для конструкции **for** языка Pascal.

## Библиографические примечания

UNCOL (Universal Compiler Oriented Language, универсальный компиляторно-ориентированный язык) — легендарный универсальный промежуточный язык, поиски которого начались со средины 50-х годов. В работе [414] было показано, как можно создавать компиляторы, соединяя начальную стадию компиляции для исходного языка программирования и заключительную для данной целевой машины. Приведенные в работе технологии “раскрутки” часто используются при перенастройке компиляторов (см. раздел 11.2). Исходное описание UNCOL можно найти в работе [410].

Перенастраиваемые компиляторы состоят из одной программы начальной стадии компиляции, к которой могут подключаться программы различных заключительных стадий, реализующих данный язык программирования на различных целевых машинах. Одним из первых примеров языка с перенастраиваемым компилятором служит Neliac [202], написанный на собственном языке (см. также описание перенастраиваемых компиляторов BCPL [369], Pascal [335] и С [217]). Авторы [331] применили идею изменения заключительной стадии к макропроцессору, текстовому редактору и компилятору Basic.

Идеал UNCOL, состоящий в реализации *n* языков на *m* машинах написанием *n* начальных и *m* заключительных частей компиляторов вместо создания *n × m* различных компиляторов, может достигаться различными путями. Один из подходов состоит в размещении начальной стадии нового языка на базе существующего компилятора. Так, в [131] описано добавление начальной стадии компилятора Fortran 77 к существующим компиляторам С [217, 373]. Организация компиляторов, разработанных для объединения различных начальных и заключительных стадий, описана в работах [105, 284, 420].

В [105], чтобы подчеркнуть значение множества доступных операторов промежуточного представления, использованы термины “объединение” и “пересечение” абстрактных машин. Множество инструкций и режимов адресации машины-пересечения ограничено, и на начальной стадии нет особой свободы выбора при генерации промежуточного кода. Машины-объединения представляют альтернативные пути реализации конструкций исходного языка. Поскольку все эти альтернативы могут не реализовываться всеми целевыми машинами непосредственно, более богатый набор инструкций машины-объединения может приводить к зависимости от целевой машины. Аналогичные применения применимы и к другим видам промежуточного кода, таким как синтаксические деревья или трехадресный код. В [143] рассмотрены способы описания доступа к стеку времени исполнения с использованием машинно-независимых операций.

Подробные описания реализации конкретных языков можно найти в работах [172, 363] (Algol 60), [146] (PL/I), [456] (Pascal) и [62] (Algol 68).

В [162, 318] обсуждается генерация оптимального кода для логических выражений. Упр. 8.15 взято из [332].

# ГЛАВА 9

# Генерация кода

Последней стадией нашей модели компиляции является генератор кода, который получает на вход промежуточное представление исходной программы и выводит эквивалентную целевую программу, как показано на рис. 9.1. Технологии генерации целевого кода, представленные в данной главе, могут применяться независимо от того, имеется ли фаза оптимизации перед генерацией кода, как в некоторых оптимизирующих компиляторах. Эта фаза пытается преобразовать промежуточный код в такой вид, по которому может быть получен более эффективный целевой код. Об оптимизации кода мы детально поговорим в следующей главе.

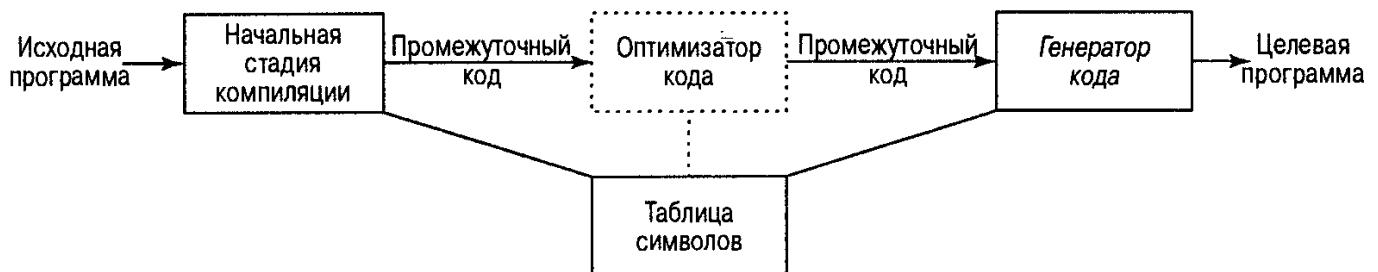


Рис. 9.1. Положение генератора кода в модели компилятора

Традиционно к генератору кода предъявляются жесткие требования. Получаемый код должен быть корректным и высококачественным, что означает эффективное использование ресурсов целевой машины. Кроме того, эффективно должен работать и сам генератор кода.

Математическая проблема генерации оптимального кода является неразрешимой. На практике мы вынуждены довольствоваться эвристическими технологиями, дающими хороший, но не обязательно оптимальный код. Выбор эвристики очень важен, так как тщательно разработанный алгоритм генерации кода может давать код, работающий в несколько раз быстрее кода, полученного с помощью недостаточно продуманного алгоритма.

## 9.1. Вопросы создания генератора кода

Хотя мелкие детали генератора кода зависят от целевой машины и операционной системы, такие вопросы, как управление памятью, выбор инструкций, распределение регистров и порядок вычислений, свойственны практически всем задачам, связанным с генерацией кода. В этом разделе мы рассмотрим общие вопросы конструирования генераторов кода.

### Вход генератора кода

Входной поток генератора кода представляет собой промежуточное представление исходной программы, полученное на начальной стадии компиляции, вместе с таблицей символов, которая используется для определения адресов времени исполнения объектов данных, обозначаемых в промежуточном представлении именами.

Как мы упоминали в предыдущей главе, имеется несколько видов промежуточных языков: линейное представление, например постфиксная запись, трехадресное представление, например четверки, виртуальное машинное представление, например код стековой машины, и графическое представление в виде синтаксических деревьев и дагов. Хотя алгоритмы в этой главе излагаются с использованием трехадресных кодов, деревьев и дагов, многие из этих технологий применимы и к другим промежуточным представлениям.

Мы считаем, что перед генерацией кода начальной стадией компиляции было выполнено сканирование, разбор и трансляция исходной программы в подробное промежуточное представление, так что значения имен в промежуточном языке могут быть представлены величинами, с которыми целевая машина может работать непосредственно (биты, целые и действительные числа, указатели и т.п.). Мы также полагаем, что были выполнены все необходимые проверки типов, так что необходимые операторы преобразования типов находятся на своих местах, и были выявлены все очевидные семантические ошибки (типа использования действительного числа в качестве индекса массива). Таким образом, стадия генерации кода может работать в предположении, что ее вход не содержит ошибок. В некоторых компиляторах семантические проверки такого типа выполняются одновременно с генерацией кода.

## Целевые программы

Выходом генератора кода является целевая программа. Подобно промежуточному коду, выход генератора кода может принимать различные виды: абсолютный машинный язык, перемещаемый машинный язык, или язык ассемблера.

Преимуществом генерации абсолютной программы на машинном языке является то, что такой код помещается в фиксированное место в памяти и немедленно выполняется; небольшие программы при этом быстро компилируются и выполняются. Такой абсолютный код генерирует ряд "студенческих" компиляторов типа WATFIV или PL/C.

Генерация перемещаемой программы на машинном языке (объектного модуля) обеспечивает возможность раздельной компиляции подпрограмм. Множество перемещаемых объектных модулей могут быть затем связаны в единое целое и загружены для выполнения специальной программой — связывающим загрузчиком. Дополнительные затраты на связывание и загрузку компенсируются возможностью раздельной компиляции подпрограмм и вызова других, ранее скомпилированных подпрограмм из объектных модулей. Если целевая машина не обрабатывает перемещение автоматически, компилятор должен предоставить загрузчику явную информацию о перемещении для связывания сегментов раздельно скомпилированных подпрограмм.

Получение в качестве выхода генератора кода программы на языке ассемблера несколько облегчает процесс генерации кода; в частности, мы можем создавать символьные инструкции и использовать возможности макросов ассемблера. Плата за эту простоту — дополнительный шаг обработки ассемблерной программы после генерации кода. Поскольку генерация ассемблерного кода не дублирует целиком задачу ассемблера, такой выбор оказывается одной из вполне разумных альтернатив, в особенности на машинах с небольшим количеством памяти, где компилятор вынужден использовать многоходовые технологии. В этой главе для большей удобочитаемости текста мы используем в качестве целевого языка ассемблерный код. Однако мы должны отметить, что поскольку адреса могут быть вычислены из смещений и другой информации, хранящейся в таблице символов, генератор кода может производить переносимые или абсолютные адреса для имен так же легко, как и символьные адреса.

## Управление памятью

Отображение имен исходной программы в адреса объектов данных в памяти во время работы программы выполняется совместно начальной стадией компилятора и генератором кода. В предыдущей главе мы полагали, что имя в трехадресной инструкции ссылается на соответствующую запись в таблице символов. В разделе 8.2 записи в таблице символов создавались при рассмотрении объявлений в процедурах. Тип, используемый в объявлении, определял размер, т.е. количество памяти, необходимой для объявленного имени. По информации из таблицы символов можно определить относительный адрес имени в области данных процедуры. В разделе 9.3 мы вкратце рассмотрим реализации статического и стекового распределений областей данных и покажем, как имена в промежуточном представлении могут быть преобразованы в адреса в целевом коде.

При генерации машинного кода метки в трехадресных инструкциях преобразуются в адреса инструкций. Этот процесс аналогичен технологии обратных поправок из раздела 8.6. Предположим, что метки представляют собой номера четверок в массиве. Так как мы сканируем четверки по очереди, можем вывести положение первой машинной инструкции, генерируемой данной четверкой, просто подсчитывая количество слов, использованных для уже сгенерированного кода. Это количество может храниться, например, в отдельном поле в массиве четверок, так что когда встречается ссылка типа  $j$ : `goto i` и  $i$  меньше текущего номера четверки  $j$ , мы можем просто сгенерировать инструкцию перехода с целевым адресом, равным расположению первой инструкции машинного кода, соответствующего четверке  $i$ . Если же переход осуществляется “вперед”, т.е.  $i$  превышает  $j$ , в списке для четверки  $i$  мы должны запомнить положение первой машинной инструкции, генерируемой для четверки  $j$ . Затем, при обработке четверки  $i$ , мы вносим корректный адрес во все инструкции, в которых использовались переходы в  $i$ .

## Выбор инструкций

Набор инструкций целевой машины определяет сложность их выбора. Важными факторами этого выбора являются единство и полнота множества инструкций. Если целевая машина не поддерживает все типы данных единственно, то каждое исключение из общего правила потребует специальной обработки.

Другими важными факторами являются скорость выполнения инструкций и идиомы языка целевой машины. Если мы не беспокоимся об эффективности целевой программы, выбор инструкций достаточно прост. Для каждого типа трехадресных инструкций мы можем разработать шаблон целевого кода, генерируемого для данной конструкции. Например, каждая трехадресная инструкция вида  $x := y + z$ , где  $x$ ,  $y$  и  $z$  распределяются статически, может быть транслирована в следующий код.

```
MOV y, R0 /* Загрузить y в регистр R0 */
ADD z, R0 /* Прибавить к R0 значение z */
MOV R0, x /* Сохранить значение R0 в x */
```

К сожалению, такая генерация кода (инструкция за инструкцией) часто приводит к плохому коду. Так, например, последовательность инструкций

```
a := b + c
d := a + e
```

будет транслирована в

```
MOV b, R0
ADD c, R0
MOV R0,a
MOV a, R0
ADD e, R0
MOV R0, d
```

Очевидно, что четвертая инструкция излишня (а если значение *a* в дальнейшем не используется, то излишня и третья).

Качество генерируемого кода определяется его размером и скоростью. Целевая машина с богатым набором инструкций может обеспечить несколько путей выполнения одной и той же операции. Поскольку различные реализации могут существенно отличаться по своей эффективности, непосредственная трансляция промежуточного кода может приводить к вполне корректному, но совершенно неприемлемому с точки зрения эффективности работы целевому коду. Например, если целевая машина имеет инструкцию инкремента INC, то трехадресный код *a := a + 1* можно реализовать более эффективно с помощью инструкции INC *a* по сравнению с более очевидной последовательностью — внести *a* в регистр, просуммировать с 1 и сохранить полученное значение в *a*.

```
MOV a, R0
ADD #1, R0
MOV R0, a
```

Знания о скорости выполнения тех или иных инструкций необходимы для создания хорошего, эффективного целевого кода, однако, к сожалению, точная информация об этом труднодоступна. Кроме того, принятие решения о способе реализации того или иного промежуточного кода зачастую требует учета контекста, в котором появляется та или иная инструкция (а также особенностей работы целевой машины — код, эффективный при работе на процессоре i80286, может оказаться не самым эффективным при работе на процессоре Pentium — и не только из-за более богатого набора инструкций. — *Прим. ред.*). Инструментарий для построения программ выбора инструкций будет рассмотрен в разделе 9.12.

## Распределение регистров

Инструкции, использующие в качестве operandов регистры, обычно короче и быстрее выполняются, чем инструкции, работающие с operandами, расположенными в памяти. Следовательно, эффективное использование регистров — еще одна важная составляющая генерации хорошего целевого кода. Использование регистров часто разделяется на две подзадачи.

1. В процессе *распределения регистров* мы выбираем множество переменных, которые будут находиться в регистрах в некоторой точке программы.
2. В последующей фазе *назначения регистров* мы выбираем конкретные регистры для размещения в них переменных.

Поиск оптимального назначения регистров переменным представляет собой сложную задачу, с точки зрения математики являющуюся NP-полной. Проблема усложняется еще и тем, что аппаратное обеспечение и/или операционная система могут накладывать дополнительные ограничения по использованию регистров.

Ряд машин требует использования *пар регистров* (четных и нечетных) для некоторых operandов и результатов. Например, в машине IBM System/370 целочисленное

умножение и целочисленное деление используют пары регистров. Инструкция умножения имеет вид

M x, y

где множимое x представляет собой четный регистр из пары соседних регистров, а множитель y — единичный регистр. Результат занимает пару соседних регистров целиком.

В инструкции деления вида

D x, y

64-битовое делимое располагается в паре регистров, четным из которых является x; y представляет делитель. После деления в четном регистре хранится остаток, а в нечетном — частное.

Рассмотрим теперь две трехадресные последовательности (рис. 9.2), отличающиеся только одним оператором во второй инструкции. Соответствующие им кратчайшие последовательности ассемблерного кода показаны на рис. 9.3.

|            |            |
|------------|------------|
| t := a + b | t := a + b |
| t := t * c | t := t + c |
| t := t / d | t := t / d |

Рис. 9.2. Две последовательности трехадресных инструкций

|          |             |
|----------|-------------|
| L R1, a  | L R0, a     |
| A R1, b  | A R0, b     |
| M R0, c  | A R0, c     |
| D R0, d  | SRDA R0, 32 |
| ST R1, t | D R0, d     |
|          | ST R1, t    |

Рис. 9.3. Оптимальные последовательности машинных кодов

Здесь  $R_i$  обозначает регистр  $i$ . (SRDA R0, 32 сдвигает делимое из R0 вправо на 32 бита в R1 и очищает R0 так, что все его биты равны биту знака.) L, ST и A означают соответственно инструкции загрузки, сохранения и сложения. Заметим, что оптимальный выбор регистра для загрузки a зависит от того, что в конечном счете произойдет с t. Стратегии распределения регистров рассматриваются в разделе 9.7.

## Выбор порядка вычислений

Порядок, в котором выполняются вычисления, также может существенно влиять на эффективность целевого кода. Как мы увидим, изменение порядка вычислений может привести к уменьшению количества необходимых для работы регистров. Выбор оптимального порядка вычислений — еще один пример NP-полной математической задачи. Вначале мы избежим этой задачи, генерируя целевой код для трехадресных инструкций в том порядке, в каком они были созданы генератором промежуточного кода.

## Подходы к генерации кода

Несомненно, самым важным критерием работы генератора кода является корректность производимого кода. Корректность приобретает особую значимость из-за огромного количества частных случаев и исключений из правил, с которыми приходится стал-

киваться генератору кода. С учетом приоритета корректности, важной целью разработки генератора кода является создание легко реализуемого, тестируемого и поддерживаемого генератора целевого кода.

В разделе 9.6 содержится простой алгоритм генерации кода, который использует информацию о последующем использовании операндов для генерации кода регистровой машины. Он рассматривает каждую инструкцию по очереди, сохраняя операнды в регистрах, насколько это оказывается возможным. Выход такого генератора может быть улучшен путем применения технологий локальной оптимизации, наподобие тех, которые будут рассмотрены в разделе 9.9.

В разделе 9.7 представлены технологии, оптимизирующие использование регистров путем рассмотрения потока управления в промежуточном коде. Особый упор делается на распределении регистров для интенсивно используемых операндов во внутренних циклах.

Разделы 9.10 и 9.11 представляют некоторые технологии выбора кода с использованием деревьев, упрощающие создание перенастраиваемых генераторов кода. Версии РСС — переносимого компилятора С — с такими генераторами кода распространились на самые разнообразные платформы, чему немало способствовала доступность операционной системы UNIX. В разделе 9.12 показано, как генерация кода может рассматриваться в качестве процесса построения дерева.

## 9.2. Целевая машина

Одним из непременных требований к построению хорошего генератора кода является близкое знакомство с целевой машиной и ее набором инструкций. К сожалению, при обсуждении общих вопросов генерации кода невозможно описать нюансы той или иной целевой машины достаточно подробно, чтобы иметь возможность генерировать для нее хороший код. В этой главе в качестве целевого компьютера мы рассмотрим регистровую машину, являющуюся типичным представителем ряда мини-компьютеров. Однако рассматриваемые здесь принципы и технологии применимы и к множеству других классов машин.

Наш целевой компьютер представляет собой машину с адресацией байтов, словом из четырех байтов и  $n$  регистрами общего назначения R0, R1, ..., R $n-1$ . Он имеет двухадресные инструкции вида

*op      source, destination*

где *op* — код операции, а *source* (источник) и *destination* (получатель) — поля данных. Среди прочих имеются следующие коды операций.

|     |                                                   |
|-----|---------------------------------------------------|
| MOV | (переместить <i>source</i> в <i>destination</i> ) |
| ADD | (прибавить <i>source</i> к <i>destination</i> )   |
| SUB | (вычесть <i>source</i> из <i>destination</i> )    |

Прочие инструкции будут представлены по необходимости.

Поля источника и получателя недостаточно длинны для хранения адресов памяти, поэтому определенные наборы битов в этих полях определяют, что следующее за инструкцией слово содержит операнды и/или адреса. Источник и получатель инструкции определяются комбинированием регистров и ячеек памяти с режимами адресации. В следующем далее описании *contents(a)* означает содержимое регистра или ячейки памяти, представленных *a*.

Далее приведены режимы адресации вместе с их ассемблерным представлением и стоимостью.

| РЕЖИМ                     | ПРЕДСТАВЛЕНИЕ | АДРЕС                     | ДОБАВОЧНАЯ СТОИМОСТЬ |
|---------------------------|---------------|---------------------------|----------------------|
| Абсолютный                | M             | M                         | 1                    |
| Регистровый               | R             | R                         | 0                    |
| Индексированный           | c(R)          | c + contents(R)           | 1                    |
| Косвенный регистровый     | *R            | contents(R)               | 0                    |
| Косвенный индексированный | *c(R)         | contents(c + contents(R)) | 1                    |

Ячейка памяти M или регистр R при использовании в качестве источника или получателя представляют сами себя. Так, в инструкции

MOV R0, M

выполняется сохранение содержимого регистра R0 в ячейке памяти M.

Смещение c от значения в регистре R записывается как c(R). Следовательно, инструкция

MOV 4(R0), M

сохраняет значение contents(4+contents(R0)) в ячейке памяти M.

Косвенная версия последних двух режимов указывается с помощью префикса \*. Таким образом, инструкция

MOV \*4(R0), M

сохраняет в ячейке памяти M значение contents(contents(4+contents(R0))).

Последний режим адресации позволяет использовать в качестве источника константу.

| РЕЖИМ      | ПРЕДСТАВЛЕНИЕ | КОНСТАНТА | ДОБАВОЧНАЯ СТОИМОСТЬ |
|------------|---------------|-----------|----------------------|
| Буквальный | #c            | c         | 1                    |

Так, инструкция

MOV #1, R0

заносит в регистр R0 константу 1.

## Стоимость инструкции

Стоимость инструкции определяется как единица плюс стоимости, связанные с режимами адресации источника и получателя (указанные в приведенной выше таблице режимов адресации в столбце "Добавочная стоимость"). Эта цена соответствует длине инструкции (в словах). Стоимость использования регистров равна нулю, в то время как стоимость использования ячеек памяти или констант равна единице, поскольку такие операнды должны быть записаны вместе с инструкцией.

Если важно количество используемой памяти, то очевидно, что мы должны по возможности минимизировать длину инструкций. При этом мы получаем дополнительную выгоду. На большинстве машин и для большинства инструкций время, требующееся для выборки инструкции из памяти, превышает время, затрачиваемое на ее исполнение. Следовательно, минимизируя длину инструкций, мы также снижаем общее время выполнения программы<sup>1</sup>. Далее приведены некоторые конкретные примеры.

<sup>1</sup> Критерии стоимости скорее иллюстративный, чем реалистичный. Использование полного слова для инструкции упрощает правило определения стоимости инструкции. Более точная оценка времени выполнения инструкции должна учитывать, требует ли инструкция загрузки значения или адреса операнда из памяти.

- Инструкция `MOV R0, R1` копирует содержимое регистра `R0` в регистр `R1`. Стоимость такой инструкции равна единице, поскольку она занимает только одно слово памяти.
- Инструкция `MOV R5, M` копирует содержимое регистра `R5` в ячейку памяти `M`. Стоимость этой инструкции равна двум, поскольку адрес ячейки памяти `M` находится в слове, следующем за инструкцией.
- Инструкция `ADD #1, R3` добавляет константу 1 к содержимому регистра `R3` и имеет стоимость два, поскольку константа 1 должна находиться в слове, следующем за инструкцией.
- Инструкция `SUB 4(R0), *12(R1)` сохраняет значение  $\text{contents}(\text{contents}(12 + \text{contents}(R1))) - \text{contents}(\text{contents}(4 + R0))$  в приемнике `*12(R1)`. Стоимость этой инструкции равна трем, поскольку константы 4 и 12 хранятся в двух словах, следующих за самой инструкцией.

Некоторые сложности генерации кода для такой машины можно увидеть, рассмотрев коды, генерируемые для трехадресной инструкции вида `a := b + c`, где `b` и `c` — простые переменные в различных ячейках памяти, обозначенных этими именами. Вот несколько примеров.

- `MOV b, R0  
ADD c, R0  
MOV R0, a` стоимость = 6
- `MOV b, a  
ADD c, a` стоимость = 6

Полагая, что `R0`, `R1` и `R2` содержат соответственно адреса `a`, `b` и `c`, можно использовать следующие инструкции.

- `MOV *R1, *R0  
ADD *R2, *R0` стоимость = 2

Считая, что `R1` и `R2` содержат соответственно значения `b` и `c` и значение `b` для дальнейших вычислений не требуется, можно использовать следующий код.

- `ADD R2, R1  
MOV R1, a` стоимость = 3

Мы видим, что для генерации хорошего кода мы должны эффективно использовать возможности режимов адресации машины. Желательно, по возможности, хранить *l*- и *r*-значения имен в регистрах, особенно если эти имена будут использоваться в ближайшее время.

## 9.3. Управление памятью во время исполнения

Как мы уже видели в главе 7, “Среды времени исполнения”, семантика процедур в языке определяет, каким образом в процессе выполнения программы имена связываются с памятью. Информация, необходимая в процессе выполнения процедуры, хранится в блоке памяти, именуемом записью активации; память для локальных имен процедуры также выделяется в ее записи активации.

В этом разделе мы рассмотрим код, генерируемый для управления записями активации в процессе выполнения программы. В разделе 7.3 были представлены две стандартные стратегии распределения памяти, а именно — статическое и стековое распределение. При статическом распределении памяти положение записи активации в памяти фиксируется во время компиляции. В случае стекового распределения новая запись активации вносится в стек для каждого выполнения процедуры. Запись снимается со стека

ка по окончании активации. Позже в этом разделе мы рассмотрим, каким образом целевой код процедуры может обращаться к объектам данных в записях активации.

Как мы видели в разделе 7.2, запись активации для процедуры содержит поля для параметров, результатов, информации о состоянии машины, локальных данных, временных переменных и т.п. В этом разделе мы проиллюстрируем стратегии распределения с использованием поля состояния машины для хранения адреса возврата и поля для локальных данных. Мы полагаем, что остальные поля обрабатываются так же, как это описано в главе 7, “Среды времени исполнения”.

Поскольку распределение и освобождение записей активации во время исполнения представляют собой части последовательностей вызова и возврата из процедуры, мы обратим основное внимание на следующие трехадресные инструкции.

1. call
2. return
3. halt
4. action (под которой подразумеваются остальные инструкции процедуры)

Например, трехадресный код процедур с и р (рис. 9.4) содержит только эти типы инструкций. Размер и схема записей активации передаются генератору кода посредством информации об именах, хранящейся в таблице символов. Для ясности мы показали на рис. 9.4 схему записей активации, а не записи в таблице символов.

Мы полагаем, что память времени исполнения разделена на области для кода, статических данных и стека, как было представлено в разделе 7.2 (дополнительная область для кучи здесь не используется).

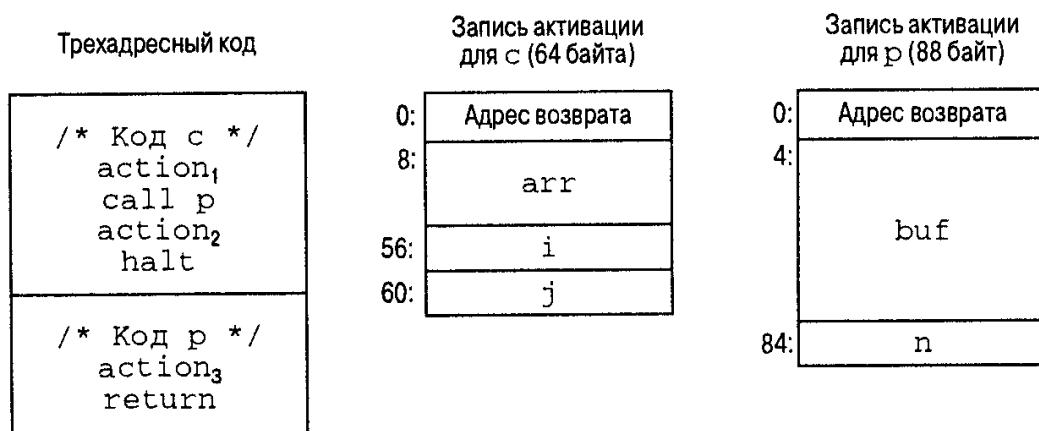


Рис. 9.4. Вход генератора кода

## Статическое распределение

Рассмотрим код, необходимый для реализации статического распределения. Инструкция call промежуточного кода реализуется последовательностью двух инструкций целевой машины: MOV сохраняет адрес возврата, а GOTO передает управление целевому коду вызываемой процедуры.

```
MOV #here+20, callee.static_area
GOTO callee.code_area
```

Атрибуты *callee.static\_area* и *callee.code\_area* являются константами, представляющими собой адрес записи активации и первой инструкции вызываемой процедуры соответственно. Источник *#here+20* инструкции MOV является адресом возврата, т.е. адресом инструкции, сле-

дующей за инструкцией GOTO (из раздела 9.2 следует, что суммарная стоимость трех констант и двух инструкций в последовательности вызова равна 5 словам, или 20 байт).

Код процедуры заканчивается возвратом в вызывающую процедуру (за исключением того, что первая процедура не имеет вызывающей, поэтому ее последней инструкцией является HALT, передающая управление операционной системе. Возврат из процедуры *callee* осуществляется инструкцией

GOTO \**callee.static\_area*

которая передает управление по адресу, сохраненному в начале записи активации.

### Пример 9.1

Код на рис. 9.5 построен по процедурам *s* и *p*, показанным на рис. 9.4. Здесь мы использовали псевдоинструкцию ACTION, представляющую трехадресный код, несущественный в данном обсуждении. Мы произвольно размещаем коды этих процедур по адресам 100 и 200 и предполагаем, что каждая инструкция ACTION имеет размер 20 байт. Записи активации для процедур распределены статически начиная с адресов 300 и 364 соответственно.

Инструкции, начинающиеся с адреса 100, реализуют следующий трехадресный код первой процедуры *s*.

action<sub>1</sub>; call p; action<sub>2</sub>; halt

Следовательно, выполнение начинается с инструкции ACTION<sub>1</sub> по адресу 100. Инструкция MOV по адресу 120 сохраняет адрес возврата 140 в поле состояния машины (которое является первым словом записи активации *p*). Инструкция GOTO по адресу 132 передает управление первой инструкции целевого кода вызываемой процедуры.

Поскольку адрес 140 в вызывающей последовательности сохранен по адресу 364, при выполнении инструкции GOTO по адресу 220 \*364 представляет собой не что иное, как 140. Таким образом, управление возвращается по адресу 140, и продолжается выполнение процедуры *s*. □

```
100: ACTION1 /* Код процедуры s */
120: MOV #140, 364 /* Сохраняем адрес возврата 140 */
132: GOTO 200 /* Вызов p */
140: ACTION2
160: HALT

...
200: ACTION1 /* Код процедуры p */
220: GOTO *364 /* Возврат по адресу, сохраненному
 в слове по адресу 364 */

...
300:
304:
...
364:
368:

/* По адресам 300-363 хранится
запись активации с */
/* Адрес возврата */
/* Локальные данные с */

/* По адресам 364-451 хранится
запись активации p */
/* Адрес возврата */
/* Локальные данные p */
```

Рис. 9.5. Целевой код для показанного на рис. 9.4 входа генератора кода

## Стековое распределение

Статическое распределение может стать стековым при использовании относительных адресов для памяти в записях активации. Положение записи активации процедуры неизвестно до момента выполнения программы. При стековом распределении это положение обычно хранится в регистре, так что доступ к слову в записи активации можно осуществить путем смещения относительно значения в этом регистре. Для этой цели подходит индексный режим адресации нашей целевой машины.

Как мы видели в разделе 7.3, относительные адреса в записи активации могут быть получены как смещения относительно любого известного положения в этой записи. Для удобства мы будем использовать положительные смещения, сохраняя в регистре SP указатель на начало записи активации на вершине стека. При вызове процедуры вызывающая процедура увеличивает SP и передает управление вызываемой процедуре. После того как управление вновь вернется вызывающей процедуре, она уменьшает SP, освобождая таким образом запись активации вызываемой процедуры<sup>2</sup>.

Код первой процедуры инициализирует стек посредством присвоения регистру SP значения начала области стека в памяти:

```
MOV #stackstart, SP /* Инициализация стека */
...Код первой процедуры...
HALT /* Прекращение выполнения */
```

Последовательность вызова процедуры увеличивает SP, сохраняет адрес возврата и передает управление вызываемой процедуре.

```
ADD #caller.recordsize, SP
MOV #here+16, *SP /* Сохранение адреса возврата */
GOTO callee.code_area
```

Атрибут *caller.recordsize* представляет размер записи активации, так что инструкция ADD приводит к тому, что SP указывает на начало следующей записи активации. Источник *#here+16* в инструкции MOV представляет собой адрес инструкции, следующей за GOTO; он сохраняется в слове по адресу, на который указывает SP.

Последовательность возврата состоит из двух частей. Вызываемая процедура передает управление по адресу возврата с помощью инструкции

```
GOTO *0(SP) /* Возврат в вызывающую процедуру */
```

Причина использования *\*0(SP)* в инструкции GOTO состоит в том, что нам требуется два уровня косвенности: *0(SP)* представляет собой адрес первого слова в записи активации, а *\*0(SP)* — сохраненный в нем адрес возврата.

Вторая часть последовательности возврата находится в вызывающей процедуре и уменьшает SP, восстанавливая его исходное значение. Таким образом, после вычитания SP указывает на начало записи активации вызывающей процедуры:

```
SUB #caller.recordsize, SP
```

Более полное обсуждение использования вызывающих последовательностей и распределения работы между вызывающей и вызываемой процедурами имеется в разделе 7.3.

<sup>2</sup> При использовании отрицательных смещений SP может указывать на конец стека, а вызываемая процедура может увеличивать значения этого регистра.

## Пример 9.2

Программа на рис. 9.6 представляет собой сокращенный вариант трехадресного кода программы Pascal, рассмотренной в разделе 7.1. Процедура *q* рекурсивна, поэтому могут одновременно существовать несколько активаций *q*.

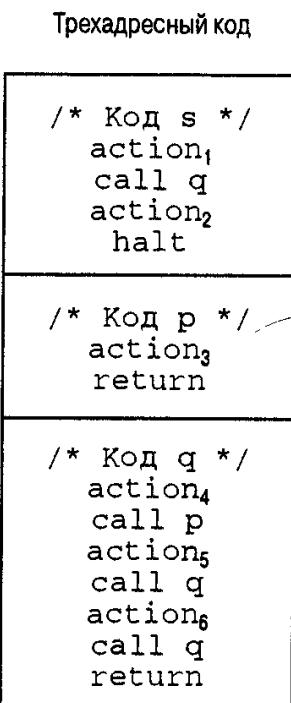


Рис. 9.6. Трехадресный код, иллюстрирующий стековое распределение

Предположим, что размеры записей активации процедур *s*, *p* и *q* определяются в процессе компиляции и равны соответственно *ssize*, *psize* и *qsize*. Первое слово каждой записи активации хранит адрес возврата. Мы полагаем, что наши процедуры начинаются с адресов 100, 200 и 300 соответственно, а стек — с адреса 600. Целевой код программы, приведенной на рис. 9.6, выглядит при этом следующим образом.

```
/* Код s */
100: MOV #600, SP /* Инициализация стека */
108: ACTION1
128: ADD #ssize, SP /* Начало последовательности вызова */
136: MOV #152, *SP /* Внесение адреса возврата в стек */
144: GOTO 300 /* Вызов q */
152: SUB #ssize, SP /* Восстановление SP */
160: ACTION2
180: HALT
...
/* Код p */
200: ACTION3
220: GOTO *0(SP) /* Возврат */
...
/* Код q */
300: ACTION4
320: ADD #qsize, SP /* Условный переход к 456 */
328: MOV #344, *SP /* Внесение в стек адреса возврата */
336: GOTO 200 /* Вызов p */
```

```

344: SUB #qsize, SP
352: ACTION5
372: ADD #qsize, SP
380: MOV #396, *SP /* Внесение в стек адреса возврата */
388: GOTO 300 /* Вызов q */
396: SUB #qsize, SP
404: ACTION6
424: ADD #qsize, SP
432: MOV #448, *SP /* Внесение в стек адреса возврата */
440: GOTO 300 /* Вызов q */
448: SUB #qsize, SP
456: GOTO *0(SP) /* Возврат */
...
600: /* Здесь начинается стек */

```

Мы полагаем, что ACTION<sub>4</sub> содержит условный переход к адресу 456 последовательности возврата из q; в противном случае рекурсивная процедура q будет бесконечно вызывать сама себя. Далее в примере предполагаем, что из первого вызова q возвращается не сразу, а из дальнейших вызовов — сразу.

Пусть *ssize*, *psize* и *qsize* равны соответственно 20, 40 и 60. Первой инструкцией по адресу 100 SP инициализируется значением 600, начальным адресом стека. SP получает значение 620 непосредственно перед передачей управления от s к q, поскольку значение *ssize* равно 20. Затем, когда q вызывает p, инструкция по адресу 320 увеличивает SP до 680, где начинается запись активации для p. После возврата управления в q SP вновь становится равным 620. Если два следующих рекурсивных вызова q сразу возвращают управление, максимальное значение SP в процессе работы равно 680. Заметим, однако, что последняя используемая ячейка памяти стека — 739, поскольку запись активации q начинается с адреса 680 и занимает 60 байт. □

## Адресация имен во время исполнения

Стратегия распределения памяти и размещение локальных данных в записи активации процедуры определяют, каким образом производится обращение к памяти для имен. В главе 8, “Генерация промежуточного кода”, мы полагали, что имя в трехадресной инструкции представляет собой указатель на запись в таблице символов для данного имени. Такой подход имеет значительное преимущество — он делает компилятор более переносимым, поскольку начальная стадия компиляции не изменяется при переносе на другую целевую машину, где требуется новая организация времени исполнения. Однако использование специфических последовательностей доступа к данным при генерации промежуточного кода может давать значительную выгоду в оптимизирующем компиляторе, поскольку позволяет оптимизатору воспользоваться деталями, обычно не проявляющимися в простых трехадресных инструкциях.

В любом случае имена должны быть в конечном счете заменены кодом для доступа к соответствующим ячейкам памяти. Рассмотрим некоторые детали простейшей трехадресной инструкции копирования *x := 0*. Предположим, что после обработки объявлений в процедуре запись таблицы символов для x содержит относительный адрес 12. Вначале рассмотрим случай, когда x находится в статически выделенной области памяти, начинающейся с адреса *static*. В этом случае адрес x во время исполнения равен *static+12*. Хотя в конечном счете компилятор в состоянии определить значение *static+12* во время компиля-

ции, положение статической области памяти может быть не известно при генерации промежуточного кода для доступа к данному имени. В таком случае имеет смысл генерировать трехадресный код для “вычисления”  $static+12$ , ясно представляя себе, что это вычисление будет выполнено в процессе генерации кода (либо загрузчиком перед запуском программы). Присвоение  $x := 0$  транслируется таким образом в  $static[12] := 0$ . Если статическая область памяти начинается с адреса 100, целевой код этой инструкции будет выглядеть как `MOV #0, 112.`

С другой стороны, предположим, что мы имеем язык типа Pascal и для доступа к нелокальным именам используются дисплеи, рассмотренные в разделе 7.4. Предположим также, что дисплей хранится в регистрах, а имя  $x$  локально по отношению к процедуре, указатель дисплея которой хранится в регистре R3. Тогда копирование  $x := 0$  можно транслировать в трехадресные инструкции

```
t1 := 12 + R3
*t1 := 0
```

в которых  $t_1$  содержит адрес  $x$ . Такая последовательность может быть реализована одной машинной инструкцией

```
MOV #0, 12(R3)
```

Заметьте, что значение в регистре R3 во время компиляции не может быть определено.

## 9.4. Базовые блоки и графы потоков

Представление трехадресных инструкций в виде так называемого графа потока (flow graph) облегчает понимание алгоритмов генерации кода (даже если такой график не строится явно). Узлы графа потока представляют вычисления, а дуги — поток управления. В главе 10, “Оптимизация кода” мы будем использовать график потока для сбора информации о промежуточной программе. Некоторые алгоритмы назначения регистров также используют графы потоков для поиска внутренних циклов, на которые программа, как ожидается, тратит основное время работы.

### Базовые блоки

*Базовым блоком* (basic block) называется последовательность смежных инструкций, в которые поток управления входит в их начале и покидает в конце, без останова программы или возможности ветвления (за исключением конца блока). Так, следующая последовательность трехадресных инструкций образует базовый блок.

```
t1 := a * a
t2 := a * b
t3 := 2 * t2
t4 := t1 + t3
t5 := b * b
t6 := t4 + t5
```

(9.1)

О трехадресной инструкции  $x := y + z$  говорят, что она *определяет*  $x$  и *использует*  $y$  и  $z$  (или *ссылается* на них). Имя в базовом блоке называется *живым* в данной точке, если его значение используется в программе после этой точки, возможно в другом базовом блоке.

Для разделения последовательности трехадресных инструкций на базовые блоки можно использовать следующий алгоритм.

### Алгоритм 9.1. Разделение на базовые блоки

*Вход.* Последовательность трехадресных инструкций.

*Выход.* Список базовых блоков, такой, что каждая трехадресная инструкция принадлежит только одному блоку.

*Метод.*

1. Вначале мы определяем набор лидеров, первых инструкций базовых блоков. Для этого применяем следующие правила.
  - i) Первая инструкция является лидером.
  - ii) Любая инструкция, представляющая собой целевую инструкцию условного или безусловного перехода, является лидером.
  - iii) Любая инструкция, непосредственно следующая за условным или безусловным переходом, является лидером.
2. Базовый блок для каждого лидера состоит из него самого и всех инструкций до следующего лидера, не включая его, либо до конца программы. □

### Пример 9.3

Рассмотрим фрагмент исходного кода, показанный на рис. 9.7. Он вычисляет скалярное произведение двух векторов  $a$  и  $b$  длиной 20. Список трехадресных инструкций, выполняющих эти вычисления на нашей целевой машине, показан на рис. 9.8.

```
begin
 prod := 0;
 i := 1;
 do begin
 prod := prod + a[i] * b[i];
 i := i + 1
 end
 while i <= 20
end
```

Рис. 9.7. Программа вычисления скалярного произведения

```
(1) prod := 0
(2) i := 1
(3) t1 := 4 * i
(4) t2 := a [t1] /* Вычисление a[i] */
(5) t3 := 4 * i
(6) t4 := b [t3] /* Вычисление b[i] */
(7) t5 := t2 * t4
(8) t6 := prod + t5
(9) prod := t6
(10) t7 := i + 1
(11) i := t7
(12) if i <= 20 goto (3)
```

Рис. 9.8. Трехадресный код вычисления скалярного произведения

Применим алгоритм 9.1 к трехадресному коду, представленному на рис. 9.8, для определения его базовых блоков. Инструкция (1) является лидером согласно правилу (i), а инструкция (3) — согласно правилу (ii), поскольку последняя инструкция может передать ей управление. В соответствии с правилом (iii) инструкция, следующая за (12) (вспомним, что рис. 9.7 — всего лишь фрагмент программы), также является лидером. Таким образом, инструкции (1) и (2) образуют базовый блок. Оставшаяся часть фрагмента программы, начинающаяся с инструкции (3), образует второй базовый блок.  $\square$

## Преобразования в базовых блоках

Базовый блок вычисляет некоторое множество выражений. Эти выражения представляют собой значения имен, “живых” при выходе из блока. Два базовых блока называются эквивалентными, если они вычисляют одинаковые наборы выражений.

К базовому блоку могут быть применены различные преобразования, не изменяющие множество вычисляемых им выражений. Многие из этих преобразований полезны для улучшения кода, окончательно генерируемого из базового блока. В следующей главе мы рассмотрим, каким образом глобальный “оптимизатор” кода пытается использовать эти преобразования для уменьшения времени работы и размера конечной программы. Имеется два важных типа локальных преобразований базовых блоков: преобразования, сохраняющие структуру, и алгебраические преобразования.

## Преобразования, сохраняющие структуру

Основными преобразованиями, сохраняющими структуру, являются следующие.

1. Устранение общих подвыражений
2. Устранение “мертвого” кода
3. Переименование временных переменных
4. Перестановка независимых соседних инструкций

Рассмотрим их подробнее. Для простоты предположим, что базовые блоки не содержат массивы, указатели и вызовы процедур.

*1. Устранение общих подвыражений.* Рассмотрим следующий базовый блок.

```
a := b + c
b := a - d
c := b + c
d := a - d
```

(9.2)

Вторая и четвертая инструкции вычисляют одно и то же выражение, а именно —  $b+c-d$ . Следовательно, этот базовый блок можно преобразовать в эквивалентный блок.

```
a := b + c
b := a - d
c := b + c
d := b
```

(9.3)

Заметим, что хотя правые части первой и третьей инструкций в (9.2) и (9.3) выглядят одинаково, вторая инструкция переопределяет  $b$ . Таким образом, значение  $b$  в третьей инструкции отличается от значения  $b$  в первой; первая и третья инструкции вычисляют разные выражения.

**2. Устранение “мертвого” кода.** Предположим, что  $x$  — “мертвая” переменная, т.е. нигде не используется после того, как в базовом блоке встречается инструкция  $x := y + z$ . В таком случае эта инструкция может быть благополучно удалена без изменения значения базового блока.

**3. Переименование временных переменных.** Предположим, что у нас есть инструкция  $t := b + c$ , где  $t$  — временная переменная. Если мы изменим эту инструкцию на  $u := b + c$ , где  $u$  — новая временная переменная, и заменим все последующие обращения к данной переменной  $t$  обращениями к  $u$ , базовый блок окажется неизмененным. Мы всегда можем преобразовать базовый блок в эквивалентный, если каждая инструкция блока, которая определяет временную переменную, определяет новую временную переменную. Такой базовый блок назовем блоком *нормального вида* (normal form block).

**4. Перестановка инструкций.** Предположим, что у нас есть блок с двумя следующими соседними инструкциями.

$$\begin{aligned}t_1 &:= b + c \\t_2 &:= x + y\end{aligned}$$

Мы можем переставить эти инструкции без изменения значения блока тогда и только тогда, когда ни  $x$ , ни  $y$  не являются  $t_1$  и ни  $b$ , ни  $c$  не являются  $t_2$ . Заметим, что базовый блок нормального вида допускает все возможные перестановки инструкций<sup>3</sup>.

## Алгебраические преобразования

Для изменения множества выражений, вычисляемых базовым блоком, на алгебраически эквивалентное множество можно использовать несчетное количество алгебраических преобразований. Нас интересуют те, которые упрощают выражения или заменяют дорогостоящие с точки зрения вычислений операторы более простыми. Например, инструкции типа  $x := x + 0$  или  $x := x * 1$  могут быть удалены из базового блока без какого-либо влияния на множество вычисляемых выражений. Оператор возведения в степень в выражении  $x := y ** 2$  обычно реализуется вызовом функции. При применении алгебраического преобразования эта инструкция заменяется более простой и дешевой, но эквивалентной инструкцией  $x := y * y$ . Более подробно алгебраические преобразования рассматриваются в разделах 9.9 и 10.3, посвященных локальной оптимизации и оптимизации базовых блоков.

## Графы потоков

Ко множеству базовых блоков, составляющих программу, мы можем добавить информацию о потоке управления с помощью построения так называемого *графа потока* (flow graph). Узлами графа потока являются базовые блоки. Один из узлов определяется как *начальный* (initial) — это блок, лидер которого является первой инструкцией программы. От блока  $B_1$  к блоку  $B_2$  имеется направленная дуга, если блок  $B_2$  может непосредственно следовать за блоком  $B_1$  в некоторой последовательности выполнения. Это возможно в следующих случаях.

1. Имеется условный или безусловный переход от последней инструкции  $B_1$  к первой инструкции  $B_2$ .

---

<sup>3</sup> Возможные в указанном только что смысле. Например, в блоке  
$$\begin{aligned}t1 &:= a + b & b &:= e + f & t2 &:= b + c\end{aligned}$$
 ничего нельзя менять, хотя он и подпадает под определение нормального вида. — Прим. ред.

2.  $B_2$  в программе непосредственно следует за  $B_1$ , при этом  $B_1$  не заканчивается инструкцией безусловного перехода.

Мы говорим, что  $B_1$  является *предшественником*  $B_2$ , а  $B_2$  — *преемником*  $B_1$ .

#### Пример 9.4

Граф потока программы, представленной на рис. 9.7, показан на рис. 9.9.  $B_1$  является начальным блоком. Обратите внимание, что переход к инструкции (3) заменен эквивалентным переходом к началу блока  $B_2$ .  $\square$

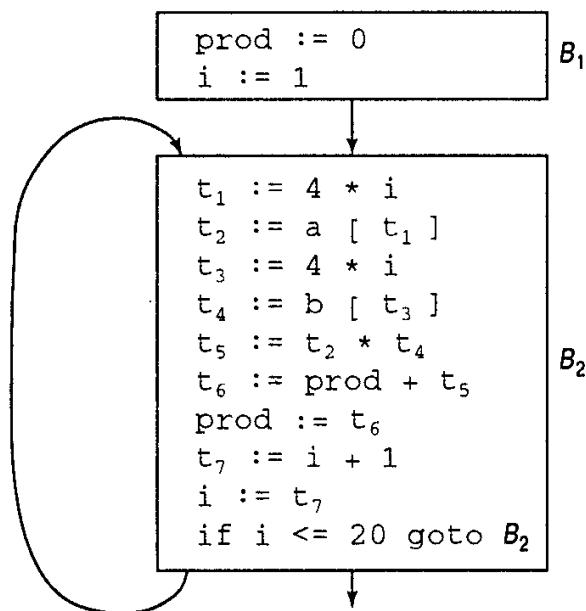


Рис. 9.9. Граф потока программы

#### Представление базовых блоков

Базовые блоки могут быть представлены различными структурами данных. Например, после разделения трехадресных инструкций на блоки в соответствии с алгоритмом 9.1 каждый базовый блок может быть представлен записью, состоящей из итогового числа четверок в блоке, за которым следуют указатель на лидера ( первую четверку ) блока и списки предшественников и преемников блоков. Иным способом представления может служить связанный список четверок в каждом блоке. Явное указание номеров четверок в инструкциях переходов в конце базовых блоков может вызвать проблемы при перемещении четверок в процессе оптимизации кода. Например, если блок  $B_2$ , который на рис. 9.9 содержит инструкции с (3) по (12), будет перемещен в массиве четверок на новое место или изменит размеры, номер (3) в инструкции `if i <= 20 goto (3)` должен быть заменен. Таким образом, предпочтительнее осуществлять переходы к блокам, а не конкретным четверкам, что и было сделано на рис. 9.9.

Важно отметить, что дуга графа потока от блока  $B$  к блоку  $B'$  не определяет условия, при которых управление будет передано от блока  $B$  к блоку  $B'$ . Дуга ничего не говорит о том, будет ли передано управление блоку  $B'$  при выполнении условия в конце блока  $B$  (если там находится условный переход) или же, напротив, в случае его невыполнения. Эта информация может быть получена только из самой инструкции перехода в блоке  $B$ .

## Циклы

Что собой представляют циклы в графе потока и как найти все циклы? Чаще всего на этот вопрос ответить очень просто. Например, на рис. 9.9 имеется один цикл, состоящий из блока  $B_2$ . Однако дать общий ответ на этот вопрос существенно сложнее, и мы подробно рассмотрим его в следующей главе. Пока лишь скажем, что цикл представляет собой набор узлов графа потока, удовлетворяющий следующим условиям.

1. Все узлы набора *сильно связаны*, т.е. имеется путь единичной или большей длины от любого узла в цикле к любому другому, полностью принадлежащий циклу.
2. Набор узлов имеет *единственный вход*, т.е. узел в цикле такой, что любой путь к узлу в цикле из узла вне цикла проходит через вход.

Цикл, не содержащий других циклов, называется *внутренним*.

## 9.5. Информация о последующем использовании

В этом разделе мы собираем информацию об использовании имен в базовых блоках. Если имя в регистре более не является необходимым, этот регистр может быть назначен для другого имени. Эта идея — хранения имени в памяти только в том случае, если оно будет использоваться впоследствии — может применяться в ряде контекстов. Так, мы использовали ее в разделе 5.8 для назначения памяти значениям атрибутов. Этую же идею использует для назначения регистров и простой генератор кода из следующего раздела. В качестве последнего применения мы рассмотрим назначение памяти временным переменным.

### Вычисление последующих использований

*Использование* имени в трехадресной инструкции определяется следующим образом. Предположим, что трехадресная инструкция  $i$  присваивает значение переменной  $x$ . Если инструкция  $j$  содержит  $x$  в качестве операнда и управление может перейти от инструкции  $i$  к инструкции  $j$  по пути, на котором не имеется присвоений переменной  $x$ , то мы говорим, что  $j$  *использует* значение  $x$ , вычисленное в  $i$ .

Для каждой трехадресной инструкции  $x := y \text{ op } z$  мы хотим найти последующие использования  $x$ ,  $y$  и  $z$ . В настоящее время мы не касаемся использования имен вне базового блока, содержащего данную трехадресную инструкцию, однако при желании мы можем попытаться определить, используется ли данное имя вне блока, с помощью технологии анализа “живых” (или активных) переменных из главы 10, “Оптимизация кода”.

Наш алгоритм для определения последующих использований выполняет обратный проход по каждому базовому блоку. Мы можем просканировать поток трехадресных инструкций для поиска концов базовых блоков в соответствии с алгоритмом 9.1. Поскольку вызовы процедур могут приводить к различным побочным эффектам, для удобства мы полагаем, что каждый вызов процедуры начинает новый базовый блок.

После того как мы нашли конец базового блока, сканируем блок от конца к началу, записывая (в таблицу символов) для каждого имени  $x$ , используется ли оно далее в блоке и, если нет, остается ли “живым” при выходе из блока. Определить, какие имена остаются “живыми” при выходе из блока, можно с помощью анализа потока данных, описанного в главе 10, “Оптимизация кода”. Однако если такой анализ не был выполнен, мы можем считать, что при выходе из блока остаются “живыми” все не временные переменные. Если алгоритмы генерации промежуточного кода или оптимизаций кода позволяют,

чтобы некоторые временные переменные использовались в нескольких блоках, то эти переменные также должны считаться “живыми”. Такие временные переменные стоит помечать соответствующим образом, иначе нам придется рассматривать все временные переменные как “живые”.

Предположим, что при обратном сканировании мы достигли инструкции  $i := x := y \text{ or } z$ . После этого мы выполняем следующие действия.

1. Присоединяем к инструкции  $i$  текущую информацию о последующем использовании и “жизненности”  $x$ ,  $y$  и  $z$ .<sup>4</sup>
2. В таблице символов мы определяем переменную  $x$  как “неживую” и “неиспользуемую в последующем”.
3. Определяем в таблице символов  $y$  и  $z$  как “живые” со следующим использованием в  $i$ . Заметим, что изменять порядок выполнения шагов (2) и (3) нельзя, поскольку  $x$  может представлять собой  $y$  или  $z$ .

Если трехадресная инструкция  $i$  имеет вид  $x := y$  или  $x := \text{op } y$ , выполняемые действия остаются теми же, что и ранее, с поправкой на отсутствие  $z$ .

## Память для временных имен

Хотя для оптимизирующего компилятора может оказаться полезным создание нового имени всякий раз, когда нужна временная переменная (см. пояснения в главе 10, “Оптимизация кода”), для хранения значений этих временных переменных требуется выделение памяти. Вместе с количеством используемых временных переменных растет и размер поля для временных имен в записи активации, рассмотренной в разделе 7.2.

Вообще говоря, мы в состоянии упаковать две временные переменные в одну ячейку памяти, если они не являются живыми одновременно. Поскольку почти все временные переменные определяются и используются внутри базовых блоков, для такой упаковки временных переменных можно применить информацию об их последующем использовании. В главе 10, “Оптимизация кода” обсуждается анализ потоков данных, необходимый для вычисления жизненности временных переменных, которые используются в разных блоках.

Мы можем распределить память для временных переменных, рассматривая каждую по очереди и назначая ее первой из ячеек памяти для временных переменных, не содержащих живую переменную. Если временная переменная не может быть назначена ни одной ранее созданной ячейке для временных переменных, в область данных текущей процедуры следует добавить память для новой временной переменной. Во многих случаях временные переменные можно упаковать в регистры, а не в ячейки памяти (см. следующий раздел).

Например, шесть временных переменных в базовом блоке (9.1) могут быть упакованы в две ячейки памяти. В следующем коде этим ячейкам соответствуют  $t_1$  и  $t_2$ .

```
t1 := a * a
t2 := a * b
t2 := 2 * t2
t1 := t1 + t2
t2 := b * b
t1 := t1 + t2
```

<sup>4</sup> Если  $x$  не является живым, то эта инструкция может быть удалена. Такое преобразование рассматривается в разделе 9.8.

## **9.6. Простой генератор кода**

Стратегия генерации кода в этом разделе создает целевой код для последовательности трехадресных инструкций. Она пересматривает каждую инструкцию, учитывая, какие из операндов находятся в данный момент в регистрах, и пытаясь извлечь пользу из этого факта. Для простоты мы полагаем, что для каждого оператора инструкции существует соответствующий оператор целевого машинного языка. Мы также считаем, что вычисленные результаты могут оставаться в регистрах до тех пор, пока это возможно. Мы сохраняем их в памяти только в следующих случаях: (a) если эти регистры требуются для проведения иных вычислений; (b) непосредственно перед вызовами процедур, переходами и помеченными инструкциями<sup>5</sup>.

Условие (b) означает, что непосредственно перед окончанием базового блока<sup>6</sup> все должно быть сохранено. Причина, по которой мы вынуждены так поступать, состоит в том, что, покидая базовый блок, мы можем попасть в несколько различных базовых блоков либо в один и тот же блок, достижимый из нескольких других. В любом случае нельзя полагать, что данные, используемые блоком, находятся в одних и тех же регистрах независимо от того, каким путем управление достигло данного блока. Следовательно, чтобы избежать возможных ошибок, наш простой алгоритм генерации кода сохраняет все данные перед тем, как пересечь границу базового блока либо перейти к вызову процедуры. Позже мы рассмотрим способы хранения некоторых данных в регистрах при пересечении границ блоков.

Мы можем получить недорогой код для трехадресной инструкции  $a := b + c$ , если сгенерируем одну инструкцию  $ADD\ Rj, Ri$  единичной стоимости, оставляя полученный результат  $a$  в регистре  $Ri$ . Такая последовательность допустима только в том случае, когда регистр  $Ri$  содержит  $b$ , регистр  $Rj$  —  $c$ , а  $b$  не является живой, т.е. не используется после выполнения инструкции.

Если  $R_i$  содержит  $b$ , но  $c$  находится в ячейке памяти (которую для удобства также обозначим как  $c$ ), мы можем генерировать последовательность

ADD C, RI СТОИМОСТЬ = 2

или

MOV C, Rj  
ADD Rj, Ri СТОИМОСТЬ = 3

с учетом того, что  $b$  не остается живой. Вторая последовательность более привлекательна в том случае, если с будет использоваться в дальнейшем, поскольку мы сможем получить его значение из регистра  $Rj$ . Имеется множество случаев, которые следует рассмотреть, — в зависимости от того, где в настоящий момент хранятся значения  $b$  и с и будет ли использоваться значение  $b$  в дальнейшем. Должны быть также рассмотрены случаи, когда  $b$  или с являются константами. Число случаев, которые необходимо рассмотреть, еще больше возрастет, если считать оператор + коммутатив-

<sup>5</sup> Однако для выполнения *символьного сброса* (*дампа*) памяти (symbolic dump), который делает доступными значения ячеек памяти и регистров, связывая их с именами, использованными в исходной программе, может оказаться более удобным сохранять в памяти переменные, определенные программистом (но не генерируемые компилятором временные переменные), сразу после их вычисления на случай не-предвиденной ошибки, вызывающей останов программы.

<sup>6</sup> Заметьте, мы не предполагаем, что четверки действительно разделены компилятором на базовые блоки; в любом случае, понятие базового блока полезно концептуально.

ным. Как видите, генерация кода приводит к изучению большого количества вариантов, и для каждого случая должен учитываться контекст, в котором используется данная трехадресная инструкция.

## Дескрипторы регистров и адресов

Алгоритм генерации кода для отслеживания связи содержимого регистров и адресов для имен использует дескрипторы.

1. Дескриптор регистров отслеживает текущее содержимое каждого регистра. Когда для вычислений требуется новый регистр, происходит обращение к дескриптору за информацией о регистре. Мы полагаем, что изначально дескриптор регистров указывает, что все регистры пусты (это не так, если значения регистров используются в разных блоках). По ходу генерации кода блока в любой момент времени каждый регистр может быть пустым или хранить значение одного или нескольких имен.
2. Дескриптор адреса отслеживает ячейку памяти (или ячейки), в которой в процессе выполнения программы может быть найдено текущее значение имени. Этой ячейкой может быть регистр, местоположение в стеке, адрес памяти или некоторое их множество, поскольку при копировании значение в старом месте сохраняется. Эта информация может храниться в таблице символов и использоваться для определения метода доступа к имени.

## Алгоритм генерации кода

Алгоритм генерации кода получает на входе последовательность трехадресных инструкций, составляющих базовый блок. Для каждой трехадресной инструкции вида  $x := y \text{ op } z$  мы выполняем следующие действия.

1. Вызываем функцию *getreg* для определения места  $L$ , где должен быть сохранен результат вычисления  $y \text{ op } z$ . Обычно  $L$  представляет собой регистр, но может быть и ячейкой памяти. Далее мы вкратце опишем работу функции *getreg*.
2. Обращаемся к дескриптору адресов  $y$  для определения  $y'$ , текущего положения  $y$  (или одного из них). Если значение  $y$  есть и в памяти, и в регистре, в качестве  $y'$  предпочтительнее регистр. Если значение  $y$  еще не находится в  $L$ , генерируем инструкцию *MOV*  $y', L$  для размещения копии  $y$  в  $L$ .
3. Генерируем инструкцию *OP*  $z', L$ , где  $z'$  — текущее положение  $z$ . Здесь также, если значение  $z$  есть и в регистре, и в памяти, предпочтается регистр. Обновляем дескриптор адреса  $x$  с тем, чтобы в нем хранилась информация о том, что местоположением  $x$  является  $L$ . Если  $L$  представляет собой регистр, обновляем его дескриптор для указания того, что он содержит значение  $x$ , и удаляем  $x$  из других дескрипторов регистров.
4. Если текущие значения  $y$  и/или  $z$  в дальнейшем не используются, не являются живыми при выходе из блока и находятся в регистрах, то изменим соответствующие дескрипторы регистров, чтобы указать, что после выполнения  $x := y \text{ op } z$  эти регистры более не содержат  $y$  и/или  $z$ .

Если текущая трехадресная инструкция содержит унарный оператор, предпринимаемые шаги аналогичны приведенным выше, так что мы опускаем их подробное описание.

Важным специальным случаем является трехадресная инструкция  $x := y$ . Если  $y$  располагается в регистре, достаточно просто изменить информацию в дескрипторах регистров и адресов с тем, чтобы указать, что  $x$  находится только в регистре, хранящем значение  $y$ . Если при этом  $y$  не используется далее и не остается живым при выходе из блока, данный регистр более не хранит значение  $y$ .

Если  $y$  хранится только в памяти, в принципе мы можем записать, что значение  $x$  располагается в памяти там же, где и  $y$ , но это резко усложнит наш алгоритм, поскольку после этого мы не сможем изменять значение  $y$ , не сохранив предварительно значение  $x$ . Таким образом, если  $y$  хранится в памяти, мы используем *getreg* для поиска регистра, в который загружаем  $y$ , и делаем этот регистр местом хранения  $x$ .

Мы можем также генерировать инструкцию `MOV y, x`, что предпочтительнее, когда значение  $x$  далее в блоке не используется. Стоит заметить, что большинство (если не все) инструкций копирования может быть устранено при использовании алгоритмов улучшения блока и распространения копирования из главы 10, “Оптимизация кода”.

После того как мы обработали все трехадресные инструкции базового блока, сохраняя (посредством инструкций `MOV`) все живые имена, значения которых находятся не в связанных с ними ячейках памяти. Мы используем дескриптор регистров для определения, значения каких имен находятся в регистрах, и дескрипторы адресов для выяснения, не сохранено ли уже значение этого имени в памяти, и получения информации о том, должна ли сохраняться данная живая переменная. Если путем анализа потока данных информация о живых переменных не была получена, то в конце блока следует считать все определенные пользователем имена живыми.

## Функция *getreg*

Функция *getreg* возвращает местоположение  $L$  в памяти для значения  $x$  в присвоении  $x := y \ op z$ . Для того, чтобы эта функция давала мудрый и дальновидный результат, нужно затратить огромные усилия. Мы же рассмотрим здесь простейший вариант *getreg*, легкий в реализации и основанный на информации о последующих использованиях имен, полученной в предыдущем разделе.

1. Если имя  $y$  находится в регистре, который не хранит значений других переменных (вспомним, что инструкция копирования типа  $x := y$  приводит к одновременному хранению в одном регистре значений двух переменных),  $y$  не является живым именем и после выполнения  $x := y \ op z$  не используется, то в качестве  $L$  возвращается регистр, в котором хранится  $y$ . Обновляется и дескриптор адресов  $y$  для отражения того факта, что  $y$  больше не находится в  $L$ .
2. При невыполнении (1) в качестве  $L$  возвращается пустой регистр, если таковой имеется.
3. При невыполнении (2), если  $x$  используется далее в блоке либо  $op$  представляет собой оператор (типа индексирования), который требует использования регистра, производится поиск занятого регистра  $R$ . Значение  $R$  сохраняется в ячейке памяти  $M$  (с помощью оператора `MOV R, M`), если оно еще не находится там, обновляется дескриптор адресов  $M$  и возвращается  $R$ . Если  $R$  хранит значение нескольких переменных, инструкция `MOV` должна быть сгенерирована для каждой из переменных, которые должны быть сохранены. В качестве регистра можно выбрать один из тех, данные которого понадобятся еще не скоро или значение которого уже есть в памяти.

Мы оставляем этот выбор неопределенным в силу того, что превосходство какого-либо из методов выбора регистра не доказано.

- Если  $x$  в блоке не используется либо не удается найти подходящий регистр, в качестве  $L$  выбирается адрес  $x$  в памяти.

Более интеллектуальная функция *getreg* должна также учитывать последующее использование  $x$  и коммутативность оператора *or* при определении регистра, предназначенного для хранения значения  $x$ . Эти расширения функции *getreg* мы оставляем читателю в качестве упражнения.

### Пример 9.5

Присвоение  $d := (a - b) + (a + c) + (a - c)$  может быть транслировано в следующую последовательность трехадресных инструкций.

```
t := a - b
u := a - c
v := u + v
d := v + u
```

При этом  $d$  остается живым по окончании блока. Приведенный выше алгоритм генерации кода приводит для данной последовательности трехадресных инструкций к коду, показанному на рис. 9.10. Рядом с кодом приведены дескрипторы регистров и адресов в процессе генерации кода. Здесь для уменьшения объема таблицы не отображен тот факт, что  $a$ ,  $b$  и  $c$  все время находятся в памяти. Мы также полагаем, что  $t$ ,  $u$  и  $v$ , представляя собой временные переменные, не сохраняются в памяти, если мы не сохраним их явно с помощью инструкции *MOV*.

| ИНСТРУКЦИЯ   | СГЕНЕРИРОВАННЫЙ КОД     | ДЕСКРИПТОР РЕГИСТРОВ               | ДЕСКРИПТОР АДРЕСОВ                        |
|--------------|-------------------------|------------------------------------|-------------------------------------------|
| $t := a - b$ | MOV a, R0<br>SUB b, R0  | Rегистры пусты<br>R0 содержит $t$  | $t$ в R0                                  |
| $u := a - c$ | MOV a, R1<br>SUB c, R1  | R0 содержит $t$<br>R1 содержит $u$ | $u$ в R1                                  |
| $v := t + u$ | ADD R1, R0              | R0 содержит $v$                    | $u$ в R1                                  |
| $d := v + u$ | ADD R1, R0<br>MOV R0, d | R1 содержит $u$<br>R0 содержит $d$ | $v$ в R0<br>$d$ в R0<br>$d$ в R0 и памяти |

Рис. 9.10. Генерация кода

Первый вызов *getreg* возвращает R0 в качестве места для хранения вычисленного значения  $t$ . Поскольку  $a$  не находится в R0, мы генерируем пару инструкций *MOV a, R0* и *SUB b, R0*. Мы также обновляем дескриптор регистра, внося в него информацию о том, что R0 содержит  $t$ . Генерация кода выполняется аналогично до тех пор, пока не появляется инструкция  $d := u + v$ . Заметьте, что R1 становится пустым, поскольку  $u$  в последующем не используется. Затем мы генерируем инструкцию *MOV R0, d* для сохранения живой переменной  $d$  в конце блока.

Стоимость кода, приведенного на рис. 9.10, равна 12. Мы можем снизить ее до 11, сгенерировав *MOV R0, R1* непосредственно после первой инструкции и удалив инст-

рукцию MOV a, R1, однако для этого требуется более интеллектуальный алгоритм генерации кода. Такая замена выгодна, поскольку загрузка регистра R1 из регистра R0 имеет меньшую стоимость, чем загрузка регистра из памяти. □

## Генерация кода для других типов инструкций

Операции индексирования и работа с указателями в трехадресных инструкциях обрабатываются так же, как и бинарные операции. В таблице на рис. 9.11 показан код, генерируемый для присвоения  $a := b[i]$  и  $a[i] := b$  (полагая, что память для  $b$  выделена статически).

| ИНСТРУКЦИЯ  | i В РЕГИСТРЕ Ri |           | i В ПАМЯТИ Mi            |           | i В СТЕКЕ                   |           |
|-------------|-----------------|-----------|--------------------------|-----------|-----------------------------|-----------|
|             | Код             | Стоимость | Код                      | Стоимость | Код                         | Стоимость |
| $a := b[i]$ | MOV b(Ri), R    | 2         | MOV Mi, R<br>MOV b(R), R | 4         | MOV Si(A), R<br>MOV b(R), R | 4         |
| $a[i] := b$ | MOV b, a(Ri)    | 3         | MOV Mi, R<br>MOV b, a(R) | 5         | MOV Si(A), R<br>MOV b, a(R) | 5         |

Рис. 9.11. Сгенерированный код для присвоения с использованием индекса

Текущее положение  $i$  определяет генерируемую последовательность кода. В зависимости от того, находится ли  $i$  в регистре, ячейке памяти  $M_i$  или стеке со смещением  $S_i$  (при этом указатель на запись активации для  $i$  содержится в регистре A), имеется три варианта генерируемого кода. Регистр R в данном случае — это регистр, возвращаемый функцией *getreg*. В первом присвоении мы предпочтаем оставить a в регистре R, если a используется далее в блоке и регистр R свободен. Во втором присвоении мы полагаем, что массив a находится в статически выделенной памяти.

На рис. 9.12 показаны последовательности инструкций, сгенерированные для присвоений с использованием указателей:  $a := *p$  и  $*p := a$ . Здесь генерируемая последовательность определяется местоположением p.

| ИНСТРУКЦИЯ | P В РЕГИСТРЕ Rp |           | P В ПАМЯТИ Mp          |           | P В СТЕКЕ                 |           |
|------------|-----------------|-----------|------------------------|-----------|---------------------------|-----------|
|            | Код             | Стоимость | Код                    | Стоимость | Код                       | Стоимость |
| $a := *p$  | MOV *Rp, a      | 2         | MOV Mp, R<br>MOV *R, R | 3         | MOV Sp(A), R<br>MOV *R, R | 3         |
| $*p := a$  | MOV a, *Rp      | 2         | MOV Mp, R<br>MOV a, *R | 4         | MOV a, R<br>MOV R, *Sp(A) | 4         |

Рис. 9.12. Сгенерированный код для присвоения с использованием указателей

Применение одного из трех вариантов генерируемого кода зависит от того, где располагается указатель p — в регистре Rp, ячейке памяти Mp или стеке со смещением Sp и указателем на запись активации для p, находящимся в регистре A. Регистр R — это регистр, возвращаемый функцией *getreg*. Во втором присвоении мы полагаем, что a находится в статически выделенной памяти.

## Условные инструкции

Условные переходы реализуются на целевых машинах одним из двух способов. Первый состоит в ветвлении, если значение определенного регистра удовлетворяет одному из шести условий: отрицательно, равно нулю, положительно, не отрицательно, не равно нулю и не положительно. На такой машине трехадресная инструкция типа `if x < y goto z` может быть реализована посредством вычитания `y` из `x` с сохранением результата в регистре `R`, а затем перехода к `z`, если значение в регистре `R` отрицательно.

Второй подход, применимый на многих машинах, использует множество *кодов условий* для указания, было ли вычисленное последним или загруженное в регистр значение отрицательным, нулевым или положительным. Часто инструкция сравнения (на нашей машине — СМР) имеет полезное свойство устанавливать код условия без реального вычисления значения. Таким образом, СМР `x, y` устанавливает код условия положительным, если `x > y`, и т.д. Машинные инструкции условного перехода осуществляют переход при выполнении определенного условия — `<, =, >, ≥, ≠` или `≤`. Мы используем инструкцию `CJ<=z` для обозначения “переход к `z`, если код условия отрицательный или нулевой”. Например, `if x < y goto z` может быть реализовано следующим образом.

```
CMP x, y
CJ< z
```

При генерации кода для машины с кодами условий полезно использовать дескриптор кода условия. Этот дескриптор содержит имя, которое последним устанавливало код условия, либо пару сравнивавшихся имен, если код был установлен таким путем. Таким образом, мы можем реализовать

```
x := y + z
if x < 0 goto t
```

как

```
MOV y, R0
ADD z, R0
MOV R0, x
CJ< t
```

с учетом того, что код условия определялся переменной `x` после `ADD z, R0`.

## 9.7. Распределение и назначение регистров

Инструкции, включающие в качестве операндов только регистры, короче и выполняются быстрее, чем аналогичные инструкции с ячейками памяти в качестве операндов. Следовательно, эффективное использование регистров — важная составная часть генерации хорошего целевого кода. В этом разделе представлены различные стратегии определения того, какие значения в программе должны находиться в регистрах (распределение регистров) и в каком регистре должно храниться каждое значение (назначение регистров).

Один из подходов к распределению и назначению регистров состоит в выделении конкретных регистров для определенных значений в объектной программе. Например, решение может состоять в назначении базовых адресов одной группе регистров; арифметических вычислений — второй; вершины стека времени исполнения — некоторому фиксированному регистру и т.д.

Преимуществом такого подхода является упрощение разработки компилятора. Недостаток состоит в том, что слишком строгое следование правилу ограничивает эффективность использования регистров; ряд регистров на некотором участке кода может оставаться неиспользованным, в то время как из-за нехватки других регистров будут производиться излишние обмены значениями между регистрами и памятью. Все же в большинстве вычислительных сред имеет смысл зарезервировать несколько регистров в качестве базовых, указателя стека и т.п. и позволить компилятору использовать остальные регистры по своему усмотрению.

## Глобальное распределение регистров

Алгоритм генерации кода из раздела 9.6 использовал регистры для хранения значений во время выполнения одного базового блока. Однако все живые переменные в конце каждого блока сохранялись в памяти. Для того чтобы сэкономить на сохранениях в памяти и соответствующих им загрузках, мы можем назначить регистры часто используемым переменным и хранить их там при переходе границ между блоками (т.е. *глобально*). Поскольку программы тратят большую часть времени на выполнение внутренних циклов, естественным подходом к глобальному назначению регистров будет попытка хранить часто используемое значение в фиксированном регистре в процессе выполнения всего цикла. Предположим, что нам известна циклическая структура графа потока и мы знаем, какие значения, вычисляемые в базовом блоке, используются вне его. В следующей главе вы познакомитесь с технологиями получения этой информации.

Одна из стратегий глобального распределения регистров состоит в назначении некоторого фиксированного числа регистров для хранения наиболее активно используемых значений в каждом внутреннем цикле (эти значения могут отличаться в разных циклах). Нераспределенные после этого регистры могут использоваться для хранения локальных значений в блоке, как это делалось в разделе 9.6. Такой подход имеет тот недостаток, что фиксированное количество регистров не всегда оптимально для глобального распределения регистров. Однако этот метод прост в реализации и был использован в Fortran H, оптимизирующем компиляторе Fortran для машин серии IBM-360 [294].

В языках типа C и Bliss программист может выполнить часть работы по распределению регистров, используя соответствующее объявление для хранения переменных в регистрах во время выполнения процедуры. Разумное использование таких объявлений может ускорить выполнение программы, но программист не должен заниматься распределением регистров до того, как выполнит профилирование программы<sup>7</sup>.

## Счетчики использований

Простой метод определения экономии, получаемой при хранении переменной  $x$  в регистре в процессе выполнения цикла  $L$ , состоит в том, что в нашей модели машины мы экономим единицу стоимости кода для каждого обращения к  $x$ , если эта переменная находится в регистре. Однако при использовании способа генерации кода для блока из предыдущего раздела весьма вероятно, что вычисленное в блоке значение  $x$  останется в регистре, если используется далее в этом блоке. Таким образом, мы подсчитываем сум-

<sup>7</sup> Кроме того, объявление переменных в качестве регистровых является всего лишь пожеланием программиста, но никак не директивой компилятору, который при генерации кода может пренебречь этими объявлениями (например, когда регистровыми объявлено больше переменных, чем имеется регистров у целевой машины). — Прим. ред.

марную экономию от хранения  $x$  в регистре, прибавляя единицу за каждое использование  $x$  в цикле  $L$ , которому не предшествует присвоение  $x$  в том же блоке. Кроме того, мы сэкономим две единицы, если избежим сохранения  $x$  в конце блока. Таким образом, если  $x$  хранится в регистре, мы добавляем к суммарной экономии по две единицы для каждого блока  $L$ , в котором  $x$  получает значение и остается живым при выходе из блока.

Вместе с тем, если переменная  $x$  является живой при входе в заголовок цикла, мы должны загрузить ее в регистр перед входом в цикл  $L$ . Такая загрузка стоит две единицы. Кроме того, для каждого из выходных блоков  $B$  цикла  $L$ , в которых  $x$  остается живым при входе в преемник блока  $B$  вне цикла  $L$ , мы должны сохранить значение  $x$ , что также стоит две единицы. Однако в предположении, что цикл выполняется много раз, этой потерей экономии можно пренебречь (она однократна при входе в цикл). Таким образом, приближенная формула для выгоды от выделения регистра для хранения переменной  $x$  в цикле  $L$  выглядит следующим образом:

$$\sum_{\text{Блоки } B \text{ в } L} (use(x, B) + 2 * live(x, B)), \quad (9.4)$$

где  $use(x, B)$  — число использований  $x$  в  $B$  до любого определения  $x$ , а  $live(x, B)$  равно 1, если переменная  $x$  остается живой при выходе из блока  $B$ , и 0 — в противном случае. Отметим приближенность (9.4), поскольку не все блоки в цикле выполняются с одинаковой частотой. Кроме того, (9.4) основана на том, что цикл выполняется “много” раз. Для других машин можно разработать формулу, аналогичную (9.4), хотя, возможно, и отличающуюся от нее.

### Пример 9.6

Рассмотрим базовые блоки во внутреннем цикле, изображенном на рис. 9.13, на котором опущены инструкции безусловных и условных переходов. Предположим, что регистры R0, R1 и R2 выделены для хранения значений в цикле. Переменные, живые при входе в блок и выходе из него, показаны непосредственно над и под блоком. Здесь имеется несколько тонкостей, относящихся к живым переменным, о которых мы поговорим в главе 10, “Оптимизация кода”. Например, обратите внимание, что и  $e$ , и  $f$  живы при выходе из блока  $B_1$ , но при входе в блок  $B_2$  жива только переменная  $e$ , а при входе в блок  $B_3$  — только  $f$ . В целом, переменные, живые в конце блока, представляют собой объединение переменных, живых при входе в блоки-преемники.

При вычислении (9.4) для  $x = a$  заметим, что переменная  $a$  жива при выходе из  $B_1$  и получает в этом блоке свое значение, но не жива при выходе из блоков  $B_2$ ,  $B_3$  или  $B_4$ . Следовательно,  $\sum_{B \in L} 2 * live(a, B) = 2$ . Поскольку  $a$  определено в  $B_1$  до использования,

$use(a, B_1) = 0$ ; кроме того,  $use(a, B_2) = use(a, B_3) = 1$  и  $use(a, B_4) = 0$ . Таким образом,  $\sum_{B \in L} use(a, B) = 2$ . Следовательно, значение (9.4) для  $x = a$  равно 4, а значит, выбирая  $a$

для хранения в одном из глобальных регистров, мы получаем экономию в 4 единицы. Значения (9.4) для  $b$ ,  $c$ ,  $d$ ,  $e$  и  $f$  равны соответственно 6, 3, 6, 4 и 4, а потому для глобальных регистров R0, R1 и R2 мы можем выбрать переменные  $a$ ,  $b$  и  $d$ . Использование R0 для хранения  $e$  или  $f$  вместо  $a$  дает тот же конечный результат. На рис. 9.14 показан ассемблерный код, сгенерированный по рис. 9.13 с применением стратегии из раздела 9.6 для генерации кода каждого базового блока. Мы не приводим здесь код для опущенных на рис. 9.13 условных и безусловных переходов в конце каждого блока, а также не выводим сгенерированный код в качестве непрерывного потока, как это

осуществляется на практике. Стоит отметить, что если мы не будем строго придерживаться стратегии глобального резервирования R0, R1 и R2, то для  $B_2$  мы можем сгенерировать код

```
SUB R2, R0
MOV R0, f
```

и сохранить тем самым еще одну единицу стоимости кода, поскольку  $a$  не является живым при выходе из этого блока. Подобная экономия возможна и для блока  $B_3$ .  $\square$

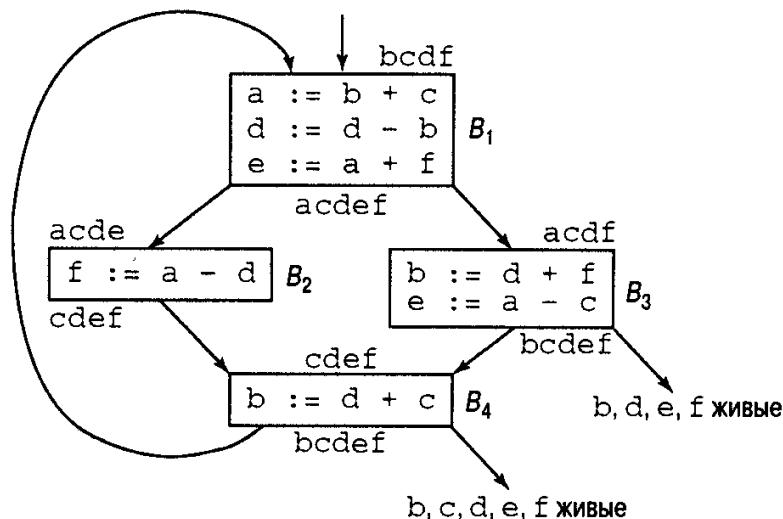


Рис. 9.13. Граф потока внутреннего цикла

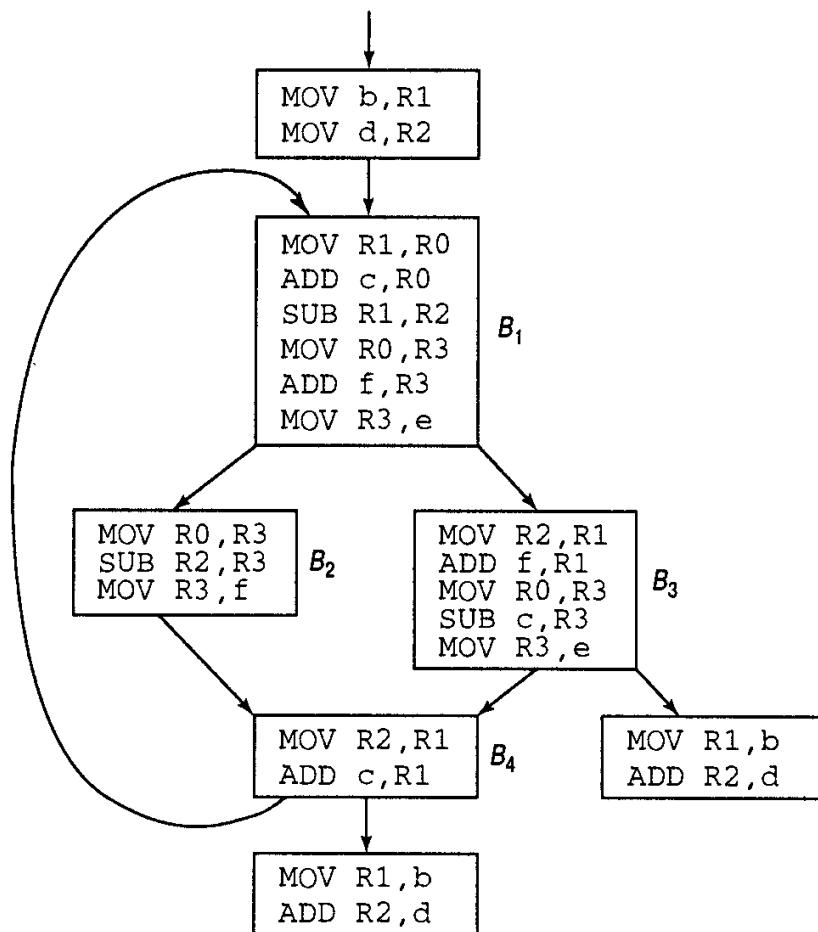


Рис. 9.14. Код, сгенерированный с использованием глобального распределения регистров

## Назначение регистров для внешних циклов

Назначив регистры и сгенерировав код для внутренних циклов, мы можем применить ту же идею и для больших циклов. Если внешний цикл  $L_1$  содержит внутренний цикл  $L_2$ , имена, хранящиеся в регистрах в  $L_2$ , не обязаны храниться в регистрах в  $L_1-L_2$ . Однако если имя  $x$  хранится в регистре в  $L_1$ , но не в  $L_2$ , мы должны принять меры по сохранению  $x$  при входе в  $L_2$  и загрузке его значения при выходе из  $L_2$  и входе в блок  $L_1-L_2$ . Аналогично, если мы храним  $x$  в регистре в  $L_2$ , но не в  $L_1$ , следует принять меры по загрузке  $x$  при входе в  $L_2$  и сохранению при выходе из него. Мы оставляем читателю в качестве упражнения вывод критерия для выбора имен, хранимых в регистрах во внешнем цикле  $L$ , по уже выбранным для хранения в регистрах именам во всех вложенных в  $L$  циклах.

## Распределение регистров путем раскраски графа

Когда для вычисления требуется регистр, а все доступные регистры используются, содержимое одного из них должно быть сохранено (*сброшено*, spilled) в ячейке памяти для его освобождения. Раскраска графа представляет собой простую технологию распределения регистров и управления их сбросом.

Этот метод требует двух проходов. При первом проходе инструкции целевой машины выбираются так, как будто у нас есть бесконечное множество символьических регистров; по сути, имена, используемые в промежуточном представлении, становятся именами регистров, а трехадресные инструкции — инструкциями машинного языка. Если обращение к переменным требует применения инструкций, использующих стековые указатели, указатели дисплея, базовые регистры или другие вспомогательные величины, мы предполагаем, что они также хранятся в регистрах, зарезервированных для этого. Обычно их использование непосредственно транслируется в режим обращения к адресам, использованным в машинных инструкциях. Если требуется более сложное обращение к адресу памяти, то оно разбивается на несколько машинных инструкций и, возможно, придется создать временный символьический регистр (или ряд таких регистров).

После выбора инструкций выполняется второй проход, в процессе которого и происходит назначение физических регистров символьическим. Цель назначения состоит в минимизации стоимости сбросов.

При втором проходе для каждой процедуры строится *граф взаимодействия регистров*, узлы которого представляют собой символьические регистры, а дуги соединяют два узла в том случае, если в момент определения одного из регистров второй оказывается жив. Например, граф взаимодействия регистров для представленного на рис. 9.13 кода будет иметь узлы для имен  $a$  и  $d$ . Поскольку в блоке  $B_1$  в момент определения  $d$  во второй инструкции  $a$  является живой переменной, эти узлы графа взаимодействия регистров будут соединены дугой.

Далее предпринимается попытка раскрасить граф  $k$  цветами, где  $k$  — количество доступных для назначения регистров (граф называется *раскрашенным*, если каждому его узлу назначен некоторый цвет таким образом, что никакие два соседних узла не имеют один и тот же цвет). Цвет в данном случае представляет регистр, а раскраска гарантирует, что никаких двух символьических регистра не будут мешать друг другу при назначении им одного и того же физического регистра.

Хотя в общем случае задача определения, может ли график быть раскрашен с использованием  $k$  цветов, является NP-полной, обычно на практике для раскраски графов используется простая и быстрая эвристическая технология. Предположим, что узел  $n$  графа  $G$

имеет меньше, чем  $k$  соседей (узлов, соединенных дугами с  $n$ ). Удалим  $n$  и его дуги из  $G$ , получив тем самым новый граф  $G'$ .  $k$ -раскраска  $G'$  решает задачу о  $k$ -раскраске  $G$ , так как теперь нам достаточно присвоить  $n$  цвет, не назначенный ни одному из его соседей.

Удаляя таким образом узлы с менее чем  $k$  соседями из графа взаимодействия регистров, мы получим либо пустой граф (в этом случае мы выполняем  $k$ -раскраску исходного графа, раскрашивая узлы в порядке, обратном порядку удаления), либо граф, в котором каждый узел имеет  $k$  или более смежных узлов. В этом случае  $k$ -раскраска графа невозможна, и требуется сброс узла путем введения кода для сохранения и перезагрузки регистра. Этот новый код приводит к изменению графа, и процесс раскраски продолжается. В [75] и [76] описан ряд эвристических приемов для выбора сбрасываемого регистра. Общее правило состоит в том, чтобы избегать добавления кода для сброса во внутренний цикл.

## 9.8. Представление базовых блоков в виде дага

Направленные ациклические графы (*даги*, см. раздел 5.2) представляют собой структуры данных, пригодные для реализации преобразований в базовых блоках. Даг дает картину того, каким образом значения, вычисляемые каждой инструкцией базового блока, используются далее в данном блоке. Построение дага по трехадресным инструкциям дает возможность определить общие подвыражения (выражения, вычисляемые более одного раза) в блоке, а также выяснить, какие имена вычисляются вне блока и используются в нем и какие инструкции в блоке могут вычислять значения, используемые вне блока.

*Даг базового блока* (далее просто даг) представляет собой направленный ациклический граф со следующими метками узлов.

1. Листья помечаются либо уникальными идентификаторами, либо именами переменных, либо константами. По оператору, применяемому к данному имени, мы определяем, требуется ли  $l$ - или  $r$ -значение; большинство листьев представляют  $r$ -значения. Листья представляют начальные значения имен, и мы помечаем их низшим индексом 0, дабы избежать путаницы с метками, обозначающими “текущие” значения имен (см. п. (3) ниже).
2. Внутренние узлы помечены символами операторов.
3. В качестве меток узлы могут также иметь последовательность идентификаторов. Смысл заключается в том, что внутренние узлы представляют вычисляемые значения, а идентификаторы, помечающие узел, считаются имеющими это значение.

Важно не путать даги и графы потоков. Каждый узел графа потока может быть представлен дагом, поскольку каждый узел графа потока обозначает не что иное, как базовый блок.

```
(1) t1 := 4 * i
(2) t2 := a [t1]
(3) t3 := 4 * i
(4) t4 := b [t3]
(5) t5 := t2 * t4
(6) t6 := prod + t5
(7) prod := t6
(8) t7 := i + 1
(9) i := t7
(10) if i <= 20 goto (1)
```

Рис. 9.15. Трехадресный код блока  $B_2$

### Пример 9.7

На рис. 9.15 показан трехадресный код, соответствующий блоку  $B_2$  на рис. 9.9. Инструкции пронумерованы, начиная с (1), исключительно для удобства. Соответствующий этому коду даг показан на рис. 9.16. Мы обсудим смысл дага после того, как приведем алгоритм его построения. Пока же заметим, что каждый узел дага представляет формулу, составленную в терминах листьев, т.е. значений переменных и констант при входе в блок. Например, узел, помеченный на рис. 9.16 как  $t_4$ , представляет формулу  $b[4*i]$ , т.е. значение слова по адресу со смещением  $4*i$  относительно адреса  $b$ , которое есть не что иное, как значение  $t_4$ .  $\square$

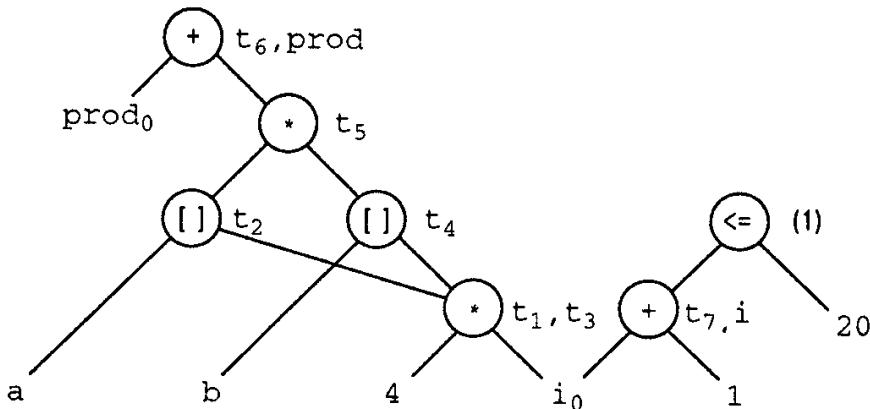


Рис. 9.16. Даг блока, представленного на рис. 9.15

### Построение дага

Для того чтобы построить даг базового блока, мы поочередно рассматриваем каждую инструкцию блока. Когда встречается инструкция вида  $x := y + z$ , мы ищем узлы, представляющие “текущие” значения  $y$  и  $z$ . Это могут быть как листья, так и внутренние узлы дага, если  $y$  и/или  $z$  вычислены предыдущими инструкциями блока. Затем мы создаем узел, помеченный как “+”, и даем ему два дочерних узла: левый представляет  $y$ , правый —  $z$ . Затем помечаем созданный узел как  $x$ . Однако если уже имеется узел, обозначающий то же значение  $y+z$ , мы добавляем не новый узел к дагу, а дополнительную метку  $x$  к этому существующему узлу.

Следует упомянуть о двух деталях. Во-первых, если имя  $x$  (не  $x_0$ ) ранее помечало некоторый другой узел, мы удаляем эту метку, поскольку “текущее” значение  $x$  представлено вновь созданным узлом. Во-вторых, для присвоения вида  $x := y$  мы не создаем новый узел. Вместо этого мы добавляем метку  $x$  к списку имен узла с “текущим” значением  $y$ .

Теперь приведем алгоритм для построения дага блока. Алгоритм очень похож на алгоритм 5.1, однако здесь к каждому узлу присоединен список идентификаторов. Мы должны предупредить читателя о том, что этот алгоритм может работать некорректно в случае присвоений массивам, косвенных присвоений с использованием указателей, а также если на одну ячейку памяти могут ссылаться два или более имен (в силу использования инструкции EQUIVALENCE или соответствия формальных и фактических параметров при вызове процедуры). В конце этого раздела мы рассмотрим изменения, которые необходимо внести в алгоритм для корректной работы в описанных ситуациях.

## Алгоритм 9.2. Построение дага

*Вход.* Базовый блок.

*Выход.* Даг базового блока, содержащий следующую информацию.

1. *Метку* для каждого узла. Для листа метка представляет собой идентификатор (допустимы также константы), а для внутреннего узла — символ оператора.
2. Список (возможно, пустой) присоединенных идентификаторов (константы не допустимы) для каждого узла.

*Метод.* Мы полагаем, что у нас есть необходимые структуры данных для создания узлов с одним или двумя дочерними, причем в последнем случае различаются “левый” и “правый” потомки. Кроме того, в структуре имеется место для меток узлов и возможность создания для каждого узла связанного списка идентификаторов, присоединенных к нему.

В дополнение к этим компонентам, нам необходимо поддерживать множество всех идентификаторов (включая константы), для которых имеются связанные с ними узлы. Узлом может быть как лист, помеченный идентификатором, так и внутренний узел с идентификатором в связанном с узлом списке. Мы считаем, что имеется функция  $\text{node}(\text{identifier})$ , которая в процессе построения дага возвращает последний созданный узел, связанный с  $\text{identifier}$ . Интуитивно  $\text{node}(\text{identifier})$  является узлом дага, представляющим значение, которое  $\text{identifier}$  имеет в данный момент процесса построения дага. На практике значение  $\text{node}(\text{identifier})$  может указывать запись в таблице символов для  $\text{identifier}$ .

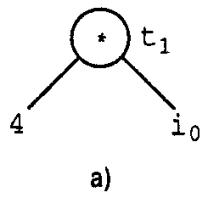
Процесс построения дага состоит в выполнении приведенных ниже шагов (1)–(3) для каждой инструкции в блоке. Изначально мы считаем, что даг не имеет узлов, и для любого аргумента значение функции  $\text{node}$  не определено. Предположим, что “текущей” трехадресной инструкцией может быть одна из трех: (i)  $x := y \text{ op } z$ , (ii)  $x := \text{op } y$  или (iii)  $x := y$ <sup>8</sup>. Оператор отношения типа  $\text{if } i \leq 20 \text{ goto}$  считается принадлежащим к виду (i) с неопределенным  $x$ .

1. Если значение  $\text{node}(y)$  не определено, создаем лист, помеченный  $y$ , и считаем, что  $\text{node}(y)$  является этим узлом. В случае (i) то же самое делаем при неопределенности  $\text{node}(z)$ .
2. В случае (i) определим, существует ли узел, помеченный  $\text{op}$ , левый потомок которого представляет собой  $\text{node}(y)$ , а правый —  $\text{node}(z)$  (тем самым выявляется наличие общих подвыражений). Если такого узла нет, создаем его. В любом случае пусть  $n$  — интересующий нас (найденный или созданный) узел. В случае (ii) попытаемся найти узел  $n$ , помеченный  $\text{op}$ , единственным потомком которого является  $\text{node}(y)$ . Если такого узла нет, создадим его. В случае (iii) пусть  $n$  представляет собой  $\text{node}(y)$ .
3. Удалим  $x$  из списка идентификаторов, присоединенного к  $\text{node}(x)$ , и добавим  $x$  к списку идентификаторов узла  $n$ , найденного при выполнении п. (2). Установим  $\text{node}(x)$  равным  $n$ . □

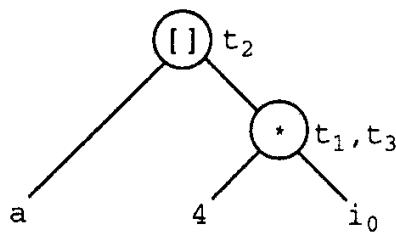
<sup>8</sup> Предполагается, что операторы имеют не более двух аргументов. Обобщение для трех и более аргументов очевидно.

### Пример 9.8

Вернемся к блоку, представленному на рис. 9.15, и проследим построение его дага, показанного на рис. 9.16. Первой инструкцией является  $t_1 := 4 * i_0$ . При выполнении шага (1) мы должны создать листья, помеченные как 4 и  $i_0$  (мы используем индекс 0, чтобы отличать на рисунке метки от присоединенных идентификаторов, но на самом деле индекс частью метки не является). При выполнении шага (2) мы создаем узел, помеченный “\*”, а на шаге (3) присоединяем к нему идентификатор  $t_1$ . Рис. 9.17 $a$  показывает даг, построенный на этом этапе.



а)



б)

Рис. 9.17. Стадии построения дага

Для второй инструкции  $t_2 := a[t_1]$  мы создаем новый лист, помеченный  $a$ , и находим созданный перед этим узел  $node(t_1)$ . Мы также создаем новый узел, помеченный  $[ ]$ , к которому присоединяем узлы для  $a$  и  $t_1$  в качестве дочерних.

Для третьей инструкции  $t_3 = 4 * i$  мы находим, что  $node(4)$  и  $node(i)$  уже существуют. Поскольку в инструкции используется оператор “\*”, мы не создаем для нее новый узел, а просто добавляем  $t_3$  к списку идентификаторов для узла  $t_1$ . Полученный в результате даг показан на рис. 9.17б. Чтобы быстро определить существование узла для  $4 * i$ , можно воспользоваться приведенным в разделе 5.2 методом номера значения.

Мы предлагаем читателю самостоятельно завершить построение дага. Упомянем лишь шаги, предпринимаемые для девятой инструкции  $i := t_7$ . Перед инструкцией (9) значением  $node(i)$  является лист с пометкой  $i_0$ . Инструкция (9) относится к типу (iii); таким образом, мы находим  $node(t_7)$ , присоединяем  $i$  к списку идентификаторов и устанавливаем  $node(i)$  равным  $node(t_7)$ . Это одна из двух инструкций (вторая — инструкция (7)), для которой значение  $node$  изменяется для идентификатора. Это изменение гарантирует, что новый узел для  $i$  окажется левым дочерним по отношению к узлу оператора  $<=$ , построенного для инструкции (10). □

### Применение дагов

При выполнении алгоритма 9.2 мы можем получить различную полезную информацию. Во-первых, заметим, что мы автоматически определяем общие подвыражения. Во-вторых, мы можем определить, значения каких идентификаторов используются в блоке;

это те идентификаторы, для которых в процессе работы были созданы листья на шаге (1) алгоритма 9.2. В-третьих, мы можем определить, какие инструкции вычисляют значения, которые могут использоваться вне блока. Это те инструкции  $S$ , для чьих узлов  $n$ , построенных или найденных на шаге (2), в конце построения дага остается справедливым соотношение  $\text{node}(x) = n$  (здесь  $x$  обозначает идентификатор, присвоенный инструкцией  $S$ ) или, что то же самое,  $x$  остается идентификатором, присоединенным к  $n$ .

### Пример 9.9

В примере 9.8 все инструкции отвечают приведенным ограничениям, поскольку при переопределении  $\text{node}$  (для  $\text{prod}$  и  $i$ ) предыдущее значение  $\text{node}$  представляло собой лист. Таким образом, значения всех внутренних узлов могут использоваться вне блока. Теперь предположим, что перед инструкцией (9) мы вставили новую инструкцию  $s$ , которая присваивает значение переменной  $i$ . При рассмотрении этой инструкции мы создаем узел  $t$  и устанавливаем  $\text{node}(i) = t$ . Однако в инструкции (9) мы переопределим  $\text{node}(i)$ . Следовательно, вычисленное инструкцией  $s$  значение не может использоваться вне блока.  $\square$

Еще одно важное использование дагов — при построении упрощенного списка четверок с учетом наличия общих подвыражений и устранения излишних присвоений типа  $x := y$ . Когда присоединенный список идентификаторов узла содержит более одного идентификатора, мы выясняем, какие из этих идентификаторов нужны вне блока. Как мы упоминали, поиск живых переменных в конце блока требует анализа потока данных, именуемого “анализом активных переменных” и обсуждаемого в главе 10, “Оптимизация кода”. Однако во многих случаях мы можем считать, что ни одно из временных имен типа  $t_1, t_2, \dots, t_7$  на рис. 9.15 не понадобится вне блока (но будьте осторожны при трансляции логических выражений; одно выражение может распространяться на несколько базовых блоков).

Вообще говоря, вычислять внутренние выражения дага можно в любом порядке, представляющем собой топологическую сортировку дага. При топологической сортировке узел не вычисляется до тех пор, пока не будут вычислены все его дочерние узлы (являющиеся внутренними). При вычислении узла мы присваиваем полученное значение одному из присоединенных идентификаторов  $x$ , предпочтительно тому, который используется вне блока. Мы не можем, однако, выбирать  $x$ , если имеется другой узел  $t$ , значение которого также сохранено в  $x$ <sup>9</sup>, и при этом  $t$  вычислено и остается живым. Здесь мы определяем узел  $t$  как живой, если его значение необходимо вне блока или  $t$  имеет все еще невычисленный родительский узел.

Если узел  $n$  имеет дополнительные присоединенные идентификаторы  $y_1, y_2, \dots, y_k$ , значения которых используются вне блока, мы присваиваем им значения с помощью инструкций  $y_1 := x, y_2 := x, \dots, y_k := x$ . Если  $n$  не имеет присоединенных идентификаторов (что может случиться, например, когда  $n$  создается посредством присвоения переменной  $x$ , которая затем переприсваивается), мы создаем новое временное имя для хранения значения  $n$ . Мы должны предупредить читателя, что при наличии присвоений с использованием указателей или массивов разрешена не любая топологическая сортировка дага; мы вкратце рассмотрим этот вопрос позднее.

---

<sup>9</sup> Согласно правилу (3) из алгоритма 9.2, описываемая ситуация невозможна. — Прим. ред.

## Пример 9.10

Перестроим базовый блок для дага, приведенного на рис. 9.16, упорядочивая узлы в порядке их создания:  $t_1, t_2, t_4, t_5, t_6, t_7, (1)$ . Заметим, что инструкции (3) и (7) исходного блока не создают новые узлы, но добавляют метки  $t_3$  и  $\text{prod}$  к спискам идентификаторов узлов  $t_1$  и  $t_6$  соответственно. Мы полагаем, что ни одна из временных переменных  $t_i$  не нужна вне блока.

Мы начинаем работу с узла, представляющего  $4 * i$ . Этот узел имеет два присоединенных идентификатора —  $t_1$  и  $t_3$ . Используем для хранения значения  $4 * i$  идентификатор  $t_1$ , так что первая инструкция нового блока будет выглядеть как

```
 $t_1 := 4 * i$
```

т.е. как и в исходном базовом блоке. Второй рассматриваемый узел помечен  $t_2$ , а инструкция, созданная на основании этого узла, выглядит как

```
 $t_2 := a [t_1]$
```

т.е. как и ранее. Следующим рассматривается узел  $t_4$ , для которого генерируется инструкция

```
 $t_4 := b [t_1]$
```

Эта инструкция в качестве аргумента использует  $t_1$  (в то время как инструкция исходного блока использовала переменную  $t_3$ ), поскольку именно  $t_1$  была выбрана для хранения значения  $4 * i$ .

Теперь рассмотрим узел  $t_5$  и сгенерируем инструкцию

```
 $t_5 := t_2 * t_4$
```

Для узла, помеченного как  $t_6$ ,  $\text{prod}$ , мы выбираем для хранения значения переменную  $\text{prod}$ , поскольку, вероятно, именно этот идентификатор, а не  $t_6$ , может быть использован вне блока. Так же, как и  $t_3$ , идентификатор  $t_6$  исчезает из кода, и следующая сгенерированная инструкция представляет собой

```
 $\text{prod} := \text{prod} + t_5$
```

Аналогично для хранения значения  $i + 1$  мы выбираем  $i$ , а не  $t_7$ . Последние сгенерированные инструкции будут следующими.

```
 $i := i + 1$
 $\text{if } i \leq 20 \text{ goto } (1)$
```

Обратите внимание, что десять инструкций на рис. 9.15 сократились до семи за счет использования информации об общих подвыражениях, полученной при построении дага, и удаления необязательных присвоений.  $\square$

## Массивы, указатели и вызовы процедур

Рассмотрим следующий базовый блок.

```
x := a[i]
a[j] := y
z := a[i]
```

(9.5)

Если мы используем алгоритм 9.2 для построения дага для (9.5),  $a[i]$  станет общим подвыражением, и “оптимизированный” блок превратится в

```
x := a[i]
z := x
a[j] := y
```

(9.6)

Однако (9.5) и (9.6) вычисляют различные значения  $z$  в случае  $i=j$  и  $y \neq a[i]$ . Проблема заключается в том, что когда мы выполняем присвоение элементу массива  $a$ , мы можем изменить  $g$ -значение выражения  $a[i]$  при неизменных  $a$  и  $i$ . Таким образом, необходимо, чтобы при обработке присвоения элементу массива  $a$  мы уничтожили (kill<sup>10</sup>) все узлы, помеченные [], левые аргументы которых представляют собой  $a$  плюс или минус константа (возможно, нуль)<sup>11</sup>. Тем самым мы делаем эти узлы недоступными для добавления дополнительных меток-идентификаторов, что, в свою очередь, защищает их от ошибочного распознавания в качестве общих подвыражений. Таким образом, для каждого узла мы должны иметь бит, указывающий, уничтожен ли данный узел. Далее, для каждого массива  $a$ , упоминаемого в блоке, удобно иметь список всех узлов, которые в настоящий момент не уничтожены, но должны быть уничтожены при обработке присвоения значения элементу массива  $a$ .

Подобная проблема возникает и при присвоениях типа  $*p := w$ , где  $p$  является указателем. Если нам неизвестно, на что может указывать  $p$ , каждый узел, который в настоящее время создается в даге, должен быть уничтожен в описанном выше смысле. Если помеченный как  $a$  узел  $n$  уничтожен и имеется последующее присвоение  $a$ , мы должны создать новый лист для  $a$  и использовать его, а не  $n$ . Позже мы рассмотрим ограничения на порядок вычислений, вызываемые уничтожением узлов.

В главе 10, “Оптимизация кода” мы обсудим методы, которые позволяют определить, что  $p$  может указывать только на некоторое подмножество идентификаторов. Если  $p$  может указывать только на  $r$  и  $s$ , должны быть уничтожены только  $node(r)$  и  $node(s)$ . Возможно также, что мы сможем определить, что выполнение условия  $i=j$  в блоке (9.5) невозможно, так что узел для  $a[i]$  не должен быть уничтожен при обработке присвоения  $a[j] := y$ . Однако обычно этот тип анализа не оправдывает затраченных усилий.

Вызов процедуры в базовом блоке уничтожает все узлы, поскольку при отсутствии знаний о вызываемой процедуре мы вынуждены предположить наихудшее — что любая переменная может быть изменена из-за побочного действия процедуры. В главе 10, “Оптимизация кода” объясняется, каким образом можно установить, что некоторые переменные не изменяются при вызове процедуры; соответственно, узлы для таких идентификаторов могут не уничтожаться.

Если мы намерены заново собрать базовый блок на основе дага и не хотим использовать порядок, в котором создавались узлы дага, то необходимо указать в даге, что некоторые кажущиеся независимыми узлы должны вычисляться в определенном порядке. Например, в (9.5) инструкция  $z := a[i]$  должна следовать за  $a[j] := y$ , а та в свою очередь — за  $x := a[i]$ . Введем в даг некоторые дуги  $n \rightarrow m$ , которые указывают не то, что  $m$  является аргументом  $n$ , а то, что при любом вычислении дага вычисление  $n$  должно осуществляться после вычисления  $m$ . При этом должны выполняться следующие правила.

1. Любое вычисление или присвоение элемента массива должно следовать за предшествующим присвоением элементу массива, если таковое имело место.
2. Любое присвоение элементу массива должно следовать за любым предшествующим вычислением  $a$ .

<sup>10</sup> Более точным было бы использование термина “убили”, но такой перевод перекликается с “живыми” переменными и может внести определенную путаницу; перевод “удалили” для дага и графов вообще имеет совершенно иной и вполне определенный смысл. — Прим. перев.

<sup>11</sup> Заметим, что аргумент [], указывающий имя массива, может быть самим  $a$  либо выражением типа  $a - 4$ . В последнем случае узел  $a$  является не дочерним узлом, а потомком дочернего по отношению к [] узла.

3. Любое использование идентификатора должно следовать за предшествующим вызовом процедуры либо косвенным присвоением с использованием указателя, если такое имеется.
4. Любой вызов процедуры или косвенное присвоение с использованием указателя должны следовать за всеми предшествующими вычислениями любых идентификаторов.

Таким образом, при переупорядочении кода ни одна инструкция, использующая массив *a*, не должна пересекать другую использующую *a* инструкцию. Кроме того, ни одна инструкция не должна пересечь вызов процедуры либо косвенное присвоение с использованием указателя.

## 9.9. Локальная оптимизация

Стратегия генерации кода “инструкция за инструкцией” часто приводит к целевому коду, содержащему излишние инструкции и не совсем оптимальные конструкции. Качество такого целевого кода может быть улучшено с помощью “оптимизирующих” преобразований целевой программы. Термин “оптимизирующий” несколько обманчив, поскольку нет никакой гарантии, что полученный в результате код оптimalен с точки зрения какого-либо математического критерия. Но несмотря на это многие простые преобразования могут значительно улучшить время выполнения или размер целевой программы, поэтому важно знать, какие типы преобразований пригодны на практике.

Простой, но эффективной технологией локального улучшения целевого кода является *локальная оптимизация* (reephole optimization<sup>12</sup>), которая представляет собой метод улучшения характеристик целевой программы путем рассмотрения коротких последовательностей целевых инструкций и замены их по возможности более короткими или быстрыми. Хотя мы рассматриваем локальную оптимизацию как технологию для улучшения целевого кода, ее также можно применить сразу после генерации промежуточного кода для улучшения промежуточного представления.

Локальная оптимизация использует небольшое перемещающееся по целевой программе окно, код в котором не обязан быть непрерывным, хотя некоторые реализации и выдвигают такое требование. Характерным для локальной оптимизации является то, что каждое улучшение кода может создать условия для дополнительных улучшений. Вообще говоря, для получения максимального эффекта необходимы повторяющиеся проходы по целевому коду. В этом разделе мы приведем следующие примеры преобразования программ, характеризующие локальную оптимизацию:

- устранение излишних инструкций,
- оптимизация потока управления,
- алгебраические упрощения,
- использование машинных идиом.

### Излишние загрузки и сохранения

Рассмотрим следующую последовательность инструкций.

$$(1) \quad \text{MOV} \quad R0, \quad a \\ (2) \quad \text{MOV} \quad a, \quad R0 \quad (9.7)$$

---

<sup>12</sup> Дословно — оптимизация смотрового отверстия. — Прим. перев.

Здесь можно удалить инструкцию (2), поскольку при ее выполнении инструкция (1) гарантирует, что значение *a* уже находится в регистре R0. Заметим, что если бы (2) имела метку<sup>13</sup>, то мы не могли быть уверены, что (1) всегда выполняется немедленно перед (2), и, таким образом, не могли бы удалить инструкцию (2) из кода. Иными словами, инструкция (2) может быть безопасно удалена из кода, если и (1), и (2) находятся в одном базовом блоке.

Хотя код типа (9.7) и не может быть получен при использовании алгоритма, предложенного в разделе 9.6, он вполне может быть сгенерирован более простым алгоритмом, вроде упоминаемого в начале раздела 9.1.

## Недостижимый код

Вторая возможность локальной оптимизации заключается в удалении недостижимых инструкций. Инструкция без метки, следующая непосредственно за безусловным переходом, может быть удалена (такое удаление можно повторить для целой последовательности инструкций). Например, в целях отладки большая программа может использовать фрагменты кода, которые выполняются только тогда, когда переменная *debug* равна 1. Например, в С исходный код может выглядеть следующим образом<sup>14</sup>.

```
#define debug 0
.
if (debug) {
 Вывод отладочной информации
}
```

Промежуточное представление этого кода может выглядеть следующим образом.

```
if debug = 1 goto L1
goto L2
L1: Вывод отладочной информации
L2:
(9.8)
```

Один из очевидных вариантов локальной оптимизации состоит в удалении переходов через инструкции перехода. Следовательно, независимо от значения переменной *debug* (9.8) может быть заменено следующим кодом.

```
if debug ≠ 1 goto L2
 Вывод отладочной информации
L2:
(9.9)
```

Теперь, поскольку в начале программы значение *debug* установлено равным 1<sup>15</sup>, оптимизация, состоящая в распространении констант, приведет к замене (9.9) следующим кодом.

```
if 0 ≠ 1 goto L2
 Вывод отладочной информации
L2:
(9.10)
```

<sup>13</sup> Одно из преимуществ генерации ассемблерного кода состоит в наличии меток, упрощающих подобную локальную оптимизацию. Если сгенерирован машинный код и желательно выполнение локальной оптимизации, можно использовать бит, помечающий инструкции, имеющие метки.

<sup>14</sup> Следует заметить, что в приведенном ниже коде *debug* является не переменной, а макросом (что, впрочем, не меняет сути последующего рассмотрения). — Прим. ред.

<sup>15</sup> Для определения того, что переменная *debug* имеет значение 0, нам необходимо выполнить глобальный анализ “определения достижимости”, о чем будет рассказано в главе 10. (Заметим, что в силу макроопределения *debug* в данном конкретном случае такой анализ не нужен. — Прим. ред.)

Поскольку вычисление аргумента первой инструкции в (9.10) дает значение `true`, инструкция может быть заменена безусловным переходом `goto L2`. После этого становится очевидно, что все инструкции вывода отладочной информации недостижимы и могут быть удалены из программы.

## Оптимизация потока управления

Алгоритмы генерации промежуточного кода из главы 8, “Генерация промежуточного кода”, часто приводят к коду, в котором имеются условные и безусловные переходы к инструкциям-переходам. Такие излишние переходы могут быть устранины либо в промежуточном представлении, либо в целевом коде путем локальной оптимизации следующих типов. Мы можем заменить последовательность переходов

```
 goto L1
 . .
L1: goto L2
последовательностью
 goto L2
 . .
L1: goto L2
```

Если после такой замены отсутствуют переходы к  $L1^{16}$ , инструкцию `L1: goto L2` можно убрать из целевого кода, если ей предшествует инструкция безусловного перехода. Аналогично последовательность

```
 if a < b goto L1
 . .
L1: goto L2
может быть заменена следующей последовательностью.
if a < b goto L2
 . .
L1: goto L2
```

И наконец, предположим, что есть только один переход к  $L1$  и что  $L1$  предшествует безусловный переход. Тогда последовательность

```
 goto L1
 . .
L1: if a < b goto L2
L3:
(9.11)
```

может быть заменена следующей последовательностью.

```
if a < b goto L2
 goto L3
 . .
L3:
(9.12)
```

Хотя число инструкций в (9.11) и (9.12) и одинаково, иногда мы можем пропустить безусловный переход в (9.12), но никогда — в (9.11). Таким образом, код (9.12) превосходит (9.11) с точки зрения времени выполнения.

---

<sup>16</sup> При использовании такой локальной оптимизации мы можем отслеживать количество переходов к каждой метке в записи таблицы символов для этой метки; при этом просмотр всего кода не нужен.

## Алгебраические упрощения

Количество вариантов алгебраических упрощений, которые могут быть предприняты при локальной оптимизации, практически бесконечно. Однако только несколько алгебраических тождеств встречаются достаточно часто, чтобы стоило рассматривать их реализацию. Например, инструкции типа  $x := x + 0$  или  $x := x * 1$  достаточно часто генерируются простейшими алгоритмами генерации промежуточного кода и легко могут быть удалены при выполнении локальной оптимизации.

## Снижение стоимости

Снижение стоимости заменяет дорогие (с точки зрения времени выполнения, используемых ресурсов и т.п.) инструкции целевой машины более дешевыми, дающими тот же результат. Ряд машинных инструкций значительно дешевле других и часто используется как частный случай более дорогих операторов. Например,  $x^2$  всегда дешевле вычислить как  $x * x$ , чем путем вызова подпрограммы возведения в степень. Умножение или деление чисел с фиксированной точкой на степень двойки дешевле реализовать как сдвиг. Деление чисел с плавающей точкой на константу, реализованное как умножение на другую константу, также может оказаться более дешевым.

## Использование машинных идиом

Целевая машина может иметь аппаратные инструкции для эффективной реализации ряда специфических операций. Определение ситуаций, позволяющих использовать специальные машинные инструкции, может существенно снизить время выполнения кода. Например, ряд машин имеет режимы адресации с автоувеличением или автоумножением, при которых к операнду прибавляется (или вычитается из него) единица до или после использования его значения. Использование таких режимов существенно повышает качество кода для внесения в стек или снятия с него, выполняемых, например, при передаче параметров процедуре. Эти режимы могут также использоваться в коде для инструкций типа  $i := i + 1$ .

## 9.10. Генерация кода на основе дагов

В этом разделе мы покажем, каким образом генерируется код базового блока по его представлению в виде дага. Преимущество этого метода заключается в том, что по дагу можно проще определить, каким образом переупорядочить окончательную последовательность вычислений, чем при рассмотрении линейной последовательности трехадресных инструкций или четверок. Центральным в нашем рассмотрении является случай, когда даг представляет собой дерево. В этом случае мы можем сгенерировать код, для которого можно доказать его оптимальность с точки зрения такого критерия, как размер программы или минимальное количество используемых временных переменных. Этот алгоритм генерации оптимального кода на основе дерева применим также в том случае, когда промежуточный код представляет собой дерево разбора.

## Переупорядочение

Обсудим вкратце влияние порядка вычислений на стоимость получаемого объектного кода. Рассмотрим следующий базовый блок, даг которого показан на рис. 9.18 и является деревом.

```
t1 := a + b
t2 := c + d
t3 := e - t2
t4 := t1 - t3
```

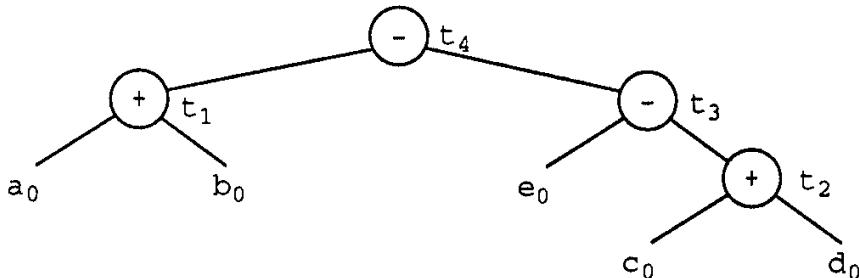


Рис. 9.18. Даг базового блока

Заметим, что порядок вычислений в блоке естественным образом получен при синтаксически управляемой трансляции выражения  $(a+b)-(e-(c+d))$  с помощью алгоритма из раздела 8.3.

Если для этих трехадресных инструкций мы сгенерируем код с помощью алгоритма из раздела 9.6, то получим последовательность инструкций, показанную на рис. 9.19 (в предположении, что доступны только два регистра ( $R0$  и  $R1$ ) и при выходе из блока жива только переменная  $t_4$ ).

```
MOV a, R0
ADD b, R0
MOV c, R1
ADD d, R1
MOV R0, t1
MOV e, R0
SUB R1, R0
MOV t1, R1
SUB R0, R1
MOV R1, t4
```

Рис. 9.19. Код базового блока

Предположим, что мы изменили порядок инструкций так, что вычисление  $t_1$  выполняется непосредственно перед вычислением  $t_4$ .

```
t2 := c + d
t3 := e - t2
t1 := a + b
t4 := t1 - t3
```

В этом случае при генерации кода в соответствии с алгоритмом из раздела 9.6 мы получим код, показанный на рис. 9.20 (при тех же предположениях, что и ранее). Выполняя вычисления в новом порядке, мы экономим две инструкции —  $MOV R0, t_1$  и  $MOV t_1, R1$ .

```

MOV c, R0
ADD d, R0
MOV e, R1
SUB R0, R1
MOV a, R0
ADD b, R0
SUB R1, R0
MOV R0, t4

```

*Рис. 9.20. Код преобразованного базового блока*

## Эвристическое упорядочение дагов

Причина, по которой рассмотренное переупорядочение улучшило код, заключается в том, что вычисление  $t_4$  следует теперь непосредственно за вычислением  $t_1$ , левого операнда  $t_4$  в дереве. То, что такая перестановка выгодна, должно быть совершенно ясно: для эффективного вычисления  $t_4$  его левый аргумент должен находиться в регистре, а вычисление  $t_1$  непосредственно перед  $t_4$  гарантирует, что это условие будет выполнено.

При выборе порядка вычисления узлов дага мы ограничены только требованием, чтобы выбранный нами порядок вычислений сохранял взаимоотношения, заданные дугами в даге. Вспомните из раздела 9.8, что эти дуги могут представлять либо взаимоотношение оператор-операнд, либо косвенные ограничения, вытекающие из возможного взаимодействия вызовов процедур и присвоений с использованием массивов и указателей. Мы предлагаем следующий эвристический алгоритм, который пытается по возможности выполнять вычисления узла немедленно после вычисления его левого аргумента. Алгоритм, приведенный на рис. 9.21, дает обратный порядок узлов.

- (1) **while** Остаются неперечисленные внутренние узлы **do begin**
- (2)     Выбираем неперечисленный узел  $n$ , все родители которого уже перечислены
- (3)     **list**  $n$ ;
- (4)     **while** Левый дочерний узел  $m$  узла  $n$   
       не имеет неперечисленных родителей  
       и не является листом **do**  
       /\* Поскольку  $n$  только что перечислен,  
        $m$  остается неперечисленным \*/  
       **begin**
- (5)             **list**  $m$ ;
- (6)              $n := m$
- end**
- end**

*Рис. 9.21. Алгоритм перечисления узлов*

## Пример 9.11

Алгоритм, приведенный на рис. 9.21, будучи применен к дереву на рис. 9.18, приводит к коду, показанному на рис. 9.20. В качестве более полного примера рассмотрим даг на рис. 9.22.

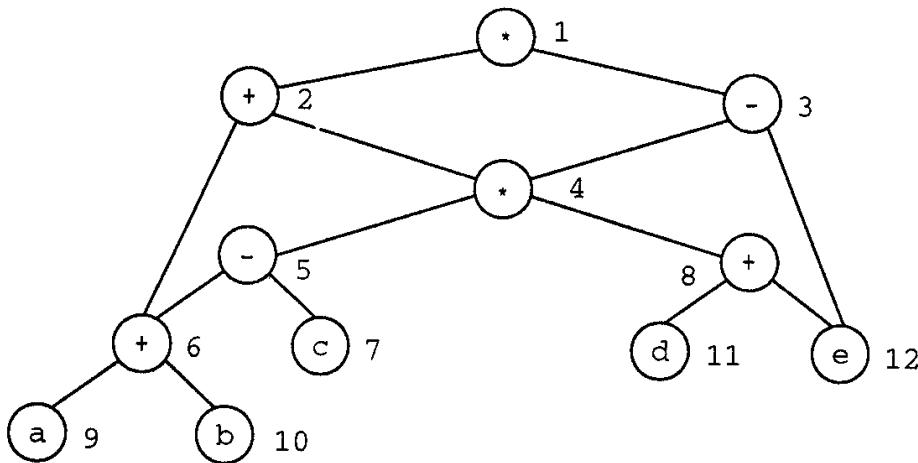


Рис. 9.22. Пример дага

Изначально единственным узлом, у которого нет неперечисленных родителей, является узел 1; так что в строке (2) мы устанавливаем  $n = 1$  и перечисляем узел 1 в строке (3). Теперь левый аргумент узла 1 — узел 2 — имеет перечисленного родителя, так что мы перечисляем 2 и устанавливаем  $n = 2$  в строке (6). После этого в строке (4) мы находим левого потомка узла 2, которым является узел 6, имеющий неперечисленного родителя 5. Следовательно, в строке (2) мы выбираем новое значение  $n$ , причем единственным кандидатом является узел 3. Перечисляем 3 и следуем далее по левой цепочке, перечисляя узлы 4, 5 и 6. После этого среди неперечисленных внутренних узлов остается только узел 8, так что перечисляем и его. В результате мы получаем список 1234568, так что предлагаемый алгоритмом порядок вычисления — 8654321. Такой порядок соответствует следующей последовательности трехадресных инструкций.

```

 $t_8 := d + e$
 $t_6 := a + b$
 $t_5 := t_6 - c$
 $t_4 := t_5 * t_8$
 $t_3 := t_4 - e$
 $t_2 := t_6 + t_4$
 $t_1 := t_2 * t_3$

```

Данная последовательность при использовании алгоритма из раздела 9.6 приводит к оптимальному коду для дага на нашей машине с любым количеством регистров. Следует отметить, что в этом примере наш эвристический метод ни разу не столкнулся с выбором на шаге (2), хотя в общем случае здесь может быть много вариантов.  $\square$

## Оптимальное упорядочение для деревьев

Оказывается, что для модели машины из раздела 9.2 мы можем дать простой алгоритм определения оптимального порядка вычисления инструкций базового блока, когда даг блока является деревом. Оптимальный порядок в данном случае означает порядок, который приводит к кратчайшей последовательности инструкций среди всех последовательностей, вычисляющих дерево. Этот алгоритм, модифицированный с учетом пар регистров и других особенностей целевых машин, используется, например, в компиляторах Algol, Bliss и C.

Данный алгоритм состоит из двух частей. Первая часть помечает все узлы дерева снизу вверх целым числом, которое обозначает минимальное количество регистров,

необходимых для вычисления дерева без записи в память промежуточных результатов. Вторая часть алгоритма представляет собой обход дерева в порядке, определяемом вычисленными метками узлов. Выходной код генерируется в процессе обхода дерева.

Интуитивно алгоритм вначале вычисляет тот операнд бинарного оператора, который требует большего количества регистров (более трудный операнд). Если требования к количеству регистров у операндов одинаковы, выбор операнда, вычисляемого первым, произволен.

## Алгоритм назначения меток

Мы используем термин “левый лист” для обозначения узла, являющегося листом и одновременно крайним слева потомком своего родителя. Все остальные листья назовем “правыми листьями”.

Назначение меток может быть выполнено путем посещения узлов в восходящем порядке; узел не посещается до тех пор, пока все его потомки не будут помечены. Порядок, в котором создаются узлы дерева разбора, подходит для нашей цели, если это дерево используется в качестве промежуточного кода, так как в этом случае метки могут быть вычислены путем синтаксически управляемой трансляции. На рис. 9.23 приведен алгоритм для вычисления метки в узле  $n$ . В важном специальном случае, когда  $n$  является бинарным узлом и его потомки имеют метки  $l_1$  и  $l_2$ , формула в строке (6) сводится к следующей:

$$label(n) = \begin{cases} \max(l_1, l_2) & \text{если } l_1 \neq l_2 \\ l_1 + 1 & \text{если } l_1 = l_2 \end{cases}$$

```
(1) if n является листом then
(2) if n является крайним слева потомком then
(3) label(n) := 1
(4) else label(n) := 0
(5) else begin /* n - внутренний узел */
 Пусть n_1, n_2, \dots, n_k - потомки n ,
 упорядоченные по значениям label так, что
 $label(n_1) \geq label(n_2) \geq \dots \geq label(n_k)$;
(6) label(n) := $\max_{1 \leq i \leq k} (label(n_i) + i - 1)$
end
```

Рис. 9.23. Вычисление меток

## Пример 9.12

Рассмотрим дерево на рис. 9.18. Обход узлов в обратном порядке<sup>17</sup> приводит к следующей очередности:  $a\ b\ t_1\ e\ c\ d\ t_2\ t_3\ t_4$ . Обратный порядок всегда пригоден для вычисления меток. Узел  $a$  помечается 1, поскольку он является левым листом. Узел  $b$  — 0, поскольку это правый лист. Метка узла  $t_1$  — 1, так как метки его потомков не

<sup>17</sup> При обходе в обратном порядке (postorder) рекурсивно посещаются поддеревья, корнями которых являются дочерние по отношению к узлу  $n$  узлы  $n_1, n_2, \dots, n_k$ , а затем сам узел  $n$ . Это порядок, в котором создаются узлы дерева восходящего разбора.

равны и максимальная метка потомка равна 1. На рис. 9.24 показано полученное в результате дерево с метками. Оно указывает, что для вычисления  $t_4$  нужны два регистра, как и для вычисления  $t_3$ .

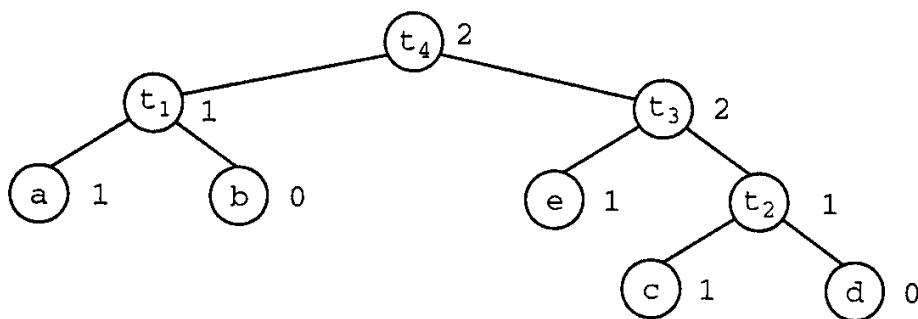


Рис. 9.24. Помеченное дерево

## Генерация кода на основе помеченного дерева

Теперь мы представим алгоритм, который принимает в качестве входа помеченное дерево  $T$  и дает на выходе последовательность машинного кода, вычисляющую  $T$  в регистр  $R0$  (его значение затем может быть сохранено в соответствующей ячейке памяти). Мы полагаем, что  $T$  содержит только бинарные операторы. Обобщение на операторы с произвольным числом operandов не представляет сложности и предлагается читателю в качестве упражнения.

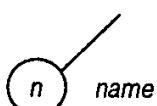
Алгоритм использует рекурсивную процедуру  $gencode(n)$  для получения машинного кода, вычисляющего поддерево (с корнем в узле  $n$ ) дерева  $T$  в регистр. Процедура  $gencode$  для распределения регистров использует стек  $rstack$ . Изначально  $rstack$  содержит все доступные регистры  $R0, R1, \dots, R(r-1)$  в указанном порядке. Вызов  $gencode$  может найти подмножество регистров (возможно, в другом порядке) в  $rstack$ . При возврате  $gencode$  процедура оставляет регистры в  $rstack$  в том же порядке, в котором они были найдены в стеке перед вызовом. Полученный в результате код вычисляет значение дерева  $T$  в регистр на вершине стека  $rstack$ .

Функция  $swap(rstack)$  переставляет два верхних регистра стека. Использование  $swap$  гарантирует, что левый потомок и его родитель вычисляются в один и тот же регистр.

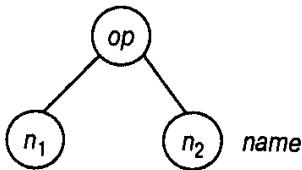
Для выделения временных ячеек памяти процедура  $gencode$  использует стек  $tstack$ . Мы полагаем, что изначально  $tstack$  содержит  $T0, T1, T2, \dots$ . На практике  $tstack$  не обязательно должен быть реализован в виде списка, если мы отслеживаем только  $i$ , такое, что  $T_i$  находится на вершине стека. Содержимое  $tstack$  всегда представляет собой суффикс  $T0, T1, T2, \dots$ .

Инструкция  $X := pop(stack)$  означает “снять значение со стека  $stack$  и присвоить его переменной  $X$ ”. И наоборот,  $push(stack, X)$  означает “поместить  $X$  в стек  $stack$ ”. Наконец,  $top(stack)$  обозначает значение на вершине стека  $stack$ .

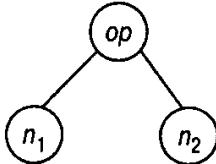
Алгоритм генерации кода состоит в вызове  $gencode$  для корня дерева  $T$ , где  $gencode$  — приведенная на рис. 9.25 процедура. Ее можно пояснить, рассмотрев отдельно каждый из пяти описанных ниже случаев. В случае 0 мы имеем следующее поддерево.



Здесь  $n$  является листом и крайним слева потомком своего родителя. Следовательно, мы генерируем только инструкцию загрузки. В случае 1 у нас имеется поддерево следующего вида.



Для него мы генерируем код для вычисления  $n_1$  в регистр  $R = \text{top}(rstack)$ , за которым следует инструкция  $op\ name$ ,  $R^{18}$ . В случае 2 мы имеем дело с поддеревом вида



где  $n_1$  может быть вычислено без сохранения в памяти, но вычисление  $n_2$  труднее (т.е. требует большего количества регистров), чем  $n_1$ . В этом случае мы производим обмен двух верхних регистров в  $rstack$  с помощью процедуры  $swap$ , затем вычисляем  $n_2$  в  $R = \text{top}(rstack)$ . После этого удаляем  $R$  из  $rstack$  и вычисляем  $n_1$  в  $S = \text{top}(rstack)$ . Заметим, что  $S$  представляет собой регистр, который в начале обработки случая 2 находился на вершине стека. Затем мы генерируем инструкцию  $op\ R, S$ , которая дает значение  $n$  (узла, помеченного  $op$ ) в регистре  $S$ . Еще один вызов  $swap$  приводит стек в то же состояние, в котором он был перед началом вызова процедуры  $gencode$ .

Случай 3 аналогичен 2, однако здесь труднее для вычисления левое поддерево, которое в данном случае вычисляется первым. Данный случай не требует применения процедуры  $swap$ .

Случай 4 осуществляется, когда оба поддерева требуют  $r$  или большего количества регистров для вычисления без сохранений в памяти. Поскольку мы вынуждены для временного хранения использовать память, вначале вычисляем правое поддерево во временную переменную  $t$  в памяти, затем вычисляем правое поддерево и, наконец, корень.

```

procedure gencode(n);
begin
/* Случай 0 */
if n — лист, представляющий операнд name and
 n — крайний левый потомок своего родителя then
 print 'MOV' || name || ',' || top(rstack)
else if n — внутренний узел с оператором op, левым
 потомком n1 и правым потомком n2 then
/* Случай 1 */
 if label(n2) = 0 then begin
 Пусть name — operand, представленный узлом n2;
 gencode(n1);
 print op || name || ',' || top(rstack)
 end
/* Случай 2 */

```

<sup>18</sup> Напомним, что правый operand операции  $op$  из поддерева в инструкции указывается первым (см. раздел 9.1). — Прим. ред.

```

else if $1 \leq \text{label}(n_1) < \text{label}(n_2)$ and
 $\text{label}(n_1) < r$ then begin
 swap(rstack);
 gencode(n2);
 $R := \text{pop}(\text{rstack});$ /* n_2 было вычислено в регистр R */
 gencode(n1);
 print op || R || ',' || top(rstack);
 push(rstack, R);
 swap(rstack)
 end
/* Случай 3 */
else if $1 \leq \text{label}(n_2) \leq \text{label}(n_1)$ and
 $\text{label}(n_2) < r$ then begin
 gencode(n1);
 $R := \text{pop}(\text{rstack});$ /* n_1 было вычислено в регистр R */
 gencode(n2);
 print op || top(rstack) || ',' || R;
 push(rstack, R)
 end
/* Случай 4; обе метки $\geq r$ (общего количества регистров) */
else begin
 gencode(n2);
 $t := \text{pop}(\text{tstack});$
 print 'MOV' || top(rstack) || ',' || t;
 gencode(n1);
 push(tstack, t);
 print op || t || ',' || top(rstack)
end
end

```

Рис. 9.25. Функция gencode

### Пример 9.13

Сгенерируем код для помеченного дерева (рис. 9.24) со стеком, изначально содержащим два регистра:  $rstack = R0, R1$ . Последовательность вызовов *gencode* и шагов вывода кода показана на рис. 9.26. В квадратных скобках приведено содержимое *rstack* при каждом вызове (вершина стека расположена справа). Полученный код представляет собой перестановку кода, приведенного на рис. 9.20.

|                               |         |                |
|-------------------------------|---------|----------------|
| <i>gencode(t<sub>4</sub>)</i> | [R1 R0] | /* Случай 2 */ |
| <i>gencode(t<sub>3</sub>)</i> | [R0 R1] | /* Случай 3 */ |
| <i>gencode(e)</i>             | [R0 R1] | /* Случай 0 */ |
| <b>print</b> MOV e, R1        |         |                |
| <i>gencode(t<sub>2</sub>)</i> | [R0]    | /* Случай 1 */ |
| <i>gencode(c)</i>             | [R0]    | /* Случай 0 */ |
| <b>print</b> MOV c, R0        |         |                |
| <b>print</b> ADD d, R0        |         |                |
| <b>print</b> SUB R0, R1       |         |                |
| <i>gencode(t<sub>1</sub>)</i> | [R0]    | /* Случай 1 */ |
|                               | [R0]    | /* Случай 0 */ |

```

gencode(a)
 print MOV a, R0
 print ADD b, R0
print SUB R1,R0

```

*Рис. 9.26. Отслеживание вызовов gencode*

Можно доказать, что *gencode* производит оптимальный код для вычисления выражений на нашей модели вычислительной машины, если предположить, что никакие алгебраические свойства операторов не учитываются и отсутствуют общие подвыражения. Доказательство, оставленное читателю в качестве упражнения, основано на том, что любая последовательность кода должна выполнить

1. операции для каждого внутреннего узла;
2. загрузку каждого листа, являющегося крайним слева потомком своего родителя;
3. сохранение для каждого узла, оба потомка которого имеют метки, не меньшие  $r$ .

Поскольку функция *gencode* выполняет именно эти действия, она оптимальна.

## Многорегистровые операции

Мы можем изменить наш алгоритм назначения меток таким образом, чтобы он обрабатывал операции типа умножения, деления или вызова функций, которые обычно требуют более одного регистра. Мы просто должны модифицировать шаг (6) на рис. 9.23 так, чтобы  $label(n)$  всегда было не меньше количества регистров, необходимых для выполнения операции. Например, если вызов функции требует всех  $r$  регистров, заменим строку (6) строкой  $label(n) = r$ . Если для умножения требуется два регистра, в бинарном случае используем

$$label(n) = \begin{cases} \max(2, l_1, l_2) & \text{если } l_1 \neq l_2 \\ l_1 + 1 & \text{если } l_1 = l_2 \end{cases}$$

где  $l_1$  и  $l_2$  — метки дочерних по отношению к  $n$  узлов.

К сожалению, такая модификация не гарантирует, что для умножения или деления (или операций с повышенной точностью) будут доступны пары регистров. На некоторых машинах можно “схватить”, считая, что операции умножения и деления требуют три регистра. Если в *gencode* никогда не используется *swap*, то *rstack* всегда будет содержать последовательные регистры  $i, i+1, \dots, r-1$  для некоторого  $i$ . Следовательно, первые три гарантированно будут содержать пару регистров. Поскольку многие операции коммутативны, мы часто можем избежать использования случая 2 в функции *gencode*, в котором используется процедура *swap*. Кроме того, даже если на вершине *rstack* нет трех последовательных регистров, шансы найти где-либо в стеке пару регистров достаточно высоки.

## Алгебраические свойства

Если мы примем во внимание алгебраические законы, которым подчиняются различные операторы, то получим возможность заменить данное дерево  $T$  деревом с меньшими метками (чтобы избежать сохранения в случае 4 в функции *gencode*) и/или меньшим количеством левых листьев (для уменьшения числа загрузок в случае 0). Например, поскольку обычно  $+$  рассматривается как коммутативная операция, мы можем заменить

дерево на рис. 9.27 $a$  деревом, приведенным на рис. 9.27 $b$ , сокращая количество левых листвьев на единицу и, возможно, уменьшая некоторые метки.

Поскольку  $+$  обычно является и коммутативной, и ассоциативной операцией, кластер узлов, помеченный на рис. 9.27 $c$  как  $+$ , можно заменить левой цепочкой поддеревьев, показанной на рис. 9.27 $d$ . Для минимизации метки корня нам требуется расположить их так, чтобы  $T_{i_1}$  был узлом с наибольшей меткой среди  $T_1, T_2, T_3$  и  $T_4$  и чтобы  $T_{i_1}$  не был листом (кроме случая, когда листьями являются все узлы  $T_1, T_2, T_3$  и  $T_4$ ).

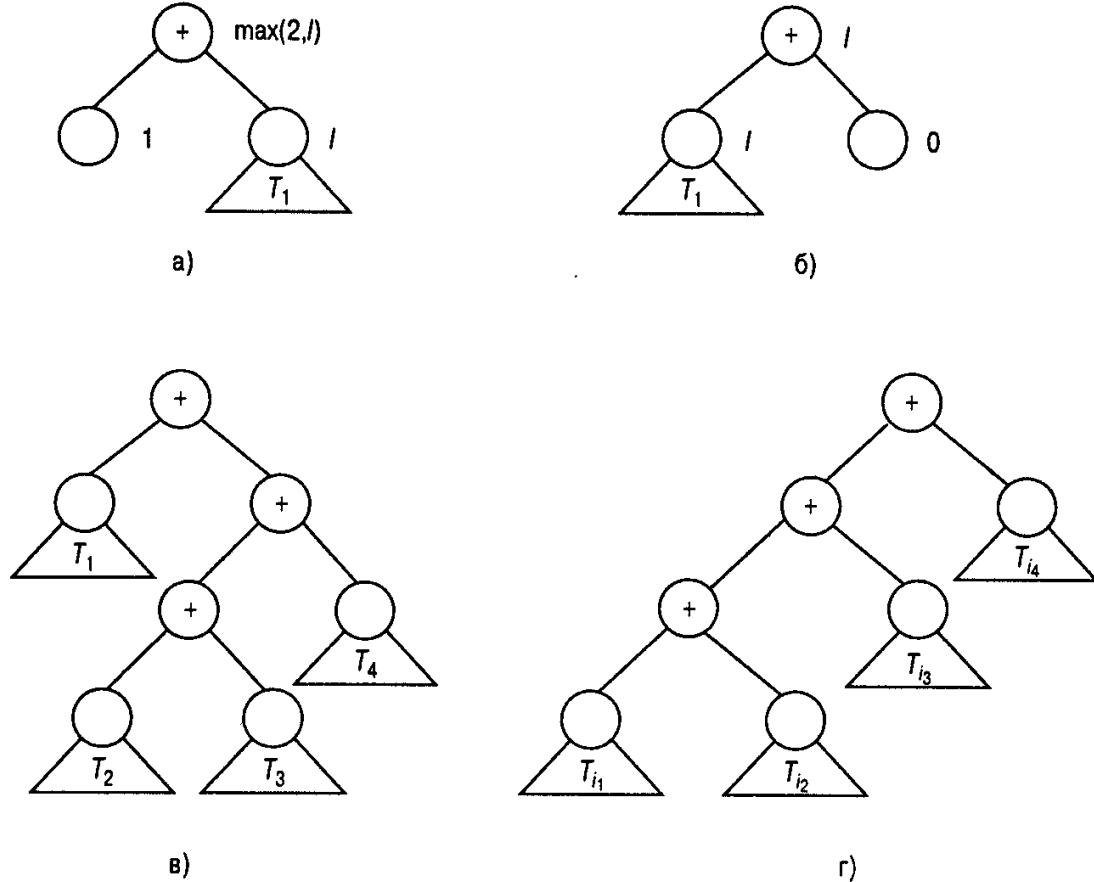


Рис. 9.27. Коммутативные и ассоциативные преобразования

## Общие подвыражения

При наличии в базовом блоке общих подвыражений даг больше не является деревом. Общие подвыражения соответствуют узлам, которые имеют более чем одного предка и называются *разделяемыми узлами* (*shared nodes*). В этом случае не могут быть непосредственно применены алгоритм назначения меток или функция *gencode*. В действительности наличие общих подвыражений делает генерацию кода с математической точки зрения заметно более сложной задачей. Бруно и Сети [68] показали, что генерация оптимального кода для дагов на однорегистровой машине является NP-полной задачей, а Ахо, Джонсон и Ульман [12] распространили полученный результат на машины с любым количеством регистров. Сложность задачи возрастает при попытках определить оптимальный порядок, в котором вычисление дага оказывается наиболее дешевым.

На практике мы можем получить вполне приемлемое решение, разбивая даг на множество деревьев путем поиска для каждого корня и/или разделяемого узла *и* максимального поддерева с *и* в качестве корня, не включающего никаких других разделяемых уз-

лов, за исключением листьев. Например, даг, приведенный на рис. 9.22, может быть разделен на деревья так, как показано на рис. 9.28. Каждый разделяемый узел с  $p$  родителями появляется в качестве листа не более чем в  $p$  деревьях. Узлы с более чем одним родительским узлом в данном дереве могут быть преобразованы в необходимое количество листьев, так что никакой лист не имеет более одного родительского узла.

После того как мы разбили даг на деревья описанным методом, мы можем упорядочить вычисление деревьев и использовать рассмотренные ранее алгоритмы для генерации кода каждого из них. Порядок вычисления деревьев должен быть установлен так, чтобы разделяемые значения, представленные листьями, были доступны к моменту вычисления дерева. Эти значения могут вычисляться и записываться в память либо храниться в регистрах при достаточном их количестве. Хотя генерируемый таким образом код и не является оптимальным, его качество в основном вполне удовлетворительное.

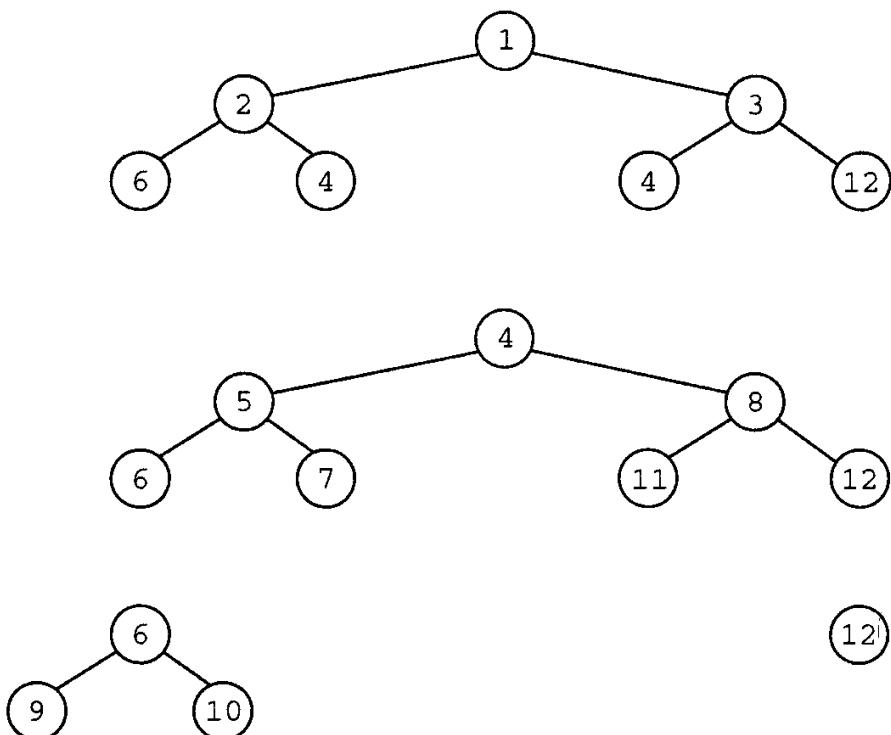


Рис. 9.28. Разбиение дага на деревья

## 9.11. Алгоритм динамического программирования для генерации кода

В предыдущем разделе процедура *gencode* приводила к оптимальному коду на основе дерева выражения за время, линейно пропорциональное размеру дерева. Эта процедура пригодна для машин, в которых все вычисления выполняются в регистрах и инструкции которых представляют собой операторы, примененные к двум регистрам или регистру и ячейке памяти.

Для расширения класса машин, для которых оптимальный код на основе дерева выражения может быть сгенерирован за линейное время, можно использовать алгоритм, основанный на принципах динамического программирования. Алгоритм динамического программирования применим к широкому классу регистровых машин с расширенным и сложным набором инструкций.

## Класс регистровых машин

Алгоритм динамического программирования может использоваться для генерации кода для любой машины с  $r$  взаимозаменяемыми регистрами  $R_0, R_1, \dots, R_{r-1}$  и инструкциями типа  $R_i := E$ , где  $E$  — произвольное выражение, содержащее операторы, регистры и ячейки памяти. Если  $E$  включает один или несколько регистров,  $R_i$  должен быть одним из них. Такая модель включает и машину, рассмотренную в разделе 9.2.

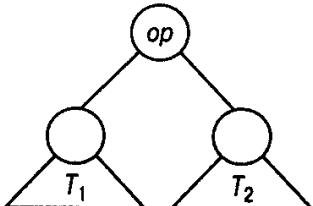
Например, инструкция  $ADD R_0, R_1$  может быть представлена в виде  $R_1 := R_1 + R_0$ , а инструкция  $ADD *R_0, R_1$  — в виде  $R_1 := R_1 + \text{ind } R_0$ , где  $\text{ind}$  означает оператор косвенного обращения.

Мы считаем, что машина имеет инструкции загрузки  $R_i := M$ , сохранения  $M := R_i$  и копирования регистров  $R_i := R_j$ . Для простоты мы также полагаем, что стоимость каждой инструкции — одна единица, хотя алгоритм динамического программирования можно легко модифицировать даже для случая, когда каждая инструкция имеет собственную стоимость.

## Принцип динамического программирования

Алгоритм динамического программирования разделяет задачу генерации оптимального кода для выражения на подзадачи генерации оптимального кода для подвыражений данного выражения. В качестве простого примера рассмотрим выражение  $E$  вида  $E_1 + E_2$ . Оптимальная программа для  $E$  образуется путем комбинации оптимальных программ для  $E_1$  и  $E_2$  в том или ином порядке, за которой следует код вычисления оператора  $+$ . Подзадачи генерации оптимального кода для  $E_1$  и  $E_2$  решаются аналогично.

Оптимальная программа, порожденная алгоритмом динамического программирования, имеет важное свойство, заключающееся в том, что она вычисляет выражение  $E = E_1 \text{ op } E_2$  “последовательно”. Чтобы понять, что это означает, можно рассмотреть синтаксическое дерево  $T$  для  $E$ .



Здесь  $T_1$  и  $T_2$  — деревья для  $E_1$  и  $E_2$  соответственно.

## Последовательное вычисление

Мы говорим, что программа  $P$  вычисляет дерево  $T$  *последовательно* (contiguously), если она вначале вычисляет те поддеревья  $T$ , которые должны быть вычислены в память. Затем программа вычисляет остаток  $T$  либо в порядке  $T_1, T_2$  и затем корень, либо —  $T_2, T_1$  и затем корень. В любом случае при необходимости используются предварительно вычисленные значения, хранящиеся в памяти. В качестве примера непоследовательного вычисления  $P$  сначала можно вычислить часть  $T_1$ , оставив полученное значение в регистре, а не в памяти, затем вычислить  $T_2$  и вновь возвратиться к оставшейся части  $T_1$ .

Можно доказать, что для определенной выше регистровой машины для любой программы  $P$ , вычисляющей дерево выражения  $T$ , можно найти эквивалентную программу  $P'$ , такую, что

1. стоимость  $P'$  не больше стоимости  $P$ ,
2.  $P'$  использует не больше регистров, чем  $P$ ,
3.  $P'$  вычисляет дерево последовательно.

Из этого результата следует, что каждое дерево выражения можно вычислить оптимальным образом с помощью последовательной программы.

Кстати, для машин с парами регистров типа IBM System/370 не всегда имеется оптимальное последовательное вычисление. Для таких машин можно привести примеры деревьев выражений, по которым оптимальная программа на машинном языке должна вначале вычислить в регистр часть левого поддерева корня, затем часть правого поддерева, затем еще одну часть левого поддерева, затем еще одну часть правого и т.д. Такие осцилляции не требуются для оптимального вычисления какого бы то ни было дерева выражения на обобщенной регистровой машине.

Свойство последовательного вычисления, определенное выше, говорит о том, что для любого дерева выражения  $T$  всегда существует оптимальная программа вычисления, состоящая из оптимальных программ вычисления поддеревьев корня, за которыми следует инструкция вычисления корня. Это свойство позволяет нам использовать алгоритм динамического программирования для создания оптимальной программы вычисления  $T$ .

## Алгоритм динамического программирования

Данный алгоритм состоит из трех фаз. Предположим, что целевая машина содержит  $r$  регистров. На первой фазе в восходящем порядке для каждого узла  $n$  дерева  $T$  вычисляется массив стоимости  $C$ ,  $i$ -й элемент которого  $C[i]$  представляет собой оптимальную стоимость вычисления в регистр поддерева  $S$  с корнем  $n$ , при условии, что для вычисления доступны  $i$  регистров,  $1 \leq i \leq r$ . Стоимость включает все необходимые для вычисления  $S$  загрузки и сохранения для данного количества регистров. В нее также входит стоимость вычисления оператора в корне  $S$ . Нулевой компонент вектора стоимости представляет собой оптимальную стоимость вычисления  $S$  в память. Свойство последовательного вычисления гарантирует, что оптимальная программа для  $S$  может быть сгенерирована путем рассмотрения комбинаций оптимальных программ только для поддеревьев корня  $S$ . Такое ограничение уменьшает число случаев, которые должны быть рассмотрены.

Для вычисления  $C[i]$  в узле  $n$  рассмотрим каждую машинную инструкцию  $R := E$ , выражение  $E$  которой соответствует подвыражению с корнем в узле  $n$ . Изучение векторов стоимости в соответствующих потомках  $n$  позволяет определить стоимость вычисления операндов  $E$ . Для операндов  $E$ , представляющих собой регистры, рассмотрим все возможные порядки вычислений поддеревьев  $T$  в регистры. Для каждого из рассматриваемых порядков вычисления первое поддерево, соответствующее регистровому операнду, можно вычислить с использованием  $i$  доступных регистров, второе — с помощью  $i-1$  и т.д. При подсчете стоимости для узла  $n$  учтем инструкцию  $R := E$ , которая использовалась для соответствующего узла  $n$ . Значение  $C[i]$  в таком случае представляет собой минимальную стоимость для всех возможных порядков вычисления.

Векторы стоимости для всего дерева  $T$  могут быть вычислены в восходящем порядке (снизу вверх) за время, линейно пропорциональное количеству узлов в дереве  $T$ . Инструкции, используемые для получения наилучшей стоимости  $C[i]$  для каждого значения  $i$ , можно хранить в узлах дерева; при этом наименьшая стоимость в векторе корневого узла  $T$  дает минимальную стоимость вычисления  $T$ .

Во второй фазе алгоритма при обходе дерева  $T$  векторы стоимости используются для определения того, какие из поддеревьев  $T$  должны вычисляться в память. В третьей фазе обход каждого дерева с использованием векторов стоимости и связанных с ними инструкций генерирует конечный целевой код; при этом первым генерируется код для поддеревьев, вычисляемых в память. Обе эти фазы также можно выполнить за время, линейно пропорциональное размеру дерева выражения.

### Пример 9.14

Рассмотрим машину с двумя регистрами —  $R0$  и  $R1$  — и следующим набором инструкций (стоимость каждой из них равна 1).

```

 $R_i := M_j$
 $R_i := R_i \text{ op } R_j$
 $R_i := R_i \text{ op } M_j$
 $R_i := R_j$
 $M_j := R_i$

```

Здесь  $R_i$  представляет собой либо  $R0$ , либо  $R1$ , а  $M_j$  обозначает адрес в памяти.

Используем алгоритм динамического программирования, чтобы сгенерировать оптимальный код для синтаксического дерева, приведенного на рис. 9.29. В первой фазе алгоритма мы вычисляем векторы стоимости, показанные около каждого узла. Чтобы проиллюстрировать вычисление стоимости, рассмотрим вектор стоимости у листа  $a$ .  $C[0]$ , стоимость вычисления  $a$  в память, равна нулю, поскольку  $a$  и так находится в памяти. Стоимость  $C[1]$  вычисления  $a$  в регистр равна 1, поскольку мы можем загрузить это значение в регистр с помощью одной инструкции  $R0 := a$ .  $C[2]$ , стоимость загрузки  $a$  в регистр при двух доступных регистрах та же, что и при одном. Таким образом, вектор стоимости в листе  $a$  представляет собой  $(0,1,1)$ .

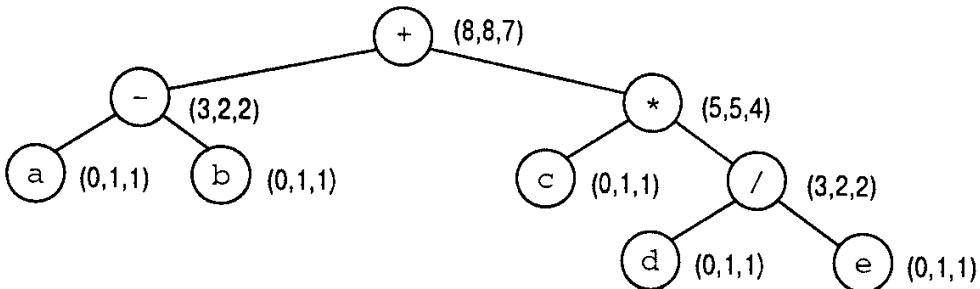


Рис. 9.29. Синтаксическое дерево для выражения  $(a-b) + c * (d/e)$  с вектором стоимости в каждом узле

Рассмотрим вектор стоимости в корне. Сначала мы определяем минимальную стоимость вычисления корня при одном или двух доступных регистрах. Поскольку корень помечен оператором  $+$ , корню соответствует машинная инструкция  $R0 := R0 + M$ . При использовании этой инструкции минимальная стоимость вычисления корня с одним доступным регистром представляет собой сумму минимальной стоимости вычисления правого поддерева в память, минимальной стоимости вычисления левого поддерева в регистр и еще одной единицы для инструкции сложения. Других способов вычисления нет. Векторы стоимости правого и левого потомков корня показывают, что минимальная стоимость вычисления корня при одном доступном регистре равна  $5+2+1=8$ .

Теперь рассмотрим минимальную стоимость вычисления корня с двумя доступными регистрами. Здесь возможны три варианта, зависящие от того, какая из инст-

рукций используется для вычисления корня и в каком порядке вычисляются левые и правые поддеревья.

1. Вычисляем левое поддерево в регистр R0 при двух доступных регистрах; после этого вычисляем правое поддерево в регистр R1 при одном доступном регистре и для вычисления корня используем инструкцию  $R0 := R0 + R1$ . Такая последовательность действий имеет стоимость  $2+5+1=8$ .
2. Вычисляем правое поддерево в регистр R1 при двух доступных регистрах; после этого вычисляем левое поддерево в регистр R0 при одном доступном регистре и для вычисления корня используем инструкцию  $R0 := R0 + R1$ . Такая последовательность действий имеет стоимость  $4+2+1=7$ .
3. Вычисляем правое поддерево в память по адресу M, вычисляем левое поддерево в регистр R0 при двух доступных регистрах и для вычисления корня применяем инструкцию  $R0 := R0 + M$ . Такая последовательность действий имеет стоимость  $5+2+1=8$ .

Второй вариант дает минимальную стоимость, равную 7.

Минимальная стоимость вычисления корня в память определяется прибавлением 1 к минимальной стоимости вычисления корня при всех доступных регистрах, т.е. мы вычисляем корень в регистр, а затем сохраняем полученный результат. Таким образом, вектор стоимости в корне представляет собой (8,8,7).

По векторам стоимости мы можем легко построить код путем обхода дерева. Для дерева на рис. 9.29, если предположить, что доступны два регистра, оптимальный код имеет следующий вид.

```
R0 := c
R1 := d
R1 := R1 / e
R0 := R0 * R1
R1 := a
R1 := R1 - b
R1 := R1 + R0
```

□

Эта технология, изначально разработанная в [10], использована в ряде компиляторов, включая, например, вторую версию переносимого компилятора С. Джонсона РСС2. Данная технология упрощает перенастройку компилятора для другой целевой машины, поскольку алгоритм динамического программирования применим к широкому классу машин.

## 9.12. Генераторы генераторов кода

Генерация кода включает выбор порядка вычисления операций, назначение регистров для хранения вычисленных значений и выбор инструкций целевого машинного языка для реализации операторов промежуточного представления программы. Даже если предположить, что порядок вычислений задан, а регистры распределены с помощью отдельных механизмов, проблема выбора используемых инструкций все равно может оказаться сложной комбинаторной задачей, в особенности для машин с богатым набором режимов адресации. В этом разделе мы представим технологии

построения деревьев, которые могут использоваться для автоматизации создания фазы выбора инструкций генератора кода на основе высокогоуровневого описания целевой машины.

## Генерация кода путем преобразования дерева

В этой главе входом для процесса генерации кода везде будет служить последовательность деревьев на семантическом уровне целевой машины. Эти деревья мы можем получить после вставки адресов времени исполнения в промежуточное представление, как описано в разделе 9.3.

### Пример 9.15

На рис. 9.30 приведено дерево для инструкции присвоения  $a[i] := b + 1$ , в которой  $a$  и  $i$  — локальные переменные; их адреса времени исполнения задаются как смещения  $const_a$  и  $const_i$  по отношению к  $SP$ , регистру, содержащему указатель на начало текущей записи активации. Массив  $a$  хранится в стеке времени исполнения. Присвоение элементу  $a[i]$  является косвенным присвоением, в котором  $r$ -значение ячейки памяти  $a[i]$  устанавливается равным  $r$ -значению выражения  $b+1$ . Адрес массива  $a$  определяется путем сложения значения константы  $const_a$  и содержимого регистра  $SP$ ; значение  $i$  находится в памяти по адресу, определяемому путем прибавления значения  $const_i$  к содержимому регистра  $SP$ . Переменная  $b$  является глобальной и располагается в памяти по адресу  $mem_b$ . Для простоты мы считаем, что все переменные имеют символьный тип.

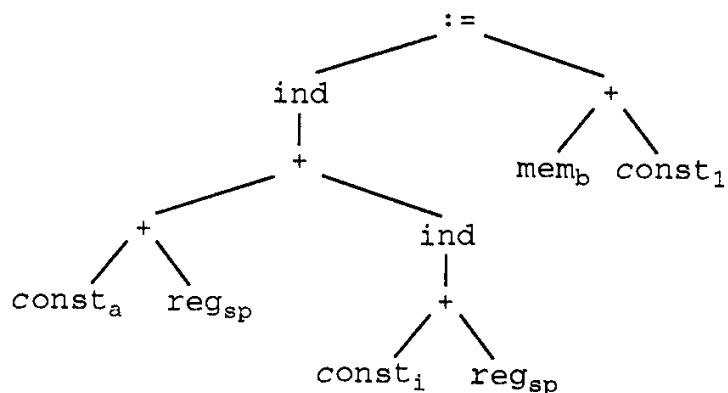


Рис. 9.30. Дерево промежуточного кода для  $a[i] := b + 1$

В приведенном дереве оператор  $ind$  рассматривает свой аргумент как адрес памяти. В качестве левого дочернего узла оператора присвоения  $ind$  дает адрес памяти, по которому будет сохранено  $r$ -значение правой части оператора присвоения. Если аргумент операторов  $+$  или  $ind$  представляет собой ячейку памяти или регистр, то в качестве значения принимается содержимое этой ячейки памяти или регистра. Листья дерева представляют собой атрибуты типа (константа, регистр, ячейка памяти) с индексами, указывающими значения этих атрибутов.  $\square$

Целевой код генерируется в процессе свертки входного дерева в единый узел путем последовательного применения правил преобразования дерева. Каждое правило преобразования представляет собой инструкцию вида

$replacement \leftarrow template \{ action \}$

где *replacement* — отдельный узел, *template* — дерево, а *action*, как и в случае схемы синтаксически управляемой трансляции, является фрагментом кода.

Множество правил преобразования дерева именуется *схемой трансляции дерева* (*tree-translation scheme*).

Каждый шаблон дерева представляет собой вычисления, выполняемые последовательностью машинных инструкций, выводимых связанным с ним действием. Обычно шаблон соответствует одной машинной инструкции. Листья шаблона являются атрибутами с низкими индексами, как и во входном дереве. Зачастую к значениям индексов в шаблонах применяется ряд ограничений, определяемых в виде семантических предикатов, которым должен удовлетворять шаблон перед тем, как можно будет говорить о его соответствии. Например, предикат может требовать, чтобы значение константы находилось в определенном диапазоне.

Схема трансляции дерева удобна для представления фазы выбора инструкций генератора кода. В качестве примера рассмотрим правило для инструкции сложения одного регистра с другим.

$$\text{reg}_i \leftarrow \begin{array}{c} + \\ / \quad \backslash \\ \text{reg}_i \quad \text{reg}_j \end{array} \quad \{ \text{ADD } R_j, \text{ } R_i \}$$

Это правило применяется следующим образом. Если входное дерево содержит поддерево, соответствующее приведенному шаблону, т.е. поддерево, корень которого помечен оператором  $+$ , а его левый и правый потомки представляют собой величины, хранящиеся в регистрах  $i$  и  $j$ , то мы можем заменить это поддерево одним узлом с меткой  $\text{reg}_i$  и вывести инструкцию  $\text{ADD } R_j, R_i$ . Одно поддерево может соответствовать нескольким шаблонам; далее мы вкратце опишем механизм выбора правила в случае возникновения конфликтов. Мы считаем также, что распределение регистров выполнено до выбора кода.

### Пример 9.16

На рис. 9.31 представлены правила преобразования дерева для нескольких инструкций нашей целевой машины. Эти правила будут использоваться в последующих примерах данного раздела. Первые два правила соответствуют инструкциям загрузки, следующие два — инструкциям сохранения, а остальные — индексированным загрузкам и сложениям. Обратите внимание, что правило (8) требует, чтобы значение константы было равно 1. Это условие может определяться семантическим предикатом.  $\square$

Схема трансляции дерева используется следующим образом. В данном входном дереве производится поиск поддеревьев, соответствующих имеющимся шаблонам. При соответствии шаблону поддерево заменяется узлом, определяемым правилом преобразования, и выполняются связанные с правилом действия. Если действие содержит последовательность машинных инструкций, они выводятся в выходной поток. Этот процесс повторяется до тех пор, пока в дереве имеются поддеревья, соответствующие шаблонам, и исходное дерево не свернуто в единственный узел. Последовательность машинных инструкций, генерируемая в процессе свертки входного дерева в один узел, образует выход схемы трансляции для данного входного дерева.

Процесс определения генератора кода аналогичен использованию схемы синтаксически управляемой трансляции для определения транслятора. Мы создаем схему трансляции дерева для описания набора инструкций целевой машины. На практике мы должны найти такую схему, которая приведет к генерации последовательности инст-

рукций с минимальной стоимостью для любого входного дерева. Существуют инструментальные средства, облегчающие автоматическое построение генератора кода по схеме трансляции дерева.

|     |                                                                                                                                          |                   |
|-----|------------------------------------------------------------------------------------------------------------------------------------------|-------------------|
| (1) | $\text{reg}_i \leftarrow \text{const}_c$                                                                                                 | { MOV #c, Ri }    |
| (2) | $\text{reg}_i \leftarrow \text{mem}_a$                                                                                                   | { MOV a, Ri }     |
| (3) | $\text{mem} \leftarrow \begin{array}{c} := \\ \text{mem}_a \quad \text{reg}_i \end{array}$                                               | { MOV Ri, a }     |
| (4) | $\text{mem} \leftarrow \begin{array}{c} := \\ \text{ind} \quad \text{reg}_j \\   \\ \text{reg}_i \end{array}$                            | { MOV Rj, *Ri }   |
| (5) | $\text{reg}_i \leftarrow \begin{array}{c} \text{ind} \\ + \\ \text{const}_c \quad \text{reg}_j \end{array}$                              | { MOV c(Rj), Ri } |
| (6) | $\text{reg}_i \leftarrow \begin{array}{c} + \\ \text{reg}_i \quad \text{ind} \\   \\ + \\ \text{const}_c \quad \text{reg}_j \end{array}$ | { ADD c(Rj), Ri } |
| (7) | $\text{reg}_i \leftarrow \begin{array}{c} + \\ \text{reg}_i \quad \text{reg}_j \end{array}$                                              | { ADD Rj, Ri }    |
| (8) | $\text{reg}_i \leftarrow \begin{array}{c} + \\ \text{reg}_i \quad \text{const}_1 \end{array}$                                            | { INC Ri }        |

Рис. 9.31. Правила преобразования дерева для некоторых инструкций целевой машины

### Пример 9.17

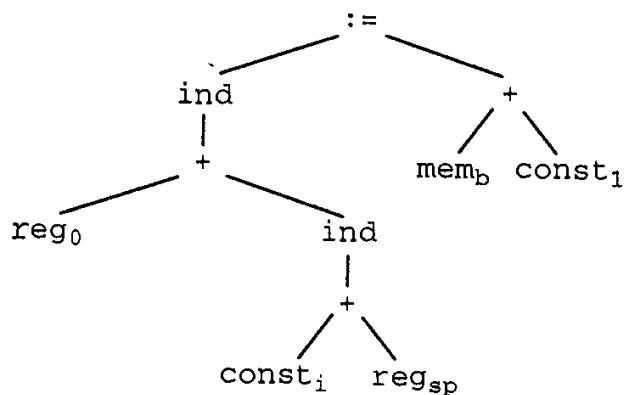
Воспользуемся схемой трансляции дерева (рис. 9.31) для генерации кода, соответствующего дереву, приведенному на рис. 9.30. Предположим, что первое правило

$$(1) \text{reg}_0 \leftarrow \text{const}_a \{ \text{MOV } \#a, \text{ R0} \}$$

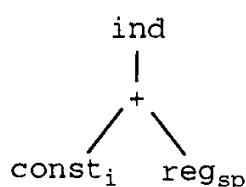
применяется для загрузки константы a в регистр R0. При этом метка крайнего слева листа заменяется  $c$   $\text{const}_a$  на  $\text{reg}_0$  и генерируется инструкция  $\text{MOV } \#a, \text{ R0}$ . Теперь седьмое правило

$$(7) \text{reg}_0 \leftarrow \begin{array}{c} + \\ \text{reg}_0 \quad \text{reg}_{sp} \end{array} \{ \text{ADD SP, R0} \}$$

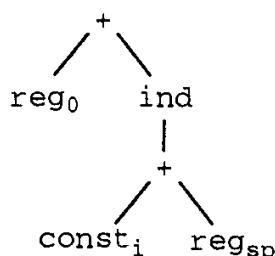
соответствует крайнему слева поддереву с меткой  $+$ . Используя это правило, мы заменим указанное поддерево одним узлом с меткой  $\text{reg}_0$  и сгенерируем инструкцию ADD SP, R0. В результате дерево выглядит следующим образом.



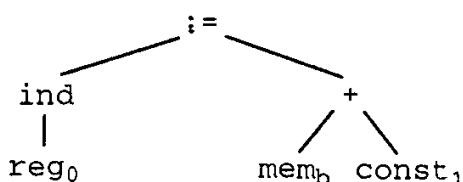
Теперь мы могли бы применить правило (5) для свертки поддерева



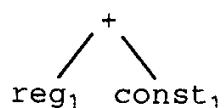
в узел с меткой  $\text{reg}_1$ . Однако мы можем использовать для свертки большего дерева



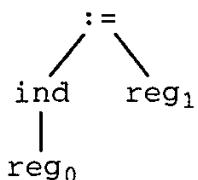
правило (6), которое приведет к единственному узлу с меткой  $\text{reg}_0$  и генерации инструкции ADD i(SP), R0. Поскольку более эффективным является использование одной инструкции для вычисления большого дерева, мы выбираем последний вариант и получаем после свертки следующее дерево.



В правом поддереве к листу  $\text{mem}_b$  мы применим правило (2), которое сгенерирует инструкцию для загрузки  $b$  в регистр 1 (для определенности). В результате правое поддерево



соответствует шаблону правила (8) и при свертке дает инструкцию INC R1. Теперь входное дерево свернуто в дерево



оно соответствует поддереву из правила (4), применение которого дает в результате единственный узел и генерирует инструкцию `MOV R1, *R0`.

Итак, в процессе свертки исходного дерева в единственный узел мы получили следующий код.

```

MOV #a, R0
ADD SP, R0
ADD i(SP), R0
MOV b, R1
INC R1
MOV R1, *R0

```

□

Ряд аспектов такого процесса свертки дерева требует дополнительных пояснений. Мы не указали, каким образом выполняется проверка соответствия дерева шаблону, в каком порядке для этого выбираются шаблоны или как поступить, когда найдено соответствие нескольким шаблонам. Заметим также, что процесс генерации кода блокируется, когда не найдено соответствие ни одному из имеющихся шаблонов. Кроме того, возможны преобразования одиночного узла, приводящие к генерации бесконечной последовательности инструкций перемещения информации между регистрами либо сохранения и загрузки.

Один из способов эффективной проверки соответствия шаблону состоит в преобразовании алгоритма Ахо-Корасика (упр. 3.32) в алгоритм нисходящего поиска соответствия шаблону дерева. Каждый шаблон можно представить в виде множества строк, а именно — множеством путей от корня к листьям. Для этого множества строк можно построить программу поиска соответствия шаблону по аналогии с упр. 3.32.

Задачи упорядочения шаблонов для проверки и множественного соответствия могут быть решены с помощью проверки соответствия дерева и шаблона совместно с алгоритмом динамического программирования из предыдущего раздела. Схема трансляции дерева может быть дополнена информацией о стоимости; с каждым из правил преобразования связывается стоимость генерируемой им последовательности инструкций.

На практике процесс преобразования дерева можно реализовать путем сопоставления поддеревьев с шаблонами в процессе обхода входного дерева вглубь и выполнения свертки при последнем посещении узла. Если при этом мы используем алгоритм динамического программирования, оптимальную последовательность соответствий шаблонам можно выбрать с учетом информации о стоимости, связанной с каждым из правил (возможно, при этом нам придется отложить принятие решения о выборе правила до получения информации о стоимости всех возможных альтернатив). Такой подход позволяет создать на основе схемы преобразования дерева небольшой и эффективный генератор кода. Кроме того, алгоритм динамического программирования освобождает создателя генератора кода от необходимости разрешения неоднозначности соответствий или выбора порядка вычислений.

## Проверка соответствия шаблону путем синтаксического анализа

Еще один подход состоит в использовании LR-синтаксического анализатора для выполнения проверки соответствия шаблонам. Входное дерево можно рассматривать как строку при использовании его префиксного представления. Так, префиксное представление дерева на рис. 9.30 выглядит следующим образом.

```
:= ind + + consta regSP ind + consti regSP + memb const1
```

Схема трансляции дерева может быть преобразована в схему синтаксически управляемой трансляции путем замены правил преобразования дерева продукциями контекстно-свободной грамматики, в которых правые части являются префиксными представлениями шаблонов инструкций.

### Пример 9.18

Схема синтаксически управляемой трансляции (рис. 9.32) основана на схеме трансляции дерева на рис. 9.31.  $\square$

|     |                                                 |                   |
|-----|-------------------------------------------------|-------------------|
| (1) | $reg_i \rightarrow const_c$                     | { MOV #c, Ri }    |
| (2) | $reg_i \rightarrow mem_a$                       | { MOV a, Ri }     |
| (3) | $mem \rightarrow := mem_a reg_i$                | { MOV Ri, a }     |
| (4) | $mem \rightarrow := ind reg_i reg_j$            | { MOV Rj, *Ri }   |
| (5) | $reg_i \rightarrow ind + const_c reg_j$         | { MOV c(Rj), Ri } |
| (6) | $reg_i \rightarrow + reg_i ind + const_c reg_j$ | { ADD c(Rj), Ri } |
| (7) | $reg_i \rightarrow + reg_i reg_j$               | { ADD Rj, Ri }    |
| (8) | $reg_i \rightarrow + reg_i const_1$             | { INC Ri }        |

Рис. 9.32. Схема синтаксически управляемой трансляции, построенная на основе рис. 9.31

На основе продукции схемы трансляции мы строим LR-синтаксический анализатор с применением одной из технологий, представленных в главе 4, “Синтаксический анализ”. Целевой код генерируется путем вывода машинных инструкций, соответствующих каждой свертке.

Обычно грамматика генерации кода весьма неоднозначна и поэтому должны быть предприняты дополнительные меры по разрешению конфликтов в процессе создания синтаксического анализатора. При отсутствии информации о стоимости общее правило состоит в предпочтении больших сверток меньшим. Это означает, что в случае конфликта свертка-свертка выбирается более длинная свертка, а при конфликте перенос-свертка — перенос. Такой подход “максимального разжевывания” приводит к выполнению большего числа операций с помощью одной машинной инструкции.

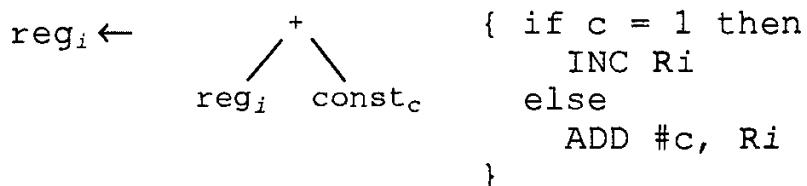
Существует ряд аспектов использования LR-синтаксического анализа для генерации кода. Во-первых, метод синтаксического анализа эффективен и хорошо изучен, а использование алгоритмов, описанных в главе 4, “Синтаксический анализ”, обеспечивает надежность и эффективность генератора кода. Во-вторых, при этом облегчается перенастройка генератора кода для другой целевой машины. Создание генератора кода для новой машины сводится к разработке грамматики, описывающей инструкции новой машины. В-третьих, качество генерируемого кода может быть сделано весьма высоким за счет добавления продукции для особых случаев, чтобы использовать преимущества машинных идиом.

Однако при этом подходе имеется и ряд трудностей. При синтаксическом анализе фиксирован порядок вычислений слева направо. Кроме того, для некоторых машин с большим количеством режимов адресации грамматика, описывающая целевую машину (и, соответственно, сам синтаксический анализатор), становится необычайно большой. Как следствие, необходимы специальные методы для работы с грамматикой описания целевой машины. При разработке грамматики следует быть крайне осторожным, чтобы полученный анализатор не мог быть заблокирован в процессе разбора дерева выражения из-за того, что грамматика не обрабатывает некоторые шаблоны операторов или синтаксический анализатор выполнил неверное разрешение возникшего конфликта. Следует также гарантировать невозможность входа синтаксического анализатора в бесконечный цикл сверток продукции с одним символом в правой части. Проблема циклов может быть разрешена путем использования технологии расщепления состояний при генерации таблиц синтаксического анализатора [163].

## Программы семантической проверки

Листья входного дерева представляют собой атрибуты типа с индексами, связывающими значения с атрибутами. В схеме трансляции генерации кода встречаются те же атрибуты, но зачастую с ограничениями на значения их индексов. Например, машинная инструкция может требовать, чтобы значения атрибута находились в определенном диапазоне или чтобы два значения атрибутов были связаны некоторым соотношением.

Такие ограничения на значения атрибутов могут быть указаны как предикаты, вызываемые перед выполнением свертки. Использование семантических действий и предикатов может обеспечить большую гибкость и простоту описания по сравнению с чисто грамматической спецификацией генератора кода. Для представления классов инструкций могут использоваться общие шаблоны, а семантические действия при этом могут использоваться для выбора инструкций в специальных ситуациях. Например, с помощью одного шаблона могут быть представлены два варианта инструкций сложения.



Конфликты действий синтаксического анализа могут быть разрешены с помощью предикатов устранения неоднозначностей, которые обеспечивают использование различных стратегий выбора в разных контекстах. Поскольку некоторые аспекты архитектуры целевой машины, например режимы адресации, могут быть выражены с помощью атрибутов, для целевой машины можно получить описание меньшего размера. Однако при этом трудно проверить, точно ли грамматика атрибутов описывает целевую машину (хотя в той или иной степени эта проблема касается всех генераторов кода).

## Упражнения

- 9.1. Постройте код следующих инструкций С для целевой машины, описанной в разделе 9.2, предполагая, что все переменные — статические. При построении кода исходите из того, что доступны три регистра.

a)  $x = 1$

- b)  $x = y$
- c)  $x = x + 1$
- d)  $x = a + b + c$
- e)  $x = a / (b + c) - d * (e + f)$

- 9.2.** Повторите упр. 9.1, предполагая, что все переменные находятся в стеке.
- 9.3.** Постройте код для следующих инструкций С для целевой машины из раздела 9.2, предполагая, что все переменные — статические. Как и в упр. 9.1, считаем, что доступны три регистра.
- a)  $x = a[i] + 1$
  - b)  $a[i] = b[c[i]]$
  - c)  $a[i][j] = b[i][k] * c[k][j]$
  - d)  $a[i] = a[i] + b[j]$
  - e)  $a[i] += b[j]$
- 9.4.** Выполните упр. 9.1 с использованием
- a) алгоритма из раздела 9.3;
  - b) процедуры *gencode*;
  - c) алгоритма динамического программирования из раздела 9.11.
- 9.5.** Постройте код для следующих инструкций С.
- a)  $x = f(a) + f(a) + f(a)$
  - b)  $x = f(a) / g(b, c)$
  - c)  $x = f(f(a))$
  - d)  $x = ++f(a)$
  - e)  $*p++ = *q++$
- 9.6.** Постройте код для следующей программы на языке С.
- ```
main () {
    int i = 0;
    int a[10];
    while (i < 10)
        a[i++] = 0;
}
```
- 9.7.** Предположим, что для цикла на рис. 9.13 мы используем три регистра для хранения a , b и c . Постройте код для блоков этого цикла. Сравните стоимость вашего кода со стоимостью кода на рис. 9.14.
- 9.8.** Постройте граф взаимодействия регистров для программы на рис. 9.13.
- 9.9.** Предположим, что для простоты мы автоматически сохраняем все регистры в стеке (или в памяти, если стек не используется) перед каждым вызовом процедуры и восстанавливаем их после возврата из нее. Как это влияет на формулу (9.4), используемую для оценки выгоды выделения регистра для данной переменной в цикле?

- 9.10. Модифицируйте функцию *getreg* из раздела 9.6 так, чтобы при необходимости она возвращала пары регистров.
- 9.11. Приведите пример дага, для которого эвристический метод упорядочения узлов, использованный для дага на рис. 9.21, не дает оптимального результата.
- *9.12. Постройте оптимальный код для следующих инструкций присвоения.

a) $x := a + b * c$
 b) $x := (a * -b) + (c - (d + e))$
 c) $x := (a / b - c) / d$
 d) $x := a + (b + c / d * e) / (f * g - h * i)$
 e) $a[i, j] := b[i, j] - c[a[k, 1]] * d[i+j]$

- 9.13. Постройте код для следующей программы на языке Pascal.

```
program forloop(input, output);
  var i, initial, final: integer;
begin
  read(initial, final);
  for i := initial to final do
    writeln(i)
end.
```

- 9.14. Постройте даг для следующего базового блока.

```
d := b * c
e := a + b
b := b * c
a := e - d
```

- 9.15. Укажите корректные порядки вычислений и имена значений в узлах дага из упр. 9.14 при условии, что в конце базового блока

- a) живы a, b и c ;
 b) жива только переменная a .

- 9.16. Какой порядок вычислений в упр. 9.15б будет наилучшим при генерации кода для машины с одним регистром? Поясните свой ответ.

- 9.17. Мы можем модифицировать алгоритм построения дага с тем, чтобы принять во внимание присвоения массивам и присвоения посредством указателей. При присвоении любому элементу массива мы считаем, что для этого массива создается новое значение, которое представлено узлом, потомками которого являются старое значение массива, значение индекса и присваиваемое значение. При присвоении посредством указателя мы создаем новое значение для каждой переменной, на которую может указывать этот указатель; дочерними для узлов каждого нового значения являются значение указателя и старое значение переменной, для которой может быть выполнено присвоение. Используя эти дополнения, постройте даг для следующего базового блока.

```
a[i] := b
*p := c
d := a[j]
e := *p
*p := a[i]
```

Предположите, что (а) p может указывать на любую переменную; (б) p может указывать только на переменные b или c . Не забудьте указать ограничения на порядок вычислений.

- 9.18. Если выполняется присвоение выражению с указателем или элементом массива, типа $a[i]$ или $*p$, которое затем используется и не может быть изменено в течение некоторого времени, мы можем распознать эту ситуацию и воспользоваться ею для упрощения дага. Например, поскольку в коде, представленном в упр. 9.17, между второй и четвертой инструкциями не происходит присвоения p , инструкцию $e := *p$ можно заменить инструкцией $e := c$ (мы знаем, что p указывает на нечто, имеющее значение c , хотя и не знаем, на что именно¹). Пересмотрите алгоритм построения дага с тем, чтобы учесть в нем описанную возможность, и примените ваш алгоритм к коду из упр. 9.17.
- **9.19. Разработайте алгоритм генерации оптимального кода для последовательности трехадресных инструкций вида $a := b + c$ на n -регистровой машине из примера 9.14. Инструкции должны выполняться в заданном порядке. Какова временная сложность вашего алгоритма?

Библиографические примечания

Читатель, который интересуется обзорами исследований, посвященных генерации кода, может обратиться к работам [16, 154, 166, 167, 185, 297, 441, 442]. Генерация кода для языка программирования Bliss рассматривается в [460], для Pascal — [31], а для PL.8 — [37].

При разработке компиляторов весьма полезна статистика использования программ. В [256] представлено эмпирическое изучение программ на языке Fortran, в [121] — некоторые статистические данные по использованию PL/I, а в [72, 402] — анализ Pascal-программ. Производительность различных компиляторов на различных множествах машинных инструкций рассмотрена в [115, 296, 403].

Многие эвристические алгоритмы, предлагаемые читателю в этой главе, использовались на практике в различных компиляторах. В [147] рассмотрено применение счетчиков для генерации кода базовых блоков. В [54] показано, что в контексте использования технологии подкачки страниц оптимальной является использованная в *getreg* стратегия освобождения регистров путем сброса тех из них, значения которых не будут использоваться дольше всего. Наша стратегия распределения фиксированного числа регистров для хранения переменных в процессе выполнения цикла упоминалась в работе [300] и была использована в реализации Fortran H [294].

В [196] приведен алгоритм оптимизации использования индексных регистров в Fortran. Раскраска графа в качестве метода распределения регистров была предложена в [125, 394]. Трактовка раскраски графа в разделе 9.7 следует работам [75, 76]. В [81] описан алгоритм приоритетной раскраски графа для распределения регистров. Другие подходы к распределению регистров рассмотрены в работах [53, 175, 222, 239, 283].

¹ Естественно, это рассуждение справедливо только в том случае, когда p не может указывать на d , так как в этом случае, несмотря на то что p остается неизменным, $*p$ принимает другое значение. — Прим. ред.

Алгоритм назначения меток из раздела 9.10 напоминает алгоритм именования рек: слияние основной реки и меньшего притока получает имя основной реки; слияние двух одинаковых рек — новое имя. Впервые алгоритм назначения меток появился в [123]. Алгоритм генерации кода с использованием этого метода был предложен в [33, 52, 327, 334, 364]. В [399] метод назначения меток был использован Сети и Ульманом в алгоритме, который, как было доказано, способен генерировать оптимальный код для деревьев выражений в очень многих ситуациях. Процедура *gencode* из раздела 9.10 является модификацией алгоритма Сети–Ульмана, описанной в [412]. В [67, 91] приведены оптимальные алгоритмы генерации кода для деревьев выражений, если целевая машина имеет регистры, которые должны использоваться в качестве стека.

В работе [10] разработан алгоритм динамического программирования, описанный в разделе 9.11. Этот алгоритм использовался в качестве базового для генератора кода в переносимом компиляторе РСС2 языка С, а также в компиляторе для машины IBM 370 [371]. Кнут [259] обобщил алгоритм динамического программирования для машин с асимметричными классами регистров, таких как IBM 7090 и CDC 6600. При разработке обобщения Кнут рассматривал генерацию кода как проблему синтаксического анализа контекстно-свободной грамматики.

В [134] приведен алгоритм для обработки общих подвыражений в арифметических выражениях. Разбиение дага на деревья и раздельное использование процедуры типа *gencode* с деревьями предложено в [441]. В [68] и [398] показано, что задача генерации оптимального кода для дагов является NP-полной. В [12] показано, что задача остается NP-полной даже для машин как с одним регистром, так и с неограниченным их множеством. В [7, 158] обсуждается важность NP-полноты задачи.

Преобразования базовых блоков изучались в [17, 116]. Локальная оптимизация обсуждается в [103–105, 142, 161, 275, 309], а в [421] предложено использовать ее для промежуточного кода.

Генерация кода рассматривается как процесс преобразования дерева в работах [74, 216, 444, 452]. Пример преобразования дерева в разделе 9.12 взят из [185]. В [6] предложена комбинация эффективной проверки соответствия деревьев шаблонам с алгоритмом динамического программирования (раздел 9.12). В [432] реализован язык генерации кода под названием *Twig*, основанный на схеме трансляции дерева, описанной в разделе 9.12. Алгоритмы общего вида для проверки соответствия деревьев шаблонам описаны в [190, 198, 271].

Подход к генерации кода с использованием LR-синтаксического анализатора для выбора инструкций описан в [22, 163, 164, 166, 167, 185]. В [152, 153] для определения и реализации генераторов кода использовались атрибутные грамматики.

Другие технологии для автоматизации построения генераторов кода предложены в работах [74, 141, 284]. Переносимость компиляторов обсуждалась также в [369, 370]. В [285, 418] описаны технологии сцепления инструкций переходов. В [461] приведен алгоритм решения упр. 9.19 с полиномиальным временем работы.

ГЛАВА 10

Оптимизация кода

В идеале компилятор должен давать столь же хороший целевой код, как и написанный вручную. Реальность же такова, что это достигается только в ограниченном количестве случаев, к тому же с огромным трудом. Однако зачастую код, который дает простой алгоритм компиляции, может быть сделан более быстрым или более компактным (или и то, и другое одновременно). Такое улучшение достигается преобразованием программы, традиционно именуемым *оптимизацией*, хотя употребление этого термина и является некорректным, поскольку крайне редко можно гарантировать, что полученный в результате преобразования код — наилучший возможный. Компилятор, применяющий улучшающие код преобразования, называется *оптимизирующими*.

Основной упор в этой главе делается на машинно-независимую оптимизацию, т.е. преобразование программы, которое улучшает целевой код без учета конкретных свойств целевой машины. Машино-зависимая оптимизация, такая как распределение регистров и применение специальных последовательностей машинных инструкций (машиных идиом), обсуждалась в главе 9, “Генерация кода”.

Максимальное вознаграждение за минимальные усилия можно получить, если определить часто выполняемые части программы и сделать их по возможности максимально эффективными. Есть поговорка, что в большинстве программ 90% времени выполнения тратится на 10% кода. Хотя реальное процентное соотношение может варьироваться, зачастую действительно небольшой фрагмент кода выполняется большую часть времени работы программы. Профилирование программы с представительными входными данными указывает наиболее “трудно проходимые” фрагменты программы. К сожалению, компилятор не имеет доступа к образцу входных данных и должен самостоятельно догадываться о “горячих точках” программы.

На практике хорошими кандидатами для улучшения являются внутренние циклы. В языке с управляемыми конструкциями типа `for` или `while` циклы ясно выделяются в структуре программы. В общем случае циклы идентифицируются процессом анализа потока управления по графу потока программы.

Эта глава изобилует полезными оптимизирующими преобразованиями и технологиями их реализации. Наилучший способ принять решение о применении того или иного способа в компиляторе — собрать статистику об исходной программе и оценить преимущества данного множества оптимизаций на представительном наборе реальных исходных программ. В главе 12, “Некоторые компиляторы”, описаны преобразования, которые доказали свою полезность в оптимизирующих компиляторах различных языков.

Одной из тем настоящей главы является анализ потока данных — процесс сбора информации о путях использования переменных в программе. Информация, собранная в различных местах программы, может быть связана с помощью простых уравнений с множествами. В этой главе мы представим ряд алгоритмов для сбора информации с использованием анализа потока данных и для эффективного использования этой информации в процессе оптимизации. Мы рассмотрим также влияние на оптимизацию таких языковых конструкций, как процедуры и указатели.

Последние четыре раздела этой главы посвящены более сложным вопросам, таким, например, как некоторые идеи теории графов, связанные с анализом потока управления, и их применение для анализа потока данных. Завершается глава обсуждением инструментария для выполнения анализа потока данных и технологий отладки оптимизированного кода. Основной упор в этой главе делается на технологиях оптимизации, применяемых к языкам в общем случае. Некоторые из компиляторов, использующие описанные здесь идеи, рассмотрены в главе 12, “Некоторые компиляторы”.

10.1. Введение

Для создания эффективных программ на целевом языке программисту нужен не только оптимизирующий компилятор. В этом разделе мы рассмотрим дополнительные возможности, доступные программисту и компилятору для создания эффективной целевой программы. Мы упомянем типы преобразований, улучшающих код, которые могут использоваться программистом и разработчиком компилятора для повышения производительности программы. Мы также рассмотрим представление программ, к которым могут быть применены эти преобразования.

Критерии для преобразований, улучшающих код

Попросту говоря, наилучшие преобразования программы — те, которые приводят к максимальному результату при минимальных усилиях. Преобразования, выполняемые оптимизирующими компилятором, должны обладать рядом свойств.

Во-первых, преобразование должно сохранять смысл программы, т.е. оптимизация не должна изменять выход программы для данного входа или приводить к ошибке (типа деления на нуль), если она не присутствовала в исходной версии программы. Это требование распространяется на всю данную главу; всякий раз, когда встает вопрос о “безопасности” преобразования, предпочтительнее не применять его вовсе, чем рисковать внесением изменений в работу программы.

Во-вторых, преобразование в среднем должно ускорять работу программы. Иногда мы заинтересованы в уменьшении размера скомпилированного кода, хотя сейчас размер кода менее важен, чем когда-то. Конечно, не всякое преобразование способно улучшить любую программу, и иногда “оптимизация” способна вызвать замедление работы программы — при том, что в среднем она повышает производительность.

В-третьих, преобразования должны стоить затрачиваемых усилий. Разработчику компилятора не имеет смысла затрачивать интеллектуальные усилия на реализацию улучшающего код преобразования и увеличивать время компиляции исходной программы, если эти усилия не окупятся при работе скомпилированной программы. Ряд локальных преобразований типа рассмотренных в главе 9, “Генерация кода”, достаточно прост для реализации и достаточно выгоден, чтобы быть включенным в любой компилятор.

Некоторые преобразования могут применяться только после кропотливого, зачастую весьма длительного анализа исходной программы, так что их применение в программах, которые будут выполняться только несколько раз, просто необоснованно. Например, в процессе отладки или компиляции “студенческих работ”, которые выполняются пару раз и навеки забываются, более полезен быстрый неоптимизирующий компилятор. Только когда рассматриваемая программа занимает значительную часть времени работы машины, оправдывается время, затраченное на обработку программы оптимизирующим компилятором.

Повышение производительности

Существенное повышение производительности программы — такое, как снижение времени работы с нескольких часов до нескольких секунд — обычно получается при совершенствовании программы на всех уровнях, от исходного текста до целевой программы, как показано на рис. 10.1. На каждом уровне возможные варианты действий простираются от поиска лучшего алгоритма до реализации данного алгоритма с помощью меньшего количества выполняемых операций.



Рис. 10.1. Возможности усовершенствования программы пользователем и компилятором

Алгоритмические преобразования иногда приводят к эффектному снижению времени исполнения программы. Например, в [55] показано, как время работы программы для сортировки N элементов снизилось с $2.02N^2$ микросекунд до $12\log_2 N$ микросекунд при замене сортировки вставкой быстрой сортировкой¹. Для $N = 100$ эта замена ускоряет программу в 2.5 раза, а при $N = 100\,000$ — более чем в 1000 раз.

К сожалению, компилятор не может найти наилучший алгоритм для данной программы, но иногда он в состоянии заменить последовательность операций алгебраически эквивалентной последовательностью и тем самым существенно снизить время работы программы. Такая экономия времени — обычное явление при применении алгебраических преобразований к программам на языках очень высокого уровня, типа языков запросов к базам данных [436].

В этом и следующем разделах для иллюстрации влияния различных видов преобразований будет использоваться программа быстрой сортировки. Программа на рис. 10.2 взята из [397], где рассматривается оптимизация этой программы вручную. Мы не будем касаться здесь алгоритмических аспектов этой программы — в действительности, чтобы эта программа работала, требуется, чтобы $a[0]$ содержал наименьший, а $a[\max]$ — наибольший сортируемые элементы.

```
void quicksort(int m, int n)
{
    int i, j;
    int v, x;
    if (n <= m) return;
    /* Здесь начинается интересующий нас фрагмент */
    i = m - 1; j = n; v = a[n];
    while (1) {
        do i = i + 1; while (a[i] < v);
        do j = j - 1; while (a[j] > v);
        if (i >= j) break;
        x = a[i]; a[i] = a[j]; a[j] = x;
    }
}
```

¹ См. в [8] обсуждение этих алгоритмов сортировки и их скоростей.

```

x = a[i]; a[i] = a[n]; a[n] = x;
/* Здесь заканчивается интересующий нас фрагмент */
quicksort(m, j); quicksort(i+1, n);
)

```

Рис. 10.2. Код быстрой сортировки на С

Выполнение преобразований на уровне исходного кода может оказаться невозможным. Например, в языках типа Pascal или Fortran программист может обращаться к элементу массива только обычным путем, как, например, к $b[i, j]$. Однако на уровне промежуточного кода могут появиться новые возможности для его улучшения. Например, трехадресный код предоставляет множество возможностей для улучшения вычисления адресов, в особенности в циклах. Рассмотрим трехадресный код для определения значения $a[i]$ в предположении, что каждый элемент массива имеет размер 4 байта:

```
t1 := 4 * i; t2 := a[t1]
```

Простейший промежуточный код будет вычислять $4 * i$ всякий раз, когда требуется обращение к элементу $a[i]$, и программист при этом не имеет возможности устранить излишние вычисления, поскольку они выполняются неявно реализацией языка программирования. В этой ситуации меры по устраниению излишнего кода должен принимать компилятор. Однако в языках типа С такое преобразование может быть выполнено программистом, поскольку обращение к элементу массива может быть переписано с использованием указателей, что делает его более эффективным². Такое переписывание аналогично преобразованиям, традиционно используемым в оптимизирующих компиляторах Fortran.

На уровне целевой машины компилятор ответственен за эффективное использование ее ресурсов. Например, хранение наиболее интенсивно используемых переменных в регистрах может значительно уменьшить время работы программы — зачастую наполовину. И здесь С позволяет программисту дать компилятору совет, какие именно переменные хранить в регистрах (в то время как большинство языков программирования не дают такой возможности). Кроме того, компилятор может значительно ускорить работу программы выбором инструкций, использующих преимущества режимов адресации машины, для выполнения задачи одной инструкцией там, где ожидается использование двух или трех — о чем уже говорилось в главе 9, “Генерация кода”.

Но даже если программист может улучшать код, все же более удобно, когда такие улучшения может взять на себя компилятор. Если компилятор генерирует эффективный код, программист может сконцентрироваться на написании ясного исходного текста, а не на внесении в него мелких улучшений.

Организация оптимизирующего компилятора

Как мы упоминали, зачастую имеется несколько уровней улучшения программы. Поскольку технологии, необходимые для анализа и преобразования программы, не претерпевают значительных изменений от уровня к уровню, в этой главе мы сконцентрируемся на преобразовании промежуточного кода с использованием приведенной на рис. 10.3 ор-

² Это примечание следует рассматривать как иллюстрацию, но не как руководство к действию, поскольку такие преобразования автоматически выполняются современными оптимизирующими компиляторами. — Прим. ред.

ганизации. Фаза улучшения кода состоит из анализа потока управления и потока данных, за которыми следует применение преобразований. Генератор кода, рассмотренный в главе 9, “Генерация кода”, дает целевую программу на основе преобразованного промежуточного кода.

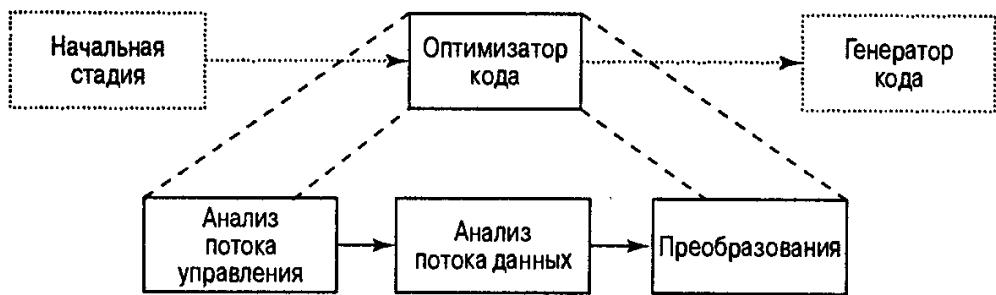


Рис. 10.3. Организация оптимизатора кода

Для удобства представления мы полагаем, что промежуточный код состоит из трехадресных инструкций. Промежуточный код приведенного фрагмента, полученный с помощью описанных в главе 8, “Генерация промежуточного кода”, технологий, показан на рис. 10.4. При использовании других промежуточных представлений временные переменные t_1, t_2, \dots, t_{15} могут не появиться в коде явным образом.

(1)	$i := m - 1$	(16)	$t_7 := 4 * i$
(2)	$j := n$	(17)	$t_8 := 4 * j$
(3)	$t_1 := 4 * n$	(18)	$t_9 := a[t_8]$
(4)	$v := a[t_1]$	(19)	$a[t_7] := t_9$
(5)	$i := i + 1$	(20)	$t_{10} := 4 * j$
(6)	$t_2 := 4 * i$	(21)	$a[t_{10}] := x$
(7)	$t_3 := a[t_2]$	(22)	goto (5)
(8)	if $t_3 < v$ goto (5)	(23)	$t_{11} := 4 * i$
(9)	$j := j - 1$	(24)	$x := a[t_{11}]$
(10)	$t_4 := 4 * j$	(25)	$t_{12} := 4 * i$
(11)	$t_5 := a[t_4]$	(26)	$t_{13} := 4 * n$
(12)	if $t_5 > v$ goto (9)	(27)	$t_{14} := a[t_{13}]$
(13)	if $i \geq j$ goto (23)	(28)	$a[t_{12}] := t_{14}$
(14)	$t_6 := 4 * i$	(29)	$t_{15} := 4 * n$
(15)	$x := a[t_6]$	(30)	$a[t_{15}] := x$

Рис. 10.4. Трехадресный код фрагмента, приведенного на рис. 10.2

Организация, показанная на рис. 10.3, обладает следующими достоинствами.

1. Операции, необходимые для реализации высокоуровневых конструкций, становятся явными в промежуточном коде, что позволяет оптимизировать их. Например, вычисление адреса $a[i]$ на рис. 10.4 выполняется явно, так что повторные вычисления $4 * i$ могут быть устранены, как описывается в следующем разделе.
2. Промежуточный код может быть (относительно) не зависящим от целевой машины, так что оптимизатор не должен сильно изменяться при переходе на другую целевую машину. Промежуточный код на рис. 10.4 основан на предположении о том, что размер элемента массива a — 4 байта. Некоторые промежуточные коды, например Р-код для Pascal, оставляют поле размера элемента массива незаполненным, предоставляя его заполнение генератору кода, и тем самым делают промежуточный код не

зависящим от размера машинного слова. Мы можем сделать то же, заменив 4 в нашем промежуточном коде символьной константой.

В оптимизаторе кода программы представлены графами потоков, дуги которых указывают поток управления, а узлы представляют базовые блоки (см. раздел 9.4). Пока не указано иное, под программой понимается отдельная процедура. Межпроцедурную оптимизацию мы рассмотрим в разделе 10.8.

Пример 10.1

На рис. 10.5 приведен график потока программы, показанной на рис. 10.4. Начальным узлом этого графа является B_1 . Все условные и безусловные переходы к инструкциям на рис. 10.4 заменены на рис. 10.5 переходами к блокам, лидерами которых являются соответствующие целевые инструкции.

На рис. 10.5 имеется три цикла. Блоки B_2 и B_3 сами являются циклами, и еще один цикл со входом B_2 сформирован блоками B_2 , B_3 , B_4 и B_5 . \square

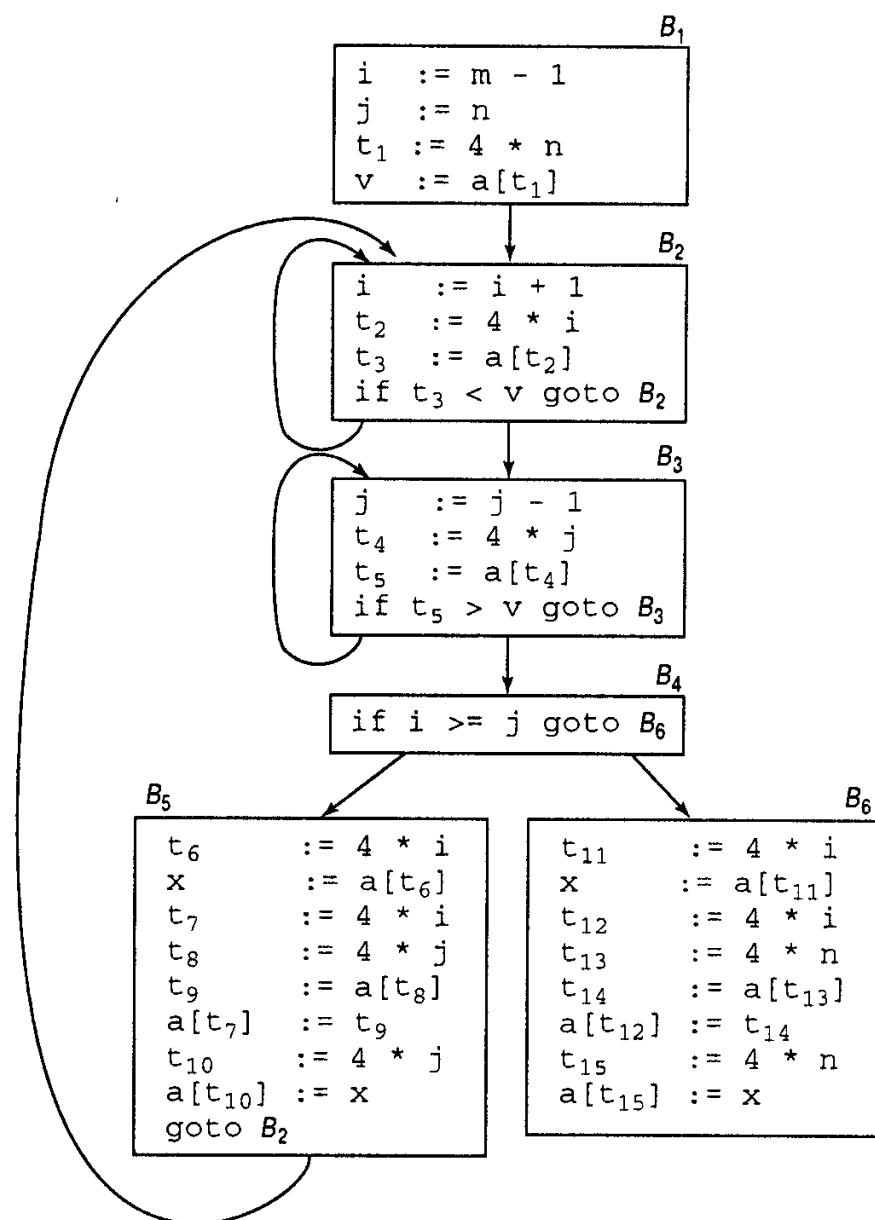


Рис. 10.5. Граф потока

10.2. Основные источники оптимизации

В этом разделе мы познакомимся с некоторыми наиболее употребительными преобразованиями, улучшающими код. Технологии для реализации этих преобразований представлены в последующих разделах. Преобразование программы называется *локальным*, если оно может быть выполнено только с инструкциями в пределах базового блока; в противном случае оно именуется *глобальным*. Многие преобразования могут выполняться как на локальном, так и на глобальном уровне. Первыми обычно выполняются локальные преобразования.

Преобразования, сохраняющие функции

Существует ряд способов, которыми компилятор может улучшить программу без изменения вычисляемой функции. Устранение общих подвыражений, размножение копий, удаление недоступного кода и дублирование констант — вот основные примеры таких преобразований, сохраняющих функции. В разделе 9.8 на примере представления базовых блоков в виде дага показано, каким образом при построении этого дага могут быть удалены локальные общие подвыражения. Другие преобразования выполняются, в основном, при глобальной оптимизации, и мы рассмотрим все их по очереди.

Зачастую программа включает несколько вычислений одного и того же значения, например смещения в массиве. Как упоминалось в разделе 10.1, некоторые из этих повторений не могут быть устранины программистом, поскольку они возникают на уровне детализации более низком, чем доступный в исходном языке. Например, блок B_5 , показанный на рис. 10.6a, дважды вычисляет значения $4 * i$ и $4 * j$.

B_5

```
t6 := 4 * i
x := a[t6]
t7 := 4 * i
t8 := 4 * j
t9 := a[t8]
a[t7] := t9
t10 := 4 * j
a[t10] := x
goto B2
```

а) До

B_5

```
t6 := 4 * i
x := a[t6]
t8 := 4 * j
t9 := a[t8]
a[t6] := t9
a[t8] := x
goto B2
```

б) После

Рис. 10.6. Локальное устранение общих подвыражений

Общие подвыражения

Выражение E называется *общим подвыражением*, если E было ранее вычислено и значения переменных в E с того времени не изменились. Повторного вычисления можно избежать, если мы сможем использовать ранее вычисленное значение. Например, присвоения t_7 и t_{10} на рис. 10.6a содержат в правой части общие подвыражения $4 * i$ и $4 * j$ соответственно. На рис. 10.6б они удалены благодаря использованию t_6 вместо t_7 и t_8 вместо t_{10} . Это изменение кода получается в результате перестройки промежуточного кода на основе дага базового блока.

Пример 10.2

На рис. 10.7 показан результат устранения как локальных, так и глобальных общих подвыражений из блоков B_5 и B_6 в графе потока на рис. 10.5. Мы сначала обсудим преобразование блока B_5 , а затем рассмотрим некоторые тонкости, вызываемые использованием массивов.

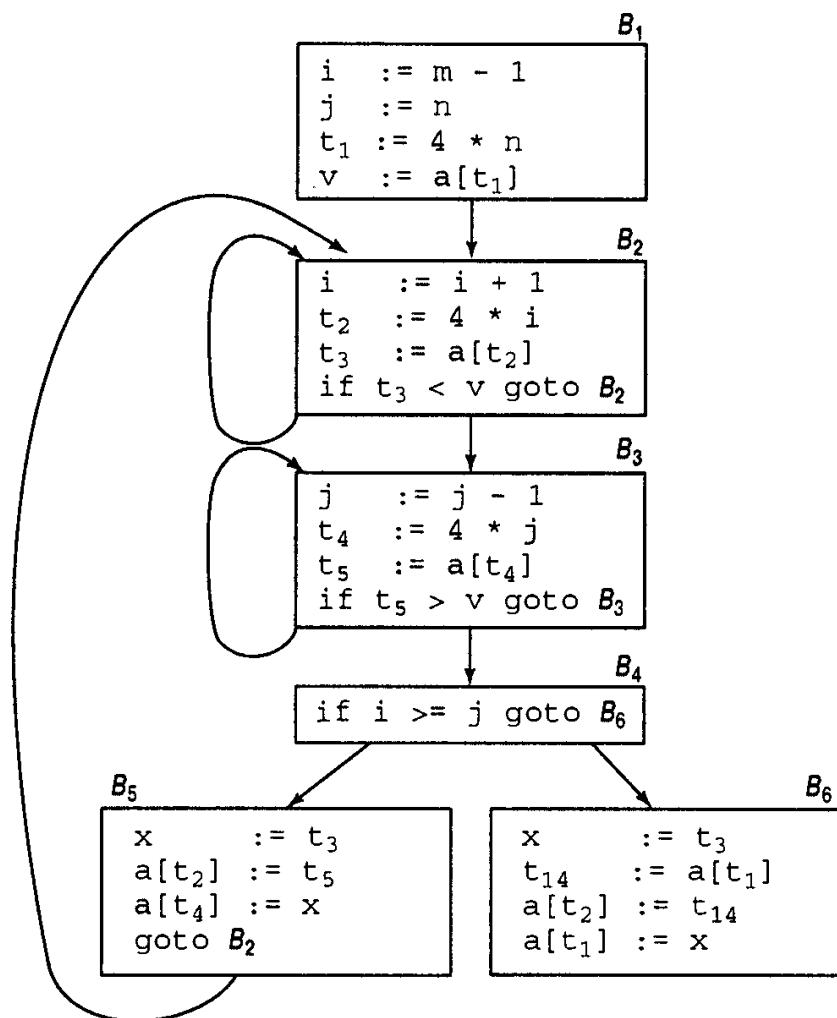


Рис. 10.7. Базовые блоки B_5 и B_6 после удаления общих подвыражений

После удаления локальных общих подвыражений блок B_5 продолжает вычислять выражения $4*i$ и $4*j$, как показано на рис. 10.6б. Оба они представляют собой общие подвыражения; в частности, три инструкции

$t_8 := 4 * j; \quad t_9 := a[t_8]; \quad a[t_8] := x$

в базовом блоке B_5 могут быть заменены инструкциями

$t_9 := a[t_4]; \quad a[t_4] := x$

с использованием t_4 , вычисленным в блоке B_3 . Как можно видеть из рис. 10.7, при передаче управления из блока B_3 после вычисления $4*j$ в блок B_5 изменения j не происходит, так что вместо значения $4*j$ можно воспользоваться значением t_4 . Второе общее подвыражение также находится в B_5 . После того как переменная t_4 заменила переменную t_8 , новое выражение $a[t_4]$ соответствует значению $a[j]$ на уровне исходного текста. Однако в процессе перехода управления из блока B_3 в блок B_5 неизменным остается не только значение j , но и значение $a[j]$ (вычисленное во временную переменную t_5),

поскольку никаких присвоений массиву a в это время не выполняется. Следовательно, инструкции

$t_9 := a[t_4]; \quad a[t_6] := t_9$

в блоке B_5 могут быть заменены одной инструкцией

$a[t_6] := t_5$

Аналогично значение, присваиваемое x в блоке B_5 на рис. 10.66, — то же, что и значение, присвоенное временной переменной t_3 в блоке B_2 . Блок B_5 на рис. 10.7 представляет собой результат устранения общих подвыражений, соответствующих выражениям $a[i]$ и $a[j]$ на уровне исходного текста, из блока B_5 на рис. 10.66. Подобная серия преобразований на рис. 10.7 проведена и над базовым блоком B_6 .

Выражение $a[t_1]$ в блоках B_1 и B_6 на рис. 10.7 не рассматривается как общее подвыражение, хотя t_1 может быть использована в обоих случаях. После того как управление покидает блок B_1 и до попадания в блок B_6 оно может пройти через блок B_5 , в котором выполняется присвоение элементам массива a . Следовательно, значение $a[t_1]$ в блоке B_6 может отличаться от его значения в блоке B_1 , и рассматривать $a[t_1]$ как общее подвыражение небезопасно. \square

Распространение копий

Блок B_5 из рис. 10.7 может быть улучшен удалением x путем применения двух новых преобразований, касающихся присвоения вида $f := g$, именуемого *инструкцией копирования*, или, для краткости, просто *копированием*. Если пристально рассмотреть пример 10.2, мы поймем, что копирования становятся более частыми хотя бы потому, что алгоритм устранения общих подвыражений вносит в код новые инструкции копирования, как и ряд других алгоритмов. Например, на рис. 10.8 при удалении общего подвыражения в $c := d + e$ алгоритм использует новую переменную t для хранения значения $d + e$. Поскольку управление может достичь выражения $c := d + e$ как после присвоения a , так и после присвоения b , будет некорректно заменять выражение $c := d + e$ присвоением $c := a$ или $c := b$.

Идея преобразования распространения копий заключается в использовании после присвоения $f := g$ переменной g вместо f везде, где только это возможно. Например, присвоение $x := t_3$ в блоке B_5 на рис. 10.7 является копированием. Применение распространения копий к B_5 дает в результате следующий код:

```
x := t3
a[t2] := t5
a[t4] := t3
goto B2 (10.1)
```

Такое преобразование может показаться никаким не улучшением кода, но, как мы увидим позже, оно дает нам возможность удалить присвоение x .

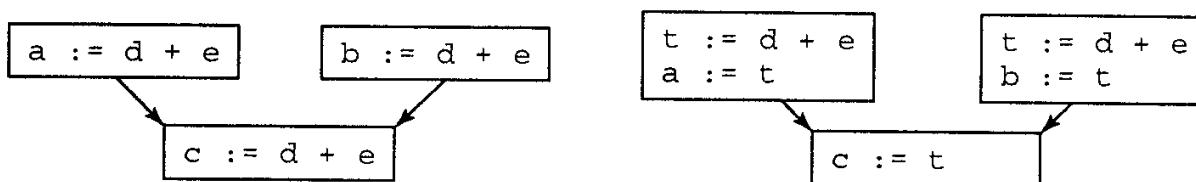


Рис. 10.8. Копирования, добавленные при удалении общего подвыражения

Удаление бесполезного кода

Переменная в некоторой точке программы считается “живой”, или активной, если ее значение будет использовано в программе в последующем; в противном случае она считается “мертвой”. Очередная идея оптимизации касается мертвого, или бесполезного кода, т.е. кода, вычисляющего значения, которые никогда не будут использованы. Хотя программист вряд ли будет создавать такой код умышленно, он может возникнуть в результате предшествующих преобразований. Например, в разделе 9.9 мы рассматривали использование переменной `debug`, которая может принимать значения `true` или `false` в различных точках программы и используется в инструкциях типа

```
if (debug) print ...
```

 (10.2)

Анализ потока данных может определить, что всякий раз по достижении данной инструкции `debug` принимает значение `false`. Обычно это происходит в силу явного присвоения `debug := false`, которое представляет собой последнее присвоение переменной `debug` перед проверкой (10.2), независимо от реальной последовательности ветвления программы. Если преобразование распространения копий заменяет `debug` значением `false`, инструкция `print` оказывается мертвой, так как не может быть достигнута. В результате мы можем удалить и проверку, и инструкцию вывода из объектного кода. В общем случае выяснение в процессе компиляции того факта, что значение выражения — константа, и использование вместо выражения этой константы называется *дублированием констант* (*constant folding*).

Одно из достоинств распространения копий состоит именно в том, что зачастую оно превращает инструкцию копирования в мертвый код. Например, распространение копий с последующим удалением бесполезного кода приводит к устраниению присвоения `x` и преобразованию (10.1) в код

```
a[t2] := t5
a[t4] := t3
goto B2
```

который представляет собой дальнейшее улучшение блока `B5` на рис. 10.7.

Оптимизации циклов

Сейчас мы вкратце рассмотрим оптимизацию очень важной части кода, а именно циклов (в особенности внутренних циклов, на которые программа затрачивает особенно много времени). Время работы программы может быть улучшено путем уменьшения количества инструкций во внутреннем цикле, даже если при этом увеличится код вне цикла. Для оптимизации циклов особенно важны три технологии: *перемещение кода* (*code motion*), которая переносит код из цикла вовне; *устранение индуктивной переменной* (*induction-variable elimination*), которую мы применим для удаления `i` и `j` из внутренних циклов `B2` и `B3` на рис. 10.7; и *снижение стоимости* (*reduction in strength*), которая замещает дорогие операции более дешевыми — например, умножение сложением.

Перемещение кода

Важным изменением, уменьшающим количество кода в цикле, является перемещение кода. Это преобразование берет из цикла выражение, приводящее к одному и тому же результату независимо от того, сколько раз выполняется цикл (*вычисление, инвариантное относительно цикла* (loop-invariant computation)), и размещает его перед циклом. Заметим, что понятие “перед циклом” предполагает существование входа в цикл. Например, вычисление $\text{limit} - 2$ является инвариантом цикла в следующей инструкции:

```
while(i <= limit - 2) /* Инструкции, не изменяющие limit */
```

Перемещение кода преобразует эту инструкцию в эквивалентный код

```
t = limit - 2;  
while(i <= t) /* Инструкции, не изменяющие limit или t */
```

Переменные индукции и снижение стоимости

Хотя перемещение кода и не применимо к рассматриваемому нами примеру быстрой сортировки, вполне возможно использование двух других технологий оптимизации. Обычно циклы обрабатываются от самого глубоко вложенного к циклам меньшей степени вложенности. Рассмотрим, например, цикл вокруг блока B_3 . На рис. 10.9 показана только та часть графа потока, которая имеет отношение к рассматриваемым преобразованиям базового блока B_3 .

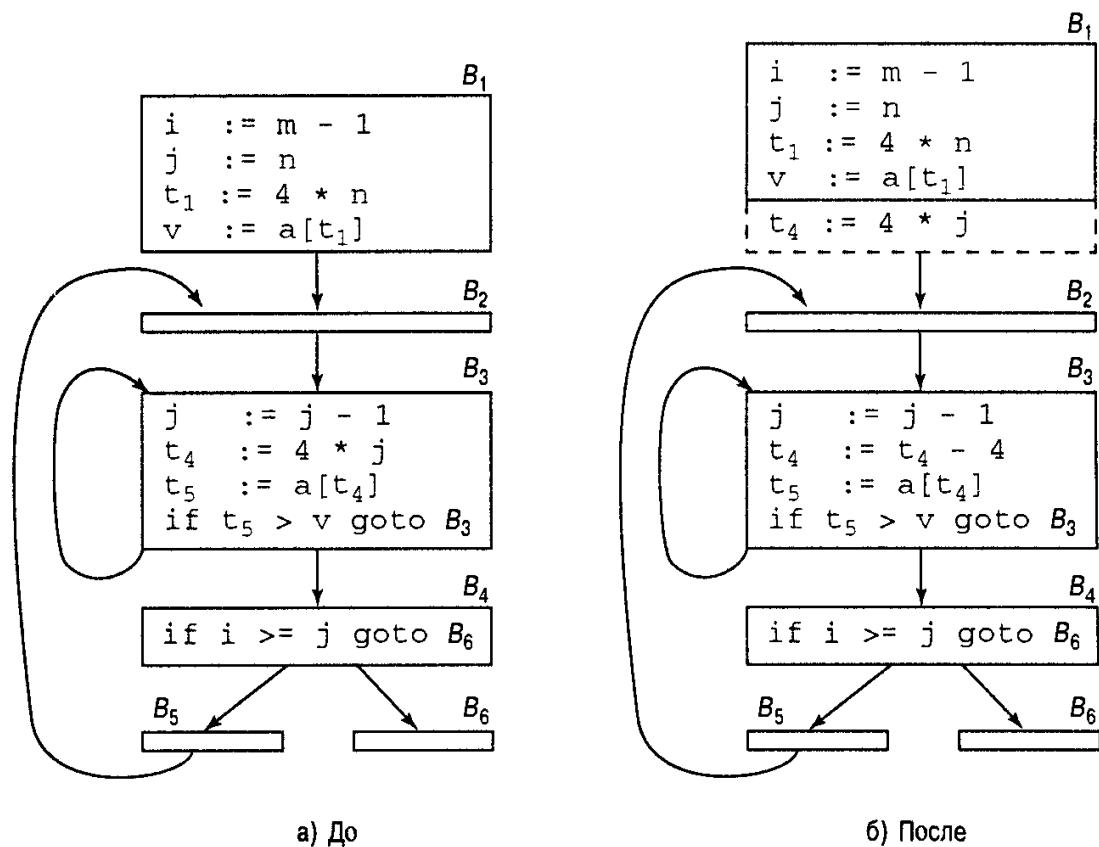


Рис. 10.9. Снижение стоимости, примененное к выражению $4 * j$ в блоке B_3

Заметьте, что значения j и t_4 взаимосвязаны и зависят от шага выполнения цикла — всякий раз, когда значение j уменьшается на 1, t_4 уменьшается на 4 (поскольку t_4 равно $4 * j$). Такие идентификаторы называются *индуктивными переменными*, или *переменными индукции* (induction variables).

При наличии в цикле нескольких переменных индукции может оказаться возможным избавиться от всех, кроме одной из них, с помощью процесса устранения индуктивных переменных. Из-за внутреннего цикла вокруг B_3 на рис. 10.9a мы не в состоянии полностью избавиться от j или t_4 , так как t_4 используется в B_3 , а j — в B_4 . Однако мы можем проиллюстрировать на этом примере технологию снижения стоимости и часть процесса устранения индуктивных переменных. В конечном счете j будет устранена при рассмотрении внешнего цикла B_2-B_5 .

Пример 10.3

Поскольку после каждого присвоения t_4 на рис. 10.9a выполняется соотношение $t_4 := 4 * j$, и t_4 не изменяется ни в каком другом месте рассматриваемого внутреннего цикла вокруг блока B_3 , отсюда следует, что непосредственно после инструкции $j := j - 1$ должно выполняться соотношение $t_4 = 4 * j - 4$. Таким образом, мы можем заменить присвоение $t_4 := 4 * j$ инструкцией $t_4 := t_4 - 4$. Единственная возникающая при этом проблема состоит в том, что при первом входе в блок B_3 переменная t_4 не имеет определенного значения. Поскольку при входе в блок B_3 должно выполняться соотношение $t_4 = 4 * j$, мы помещаем инициализирующую t_4 инструкцию в конце блока, в котором инициализируется j , что на рис. 10.9b показано обозначенным пунктиром дополнением к блоку B_1 .

Замена умножения вычитанием ускорит объектный код, если умножение выполняется медленнее, чем сложение или вычитание, что справедливо для многих машин. \square

В разделе 10.7 обсуждаются способы обнаружения переменных индукции и преобразования, которые могут быть применены к ним. Этот же раздел мы завершим еще одним примером устранения переменной индукции, в котором i и j рассматриваются в контексте внешнего цикла, содержащего B_2 , B_3 , B_4 и B_5 .

Пример 10.4

После снижения стоимости, примененного к внутренним циклам вокруг B_2 и B_3 , переменные i и j используются только в блоке B_4 для определения дальнейших действий программы. Мы знаем, что значения i и t_2 удовлетворяют соотношению $t_2 = 4 * i$, а значения j и t_4 — соотношению $t_4 = 4 * j$, так что проверка $t_2 \geq t_4$ эквивалентна проверке $i \geq j$. При выполнении такой замены переменные i и j становятся “мертвыми”, а присвоение им значений — бесполезным кодом, который может быть удален из программы, давая в результате граф потока, показанный на рис. 10.10. \square

Проведенные преобразования оказались достаточно эффективными. На рис. 10.10 количество инструкций в блоках B_2 и B_3 уменьшилось с 4 в исходном графике потока на рис. 10.5 до 3, с 9 до 3 в блоке B_5 и с 8 до 3 в блоке B_6 . Правда, блок B_1 вырос с 4 инструкций до 6, но блок B_1 выполняется только один раз, так что этот рост блока не скажется на общем времени работы программы.

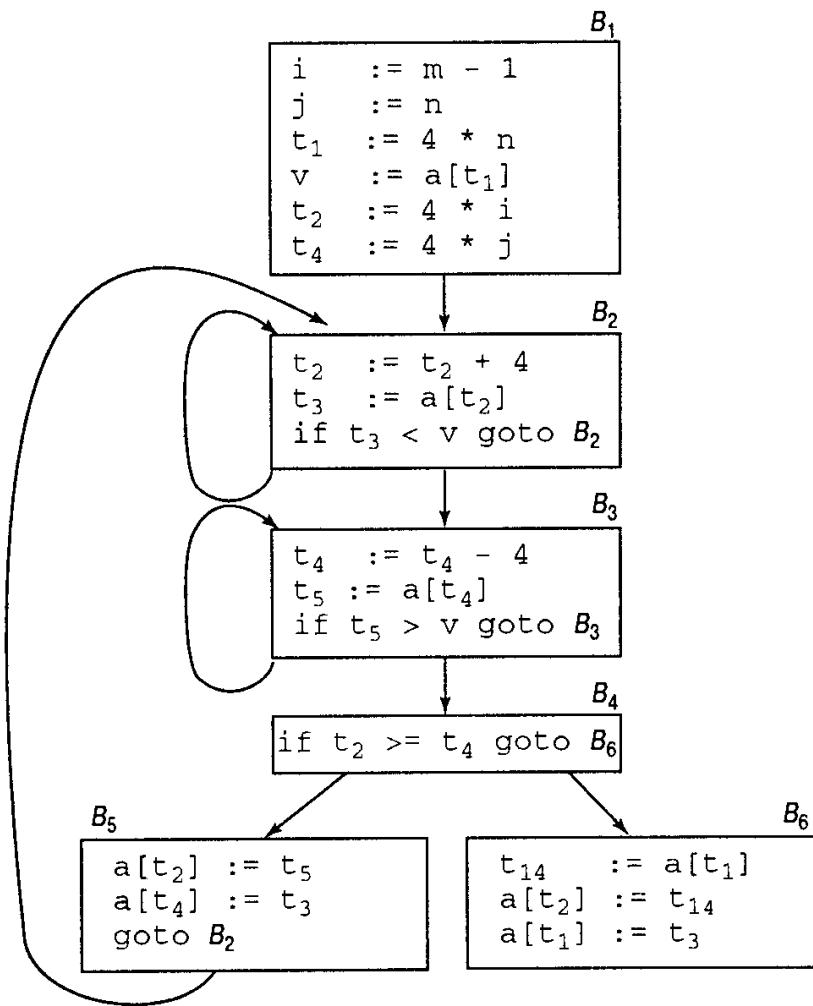


Рис. 10.10. Граф потока после устранения переменных индукции

10.3. Оптимизация базовых блоков

В главе 9, “Генерация кода”, мы встречались со множеством преобразований базовых блоков, улучшающих код. В это множество входили преобразования, сохраняющие структуру, такие как устранение общих подвыражений и бесполезного кода, и алгебраические преобразования типа снижения стоимости.

Многие из преобразований, сохраняющих структуру, могут быть реализованы построением дага для базовых блоков. Вспомните, что в даге для каждого начального значения переменной из базового блока имеется свой узел и с каждой инструкцией *s* базового блока связан узел *n* дага. Дочерние по отношению к *n* узлы соответствуют инструкциям, являющимся последними перед *s* определениями операндов, используемых в этой инструкции. Узел *n* помечается оператором, используемым в *s*; кроме того, с узлом *n* связывается список переменных, для которых этот узел является последним определением в блоке. Мы также помечаем те узлы (если такие имеются), значения которых остаются “живыми” при выходе из блока; эти узлы образуют множество выходных узлов.

Общие подвыражения могут быть обнаружены путем проверки при добавлении нового узла *m*, есть ли в даге узел *n* с такими же дочерними узлами, в том же порядке и с тем же оператором. В таком случае *n* вычисляет то же значение, что и *m*, и может использоваться вместо него.

Пример 10.5

Даг для блока (10.3)

$$\begin{aligned} a &:= b + c \\ b &:= a - d \\ c &:= b + c \\ d &:= a - d \end{aligned} \quad (10.3)$$

показан на рис. 10.11. При построении узла для третьей инструкции $c := b + c$ мы знаем, что использование b в этой инструкции приводит к обращению к узлу, помеченному на рис. 10.11 знаком “−”, поскольку это — последнее определение b . Таким образом, мы не спутаем значения, вычисленные в первой и третьей инструкциях.

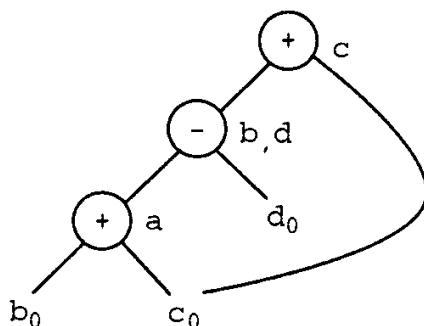


Рис. 10.11. Даг для базового блока (10.3)

Однако узел, соответствующий четвертой инструкции $d := a - d$, имеет оператор “−”, а в качестве дочерних — узлы, помеченные a и d_0 . Поскольку оператор и дочерние узлы в этом случае те же, что и для узла, соответствующего второй инструкции блока, мы не создаем новый узел, а добавляем d к списку определений для узла, помеченного знаком “−”. □

Может показаться, что поскольку в даге на рис. 10.11 имеется только три узла, блок (10.3) можно заменить блоком с тремя инструкциями. В самом деле, если либо переменная b , либо переменная d не остается активной при выходе из блока, то вычислять эту переменную нет необходимости, а для получения ее значения можно использовать вторую переменную из списка узла, помеченного на рис. 10.11 знаком “−”. Например, если на выходе из блока b не активна, можно использовать следующий код:

$$\begin{aligned} a &:= b + c \\ d &:= a - d \\ c &:= d + c \end{aligned}$$

Однако если и b , и d активны на выходе из блока, то четвертая инструкция должна использоваться для копирования значения из одной переменной в другую³.

Обратите внимание, что при поиске общих подвыражений мы в действительности ищем выражения, которые гарантированно вычисляют одно и то же значение, вне зависимости от того, каким образом вычисляется это значение. Таким образом, метод дага не заметит тот факт, что выражения, вычисляемые первой и четвертой инструкциями последовательности

³ Вообще говоря, мы должны быть внимательны при построении кода на основе дагов при выборе имен переменных, соответствующих узлам. Если переменная x определена дважды либо выполняется одно присвоение и начальное значение x_0 также используется, мы должны гарантировать, что значение x остается неизменным, пока не будут выполнены все инструкции, использующие узел, хранящий предыдущее значение x .

```

a := b + c
b := b - d
c := c + d
e := b + c

```

(10.4)

одинаковы, а именно — $b+c$. Однако эту эквивалентность можно обнаружить применением алгебраических тождеств к дагу, о чём будет рассказано позже. Даг для этой последовательности показан на рис. 10.12.

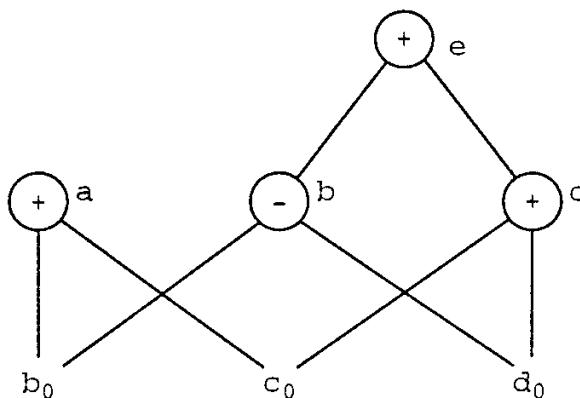


Рис. 10.12. Даг для базового блока (10.4)

Операция над дагом, соответствующая устранению бесполезного кода, реализуется очень просто. Мы удаляем из дага все корни (узлы без предков), которые не имеют живых переменных. Неоднократное применение этого преобразования удалит из дага все узлы, соответствующие бесполезному коду.

Использование алгебраических тождеств

Алгебраические тождества представляют еще один важный класс методов оптимизации базовых блоков. В разделе 9.9 мы встречались с некоторыми простыми алгебраическими преобразованиями, которые могут использоваться в процессе оптимизации. Например, мы можем воспользоваться арифметическими тождествами, такими как

$$\begin{aligned}
 x + 0 &= 0 + x = x \\
 x - 0 &= x \\
 x * 1 &= 1 * x = x \\
 x / 1 &= x
 \end{aligned}$$

Другой класс методов алгебраической оптимизации включает снижение стоимости, т.е. замену более дорогих операторов дешевыми — как в приведенных примерах:

$$\begin{aligned}
 x ** 2 &= x * x \\
 2.0 * x &= x + x \\
 x / 2 &= x * 0.5
 \end{aligned}$$

Третий класс родственных методов оптимизации — дублирование констант. В этом случае мы вычисляем константные выражения в процессе компиляции и заменяем их вычисленным значением⁴. Таким образом, выражение $2 * 3.14$ заменяется константой 6.28 . Многие константные выражения появляются при использовании символьных констант.

⁴ Арифметические выражения должны вычисляться во время компиляции так же, как и в процессе работы программы. Томпсон (K. Thompson) предложил элегантное решение, состоящее в компиляции константного выражения, выполненного фрагмента целевого кода и замене выражения полученным результатом — при этом компилятор не обязан содержать в себе интерпретатор.

Процесс построения дага может помочь нам применить приведенные и другие, более общие алгебраические преобразования, такие как коммутативность и ассоциативность. Например, рассмотрим коммутативную операцию $*$, для которой $x^*y = y^*x$. Перед созданием нового узла, помеченного “ $*$ ”, с левым дочерним узлом t и правым дочерним узлом n мы проверяем, не существует ли уже такой узел. Используя свойство коммутативности, после этого мы проверяем, нет ли такого узла, но с левым дочерним узлом n и правым дочерним узлом t .

Операторы отношения $<=$, $>=$, $<$, $>$, $=$ и \neq иногда приводят к неожиданным общим подвыражениям. Например, условие $x>y$ можно также проверить выполнением вычитания и проверкой полученного результата (впрочем, вычитание может привести к переполнению или потере значимости, чего не происходит при использовании инструкции сравнения). Таким образом, для $x-y$ и $x>y$ должен быть сгенерирован один узел дага.

Для выявления общих подвыражений может использоваться и ассоциативный закон. Например, если в исходном коде имеются присвоения

```
a := b + c  
e := c + d + b
```

может быть сгенерирован следующий промежуточный код.

```
a := b + c  
t := c + d  
e := t + b
```

Если значение t вне этого блока не требуется, данная последовательность инструкций может быть заменена следующей:

```
a := b + c  
e := a + d
```

с учетом свойств ассоциативности и коммутативности операции сложения.

Разработчик компилятора должен тщательно проверить спецификации языка для определения, какие изменения порядка вычислений разрешены, поскольку компьютерная арифметика не всегда подчиняется алгебраическим тождествам математики. Например, стандарт Fortran 77 указывает, что компилятор может вычислять любые математически эквивалентные выражения при условии сохранения порядка вычислений, определяемого скобками. Следовательно, компилятор может вычислять x^*y-x^*z как $x^*(y-z)$, но не может вычислять $(a+b)-c$ вместо $a+(b-c)$. Компилятор Fortran, таким образом, для полного соответствия определению языка должен отслеживать наличие скобок в исходном тексте.

10.4. Циклы в графах потока

Перед тем как перейти к рассмотрению оптимизации циклов, нам необходимо определить, из чего состоит цикл в графике потока. Мы используем отношение “доминирования” одного узла над другим для определения “естественных циклов” и специальный класс приводимых графов потока. Алгоритм для поиска доминаторов и проверки приводимости графов потока будет дан в разделе 10.9.

Доминаторы

Мы говорим, что узел d графа потока *доминирует* над узлом n (записывается как $d \text{ dom } n$), если любой путь из начального узла графа потока в узел n проходит через d . При таком определении каждый узел доминирует над собой, а вход в цикл (определенный в разделе 9.4) доминирует над всеми узлами цикла.

Пример 10.6

Рассмотрим граф потока (рис. 10.13) с начальным узлом 1. Начальный узел доминирует над всеми узлами. Узел 2 доминирует только над собой, поскольку управление может достичь любого другого узла по пути, начинающемуся с $1 \rightarrow 3$. Узел 3 доминирует над всеми узлами, кроме 1 и 2; узел 4 — над всеми узлами, кроме 1, 2 и 3, поскольку все пути от узла 1 должны начинаться как $1 \rightarrow 2 \rightarrow 3 \rightarrow 4$ или $1 \rightarrow 3 \rightarrow 4$. Узлы 5 и 6 доминируют только над собой, поскольку поток управления может пройти через любой из них, минуя другой. И наконец, узел 7 доминирует над 7, 8, 9, 10; узел 8 — над 8, 9, 10; узлы 9 и 10 доминируют только над собой. \square

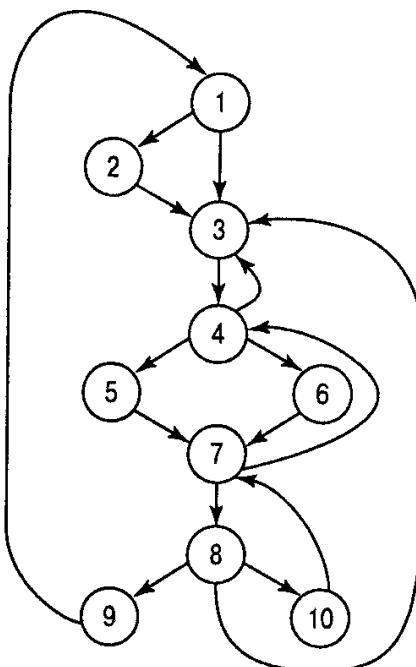


Рис. 10.13. Граф потока

Удобным представлением информации о доминаторах является дерево, называемое **деревом доминаторов**, в котором начальный узел является деревом, а каждый узел d доминирует над своими потомками в дереве. Так, на рис. 10.14 показано дерево доминаторов для графа потока, изображенного на рис. 10.13.

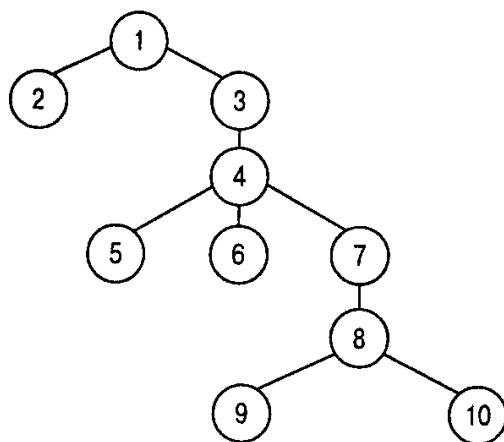


Рис. 10.14. Дерево доминаторов для графа потока на рис. 10.13

Существование дерева доминаторов следует из их свойств. Каждый узел n имеет единственный *непосредственный доминатор* m , который представляет собой последний доминатор n на любом пути от начального узла к n . Используя отношение dom , можно сказать, что непосредственный доминатор m обладает тем свойством, что если $d \neq n$ и $d \text{ dom } n$, то $d \text{ dom } m$.

Естественные циклы

Одним из важных применений информации о доминаторах является определение циклов в графе потока, пригодных для улучшения. Такие циклы имеют два важных свойства.

1. Цикл должен иметь единственную входную точку, называемую “заголовком” (header). Эта входная точка доминирует над всеми узлами в цикле, иначе она не является единственной точкой входа в цикл.
2. Должен быть как минимум один путь итерации цикла, т.е. как минимум один путь назад к заголовку.

Один из способов поиска всех циклов в графе потока состоит в поиске дуг в графе потока, у которых голова доминирует над хвостом (если $a \rightarrow b$ является дугой, то головой этой дуги является b , а хвостом — a). Такие дуги мы называем *обратными* (back edges).

Пример 10.7

На рис. 10.13 есть дуга $7 \rightarrow 4$; при этом $4 \text{ dom } 7$. Другими дугами с аналогичным свойством являются $10 \rightarrow 7$, $4 \rightarrow 3$, $8 \rightarrow 3$ и $9 \rightarrow 1$. Заметим, что это те дуги, о которых можно говорить как о формирующих циклы в графе потока. □

Для данной обратной дуги $n \rightarrow d$ мы определяем *естественный цикл* дуги как d плюс множество узлов, которые могут достигнуть n , минуя d . Узел d является заголовком цикла.

Пример 10.8

Естественный цикл дуги $10 \rightarrow 7$ состоит из узлов 7 , 8 и 10 , поскольку 8 и 10 — все узлы, которые могут достичь 10 , минуя 7 . Естественным циклом дуги $9 \rightarrow 1$ является весь граф потока (не забывайте о пути $10 \rightarrow 7 \rightarrow 8 \rightarrow 9!$). □

Алгоритм 10.1. Построение естественного цикла обратной дуги

Вход. Граф потока G и обратная дуга $n \rightarrow d$.

Выход. Множество $loop$, состоящее из всех узлов естественного цикла $n \rightarrow d$.

Метод. Начиная с узла n , мы рассматриваем каждый узел $m \neq d$ (о котором известно, что он находится в цикле), чтобы убедиться, что предшественник m уже помещен в $loop$. Алгоритм приведен на рис. 10.15. Каждый узел в $loop$, за исключением d , помещается в стек $stack$, так что его предшественники оказываются проверенными. Заметим, что, поскольку d изначально находится в цикле, мы никогда не рассматриваем его предшественников и, таким образом, находим только те узлы, которые достигают n , не проходя через d . □

```
procedure insert( $m$ )
if  $m \notin loop$  then begin
     $loop := loop \cup \{m\}$ ;
    Поместить  $m$  в  $stack$ 
end;
```

```

/* Далее следует основная программа */
stack := empty;
loop := {d};
insert(n);
while stack не пуст do begin
    Снять со стека stack первый элемент m
    for Каждого узла p, являющегося
        предшественником m do insert(p)
end

```

Рис. 10.15. Алгоритм построения естественных циклов

Внутренние циклы

Если при рассмотрении циклов в графах потока мы пользуемся естественными циклами, в наших руках оказывается то очень полезное свойство, что два цикла с разными заголовками могут быть либо разъединены, либо один из них полностью содержится (*вложен*) в другом. Таким образом, если не обращать внимания на циклы с одними и теми же заголовками, у нас есть естественное понятие *внутреннего цикла*: это цикл, не содержащий других циклов.

Когда у двух циклов совпадает заголовок, как в случае, показанном на рис. 10.16, сказать, какой из циклов внутренний, становится сложно. Например, если проверка в конце блока B_1 выглядит как

```
if a = 10 goto B2
```

то, вероятно, внутренним циклом является $\{B_0, B_1, B_3\}$. Однако без детального анализа кода мы не можем быть в этом уверены. Может быть, а почти всегда равно 10, и обычной ситуацией оказывается множество проходов по циклу $\{B_0, B_1, B_2\}$ перед попаданием в блок B_3 . Следовательно, при наличии двух естественных циклов с одним заголовком, не вложенных один в другой, они объединяются и рассматриваются как единый цикл.

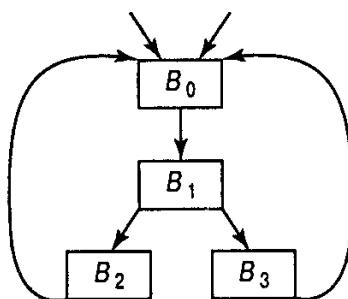


Рис. 10.16. Два цикла с одним заголовком

Предзаголовки

Ряд преобразований требует от нас вынесения инструкций “перед заголовком”. Таким образом, рассмотрение цикла L начинается с создания нового блока, именуемого *предзаголовком* (preheader). Предзаголовок имеет в качестве преемника только заголовок цикла, а все дуги, которые ранее входили в заголовок L извне цикла, теперь входят в предзаголовок. Дуги изнутри цикла, входящие в заголовок, остаются неизменными. Создание предзаголовка показано на рис. 10.17. Изначально предзаголовок пуст, однако преобразования L могут помещать в него инструкции.

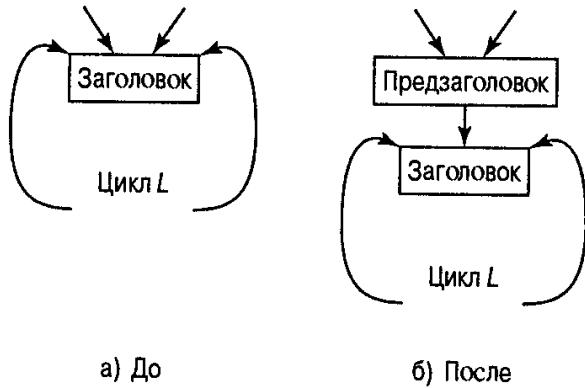


Рис. 10.17. Создание предзаголовка

Приводимые графы потоков

Графы потоков, встречающиеся на практике, часто принадлежат классу приводимых графов потока, определенному ниже. Использование только структурированных инструкций управления потоком, таких как `if-then-else`, `while-do`, `continue` и `break`, приводит к программам, графы потоков которых всегда приводимы⁵. Более того, почти всегда приводимы даже программы, написанные с применением инструкции `goto` программистами, не знакомыми с разработкой структурированных программ.

Был предложен ряд определений приводимого графа потока. То, которое примем мы, приводит к одному из наиболее важных свойств приводимых графов потока — единственный вход в цикл осуществляется через его заголовок; нет никаких переходов в середину цикла извне. Упражнения и библиографические примечания содержат краткую историю этой концепции.

Граф потока G является *приводимым* тогда и только тогда, когда мы можем разделить дуги на две непересекающиеся группы, часто именуемые *прямыми* и *обратными* дугами (forward и back edges). Эти группы обладают следующими свойствами.

1. Прямые дуги образуют ациклический граф, в котором из начального узла G может быть достигнут любой узел.
2. Обратные дуги состоят только из дуг, головы которых доминируют над хвостами.

Пример 10.9

Граф потока на рис. 10.13 приводим. В общем случае, если мы знаем отношение *dom* для графа потока, мы можем найти и удалить все обратные дуги. Если граф приводим, оставшиеся дуги должны быть прямыми и для проверки приводимости графа потока достаточно убедиться, что прямые дуги образуют ациклический граф. В случае, представленном на рис. 10.13, очень просто проверить, что мы удаляем пять обратных дуг $4 \rightarrow 3$, $7 \rightarrow 4$, $8 \rightarrow 3$, $9 \rightarrow 1$ и $10 \rightarrow 7$, головы которых доминируют над хвостами, после чего получаем ациклический граф. \square

Пример 10.10

Рассмотрим граф, показанный на рис. 10.18. Начальным узлом этого графа является узел 1. У этого графа потока нет обратных дуг, и, следовательно, он может быть приводимым только при отсутствии в нем циклов. Но поскольку это не так, данный

⁵ Отметим, что в языках типа Pascal или C использование неструктурных инструкций `goto`, мягко говоря, не приветствуется. — Прим. ред.

граф не является приводимым. Интуитивно причиной неприводимости графа является цикл 2–3,войти в который можно двумя различными путями — как через узел 2, так и через узел 3. \square

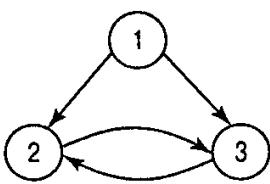


Рис. 10.18. Неприводимый граф потока

Ключевым свойством приводимого графа потока для анализа циклов является то, что в таком графе любое множество дуг, которое мы неформально рассматриваем как цикл, должно содержать обратные дуги. В действительности, чтобы найти все циклы в программе с приводимым графом потока, мы должны проверить только естественные циклы обратных дуг. Граф на рис. 10.18 выглядит как имеющий цикл, состоящий из узлов 2 и 3, однако обратной дуги, для которой этот цикл является естественным, в данном случае нет. На самом деле этот “цикл” имеет два заголовка, 2 и 3, что делает невозможным непосредственное применение многих технологий оптимизации кода, таких как описанные в разделе 10.2, — перемещение кода или устранение индуктивной переменной.

К счастью, неприводимые структуры потоков управления, наподобие приведенных на рис. 10.18, настолько редко появляются в большинстве реальных программ, что изучение циклов с несколькими заголовками не представляется важным. Более того, имеются языки (типа Bliss и Modula 2), в которых программы могут иметь исключительно приводимые графы потоков, а для многих других языков это справедливо, если мы не используем операторы безусловного перехода.

Пример 10.11

Вернемся вновь к рис. 10.13 и обратим внимание, что единственным “внутренним циклом”, т.е. циклом без подциклов является $\{7, 8, 10\}$ — естественный цикл для обратной дуги $10 \rightarrow 7$. Множество $\{4, 5, 6, 7, 8, 10\}$ представляет собой естественный цикл для обратной дуги $7 \rightarrow 4$ (заметим, что 8 и 10 могут достичь узел 7 по дуге $10 \rightarrow 7$). Наша интуиция, подсказывающая, что $\{4, 5, 6, 7\}$ тоже является циклом, на этот раз нас обманывает, поскольку и 4, и 7 могут быть входами извне, что нарушает правило единственности входа в цикл. Однако нет причин полагать, что программа затратит очень много времени на обход узлов $\{4, 5, 6, 7\}$ — ведь можно предположить, что переход из узла 7 в узел 8 совершается не менее часто, чем в узел 4. Включая в цикл узлы 8 и 10, мы более надежно изолируем труднопроходимый участок от остальной части программы.

Следует, однако, четко представлять опасность неверных предположений о частотах ветвлений. Например, если мы перенесем инвариантные инструкции из узлов 8 или 10 цикла $\{7, 8, 10\}$ и при этом окажется, что управление чаще передается по пути $7 \rightarrow 4$, чем по дуге $7 \rightarrow 8$, мы на деле увеличим количество выполнений перемещенных инструкций. Методы избежания этой проблемы будут описаны в разделе 10.7.

Следующий, больший по размеру цикл представляет собой множество узлов $\{3, 4, 5, 6, 7, 8, 10\}$ и является естественным циклом для дуг $4 \rightarrow 3$ и $8 \rightarrow 3$. Как и ранее, наша интуиция подводит нас, воспринимая $\{3, 4\}$ как цикл, поскольку в данном случае также нарушается условие единственности входа. Последний цикл для обратной дуги $9 \rightarrow 1$ представляет собой весь граф потока. \square

У приводимых графов потока имеется ряд других полезных свойств, о которых мы поговорим при рассмотрении вопросов поиска в глубину и анализа интервалов в разделе 10.9.

10.5. Введение в глобальный анализ потоков данных

Для оптимизации кода и генерации эффективного кода компилятор нуждается в сборе информации о программе в целом и каждом блоке графа потока в отдельности. Например, в разделе 9.7 мы видели, что знание об активности тех или иных переменных при выходе из блока позволяет улучшить использование регистров. В разделе 10.2 предполагалось использование знаний о глобальных общих подвыражениях для устранения излишних вычислений. В разделах 9.9 и 10.3 обсуждались вопросы определения достижимости кода, в частности определение того, когда в последний раз устанавливалось значение переменной `debg` до достижения данного блока, чтобы на основе этой информации попытаться выполнить преобразования типа дублирования констант и устранения бесполезного кода. Перечисленное — всего лишь несколько примеров *информации о потоке данных*, которая собирается оптимизирующим компилятором с помощью процесса, известного как *анализ потока данных*.

Информация о потоке данных может быть собрана путем создания и решения системы уравнений, связывающей информацию в разных точках программы. Типичное уравнение имеет вид

$$out[S] = gen[S] \cup (in[S] - kill[S]) \quad (10.5)$$

и может быть прочитано следующим образом: информация в конце инструкции либо генерируется в инструкции, либо приходит извне в начале инструкции и не уничтожается в процессе прохода управления по ней. Такие уравнения называются *уравнениями потока данных*.

Детали создания и решения уравнений потока данных зависят от трех факторов.

1. Понятия генерации и уничтожения зависят от интересующей нас информации, т.е. от того, какая именно задача анализа потока данных решается. Кроме того, для некоторых задач вместо “течения по потоку управления” и определения $out[S]$ на основе $in[S]$ мы вынуждены идти в обратном направлении и определять $in[S]$ через $out[S]$.
2. Поскольку потоки данных следуют вместе с потоками управления, на их анализ влияют управляющие инструкции программы. В действительности при записи $out[S]$ мы неявно предполагаем, что имеется единственная точка выхода из инструкции; в общем случае уравнения определяются на уровне базовых блоков, а не инструкций, поскольку базовые блоки имеют единственные выходные точки.
3. Дополнительные тонкости связаны с такими инструкциями, как вызовы процедур, присвоение посредством указателей и присвоение переменным массивов.

В этом разделе мы рассмотрим задачу нахождения множества определений, достигающих некоторой точки программы, и его использование при поиске возможностей дублирования констант. Позже в этой главе ту же информацию будут использовать алгоритмы перемещения кода и устранения переменных индукций.

Сначала мы рассмотрим программы, построенные с применением инструкций `if` и `do-while`. Предсказуемость потока управления в этих инструкциях позволяет нам сосредоточиться на идеях, необходимых для формирования и решения уравнений потока данных.

Присвоения в этом разделе либо являются инструкциями копирования, либо имеют вид $a := b+c$. В этой главе мы зачастую используем “+” как типичный оператор. Все, что мы говорим о нем, непосредственно переносимо и на другие операторы, включая операторы с одним операндом либо с более чем двумя операндами.

Точки и пути

Внутри базового блока мы говорим о *точке* между двумя соседними инструкциями, а также о точках перед первой и после последней инструкции. Следовательно, в блоке B_1 на рис. 10.19 содержится четыре точки: по одной после каждой инструкции присвоения и одна перед всеми тремя присвоениями.

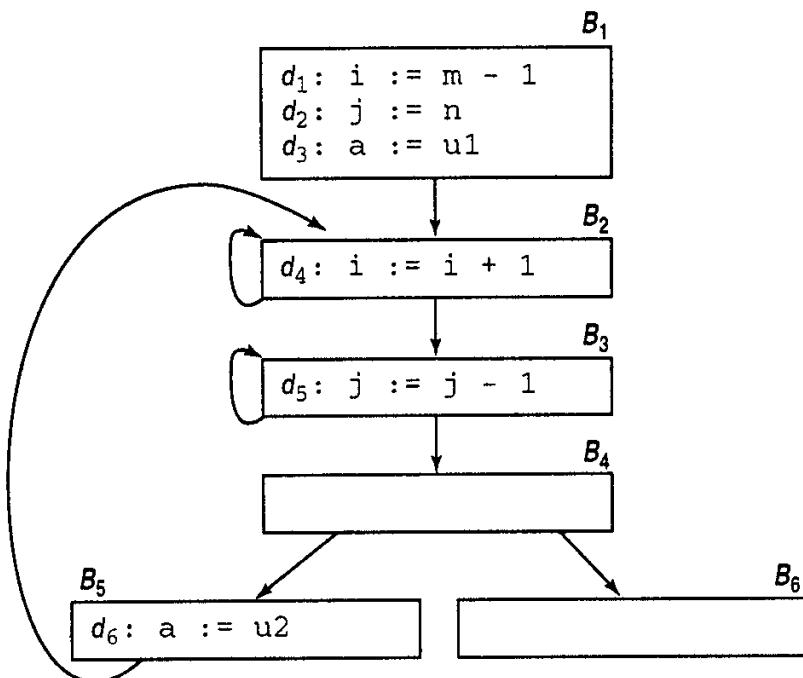


Рис. 10.19. Граф потока

Теперь глобально рассмотрим все точки во всех блоках. Путем от p_1 к p_n называется последовательность точек p_1, p_2, \dots, p_n , такая, что для каждого i между 1 и $n-1$

1. либо p_i является точкой, непосредственно предшествующей инструкции, а p_{i+1} — точкой, непосредственно следующей за этой инструкцией в том же блоке,
2. либо p_i представляет собой конец некоторого блока, а p_{i+1} — начало следующего блока.

Пример 10.12

На рис. 10.19 имеется путь от начала блока B_5 к началу блока B_6 . Он проходит от конечной точки B_5 через все точки блоков B_2, B_3 и B_4 перед тем, как достичь начала блока B_6 . \square

Достигающие определения

Определение (definition) некоторой переменной x представляет собой инструкцию, которая присваивает или может присвоить значение переменной x , а также инструкцию чтения значения из устройства ввода-вывода и сохранения его в перемен-

ной x . Такие инструкции точно определяют значение x и называются *однозначными определениями* x . Имеется также ряд инструкций другого типа, которые могут определять значение x . Чаще всего встречаются следующие типы таких *неоднозначных определений* x .

1. Вызов процедуры с передачей x в качестве параметра (в том случае, когда передача осуществляется не по значению) либо процедуры, которая имеет доступ к x в силу того, что x находится в области видимости процедуры. Кроме того, мы должны рассмотреть случай псевдонимов, когда x идентифицировано с другой переменной, передаваемой в качестве параметра или находящейся в области видимости. Эти вопросы рассматриваются в разделе 10.8.
2. Присвоение посредством указателя, который может указывать на x . Например, присвоение $*q := y$ является определением x , если q может указывать на x . Методы определения того, куда может указывать указатель, также рассматриваются в разделе 10.8, но при отсутствии информации мы должны предполагать, что присвоение через указатель является определением каждой переменной.

Мы говорим, что определение d *достигает* точки p , если имеется путь от точки, непосредственно следующей за d , к p , такой, что d не уничтожается вдоль этого пути. Интуитивно, если определение d некоторой переменной a достигает точки p , то d может быть местом, где последний раз определяется a , используемое в p . Мы *уничтожаем* определение a , если между двумя точками вдоль пути имеется определение a . Заметим, что только однозначное определение a уничтожает другое определение a (таким образом, точка может быть достигнута однозначным и неоднозначным определениями той же переменной, встречающейся вдоль пути после однозначного).

Например, на рис. 10.19 определения $i := m-1$ и $j := n$ в блоке B_1 достигают начала блока B_2 — так же, как и определение $j := j-1$, при условии, что в блоках B_4 , B_5 и оставшейся части блока B_3 , следующей за определением, нет ни присвоений j , ни чтения значения этой переменной. Заметим, однако, что присвоение j в блоке B_3 уничтожает определение $j := n$, так что последнее не достигает B_4 , B_5 или B_6 .

Имеющееся у нас определение достижимости иногда допускает определенные неточности, но все они “безопасны”. Например, заметим, что наше предположение о том, что могут быть пройдены все пути графа потока, может на практике не выполняться. Так, для любых значений a и b управление не в состоянии достичь присвоения $a := 4$ в следующем фрагменте:

```
if a = b then a := 2
else if a = b then a := 4
```

В общем случае задача о доступности всех путей графа потока неразрешима, так что мы не будем пытаться ее решать.

Один из важнейших принципов при разработке преобразований, улучшающих код, состоит в том, что в случае малейших сомнений мы должны принимать только консервативные решения. Решение является *консервативным*, если оно никогда не приводит к изменению результата работы программы. В применении к определению достижимости консервативное решение состоит в предположении, что определение может достичь некоторой точки, даже если оно ее не достигает. Таким образом, мы допускаем наличие путей, которые не могут быть пройдены при выполнении программы, и разрешаем определениям проходить через неоднозначные определения той же переменной.

Анализ потока данных структурированной программы

Графы потоков для управляющих инструкций типа **do-while** обладают полезным свойством: у них есть одна начальная точка, через которую поток управления входит в конструкцию, и одна конечная, через которую управление покидает конструкцию после ее выполнения. Мы используем это свойство, когда говорим об определениях, достигающих начала и конца инструкций со следующим синтаксисом

$$\begin{aligned} S &\rightarrow \text{id} := E \mid S ; S \mid \text{if } E \text{ then } S \text{ else } S \mid \text{do } S \text{ while } E \\ E &\rightarrow \text{id} + \text{id} \mid \text{id} \end{aligned}$$

Выражения в таком языке аналогичны выражениям в промежуточном коде, однако графы потоков инструкций могут быть только определенных типов, показанных на рис. 10.20. Основная цель этого раздела — изучение уравнений потока данных, подытоженных на рис. 10.21.

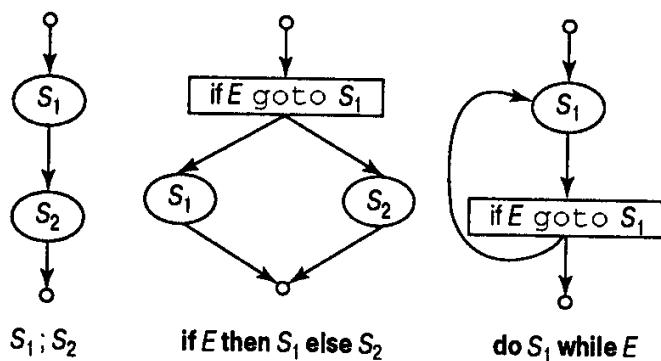


Рис. 10.20. Некоторые структурированные конструкции

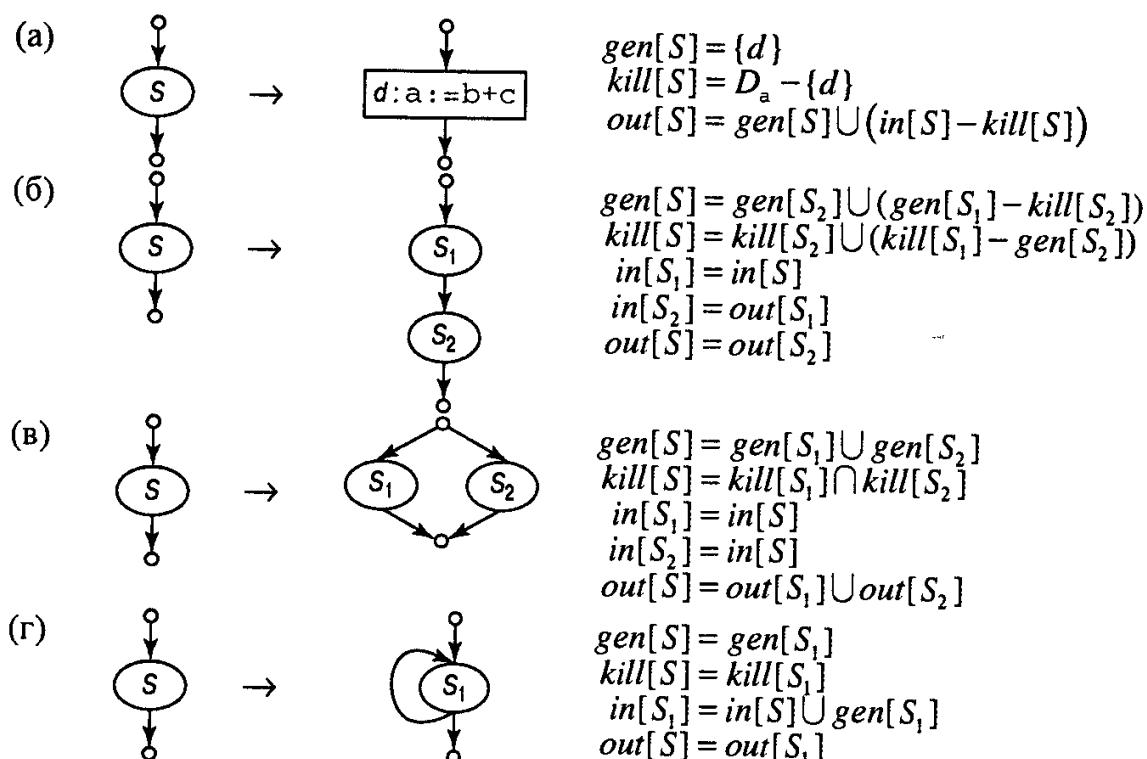


Рис. 10.21. Уравнения потока данных для достигающих определений

Область (region) в графике потока мы определяем как множество узлов N , включающее заголовок (header), который доминирует над всеми узлами области. Все дуги между уз-

лами N находятся в области, возможно, за исключением некоторых, входящих в заголовок⁶. Часть графа потока, соответствующая инструкции S , представляет собой область, при выходе из которой поток управления может попасть только в один внешний блок.

Для удобства мы полагаем, что непосредственно перед входом в область и выходом из нее поток управления проходит через искусственные блоки, не содержащие инструкций (они указаны на рис. 10.20 маленькими кружками). Мы говорим, что начальные точки этих искусственных блоков на входе в область и выходе из нее являются соответственно *начальной* и *конечной* точками инструкции.

Уравнения на рис. 10.21 представляют собой индуктивное, или синтаксически управляемое, определение множеств $in[S]$, $out[S]$, $gen[S]$ и $kill[S]$ для всех инструкций S . Множества $gen[S]$ и $kill[S]$ являются синтезируемыми атрибутами; они вычисляются снизу вверх, от наименьших инструкций к большим. Мы хотим, чтобы определение d находилось в $gen[S]$, если d достигает конца S , независимо от того, достигает ли оно начала S или нет. Иначе говоря, d должно присутствовать в S и достигать конца S по пути, не выходящему за пределы S (это дает основание для того, чтобы говорить, что $gen[S]$ является множеством определений, “сгенерированных инструкцией S ”).

Аналогично $kill[S]$ представляет собой множество определений, которые никогда не достигают конца S , даже если достигают его начала. Следовательно, имеет смысл рассматривать эти определения как “уничтоженные инструкцией S ”. Для того чтобы определение d попало в $kill[S]$, каждый путь от начала до конца S должен иметь однозначное определение той же переменной, что и в d , и если d находится в S , то на каждом пути после определения d должно быть еще одно определение той же переменной⁷.

Правила для gen и $kill$ относительно просты для понимания. Для начала рассмотрим правила на рис. 10.21a для отдельного присвоения переменной a . Несомненно, присвоение является определением переменной a (назовем его определением d). Тогда d является единственным определением, гарантированно достигающим конца инструкции, независимо от того, достигает ли оно начала. Следовательно, $gen[S] = \{d\}$. Однако d уничтожает все другие определения a , так что мы можем записать $kill[S] = D_a - \{d\}$, где D_a — множество всех определений переменной a в программе.

Правило для каскада инструкций, показанного на рис. 10.21b, немного более тонкое. При каких условиях определение d генерируется инструкцией $S = S_1 ; S_2$? Прежде всего, если оно генерируется S_2 , то оно, конечно, генерируется S . Если d генерируется S_1 , то оно достигнет конца S , если не будет уничтожено в S_2 . Следовательно, мы можем записать

$$gen[S] = gen[S_2] \cup (gen[S_1] - kill[S_2])$$

Применяя аналогичные рассуждения к уничтожению определений, получим

$$kill[S] = kill[S_2] \cup (kill[S_1] - gen[S_2])$$

Для инструкции if , показанной на рис. 10.21c, мы можем заметить, что если любая из ветвей “ if ” генерирует определение, то оно достигает конца S . Следовательно, $gen[S] = gen[S_1] \cup gen[S_2]$. Однако для уничтожения определения d переменная, определяемая d , должна уничтожаться на любом из путей от начала S к концу. В частности, она должна быть уничтожена в каждой из двух ветвей, так что $kill[S] = kill[S_1] \cap kill[S_2]$.

⁶ Цикл представляет собой специальный случай строго связанной области, все обратные дуги которой входят в заголовок.

⁷ В этом вводном разделе мы полагаем, что все определения однозначны. В разделе 10.8 мы встретимся с модификациями, необходимыми для работы с неоднозначными определениями.

И наконец, рассмотрим правила для цикла, показанного на рис. 10.21г. Попросту говоря, цикл никак не влияет на множества gen и $kill$. Если определение d генерируется в S_1 , то оно достигает и конца S_1 , и конца S . И обратно, если d генерируется в S , то оно генерируется только в S_1 . Если d уничтожается в S_1 , то проход по циклу ничем помочь не может — переменная определения d будет переопределяться в S_1 при каждом проходе. И наоборот, если d уничтожается в S , то оно должно уничтожаться в S_1 . Следовательно, $gen[S] = gen[S_1]$ и $kill[S] = kill[S_1]$.

Консервативная оценка информации потока данных

В правилах для вычисления множеств gen и $kill$ на рис. 10.21 имеется один тонкий просчет. Мы основывались на предположении, что условное выражение E в инструкциях `if` и `do` “неинтерпретируемо”, т.е. существуют такие входные данные, которые обеспечивают проход по каждой из имеющихся ветвей. Говоря другими словами, мы полагаем, что любой путь в графе потока является также *путем выполнения*, т.е. таким, что имеется по крайней мере один набор входных данных, при запуске программы с которым этот путь будет выполнен.

Однако это условие выполняется не всегда, и в действительности в общем случае мы не в состоянии определить, будет ли пройдена та или иная ветвь. Предположим, что выражение E в инструкции `if` всегда истинно. В таком случае путь через S_2 на рис. 10.21в никогда не будет пройден. Отсюда вытекают два следствия. Во-первых, определение, генерируемое S_2 , в действительности не генерируется инструкцией S , поскольку из начала S невозможно попасть в S_2 . Во-вторых, ни одно из определений в $kill[S_1]$ не может достичь конца S ; следовательно, каждое такое определение должно входить в множество $kill[S]$, даже если оно не входит в $kill[S_2]$.

При сравнении вычисленного множества gen с “истинным” множеством gen мы найдем, что истинное gen всегда является подмножеством вычисленного множества gen . Однако, истинное множество $kill$ является надмножеством вычисленного $kill$. Те же соотношения выполняются и при рассмотрении других правил на рис. 10.21. Например, если выражение E в конструкции `do-S-while-E` никогда не будет ложно, мы никогда не выйдем из цикла. Следовательно, истинное $gen = \emptyset$, и циклом уничтожается любое определение. Случай последовательных инструкций, показанный на рис. 10.21б, где из-за наличия бесконечного цикла нельзя выйти из S_1 или S_2 , мы оставляем читателю в качестве упражнения.

Естественно, возникает вопрос — представляют ли эти отличия истинных и вычисленных gen и $kill$ серьезное препятствие для анализа потока данных. Ответ зависит от предполагаемого использования этих данных. В случае достигающих определений эта информация обычно используется для выяснения, ограничивается ли в данной точке переменная x несколькими возможными значениями. Например, если мы выясняем, что единственное определение x , достигающее заданной точки, имеет вид $x := 1$, мы можем сделать вывод, что в данной точке x равно 1, и заменить ссылки на x ссылками на 1.

Следовательно, переоценка множества определений, достигающих интересующей нас точки, не выглядит серьезной проблемой; в худшем случае она не даст нам выполнить оптимизацию там, где на самом деле это вполне возможно. В тоже время недооценка множества определений представляет собой фатальную ошибку; она способна привести нас к таким преобразованиям, которые вызовут изменения в том, что вычисляет программа. Например, мы можем прийти к выводу, что в данной точке все определения x дают значение 1, и заменить x этим значением, хотя на самом деле ненайденное нами

определение может давать переменной x значение 2. Нетрудно представить последствия внесенной нами замены в этом случае. Таким образом, для случая достигающих определений мы называем их множество *безопасным* или *консервативным*, если оценка представляет собой надмножество (не обязательно собственное надмножество) истинного множества достигающих определений. Если же оценка не обязательно представляет собой надмножество, мы говорим о *небезопасной* оценке.

Для каждой задачи анализа потока данных следует изучить влияние неточных оценок на возможные типы вносимых в программу изменений. В общем случае мы принимаем *безопасные отклонения* (в том смысле, что они могут запретить нам проведение оптимизации там, где она вполне возможна) и отвергаем *небезопасные* (в том смысле, что они могут привести к “*оптимизации*”, меняющей наблюдаемое поведение программы). В каждой конкретной задаче анализа потока данных безопасным обычно является либо подмножество, либо надмножество истинного множества (но не оба одновременно).

Возвращаясь теперь к влиянию безопасности на оценки множеств определений *gen* и *kill*, заметим, что наши отклонения — надмножества для *gen* и подмножества для *kill* — являются безопасными. Интуитивно увеличение *gen* вносит во множество новые определения, которые могут достичь данной точки, и не препятствует ее достижению теми определениями, которые реально делают это. Аналогично уменьшение множества *kill* в состоянии только увеличить количество определений, достигающих интересующей нас точки.

Вычисление *in* и *out*

Многие задачи анализа потока данных могут быть решены путем синтезируемых трансляций, подобных используемым для вычисления *gen* и *kill*. Например, для каждой инструкции S мы можем захотеть найти множество переменных, определяемых в ней. Такую информацию можно получить путем вычислений, аналогичных вычислениям для *gen*, причем аналог множества *kill* в этом случае нам не требуется. Эта информация может использоваться, например, для определения инвариантов цикла.

Однако имеется информация и другого вида — так, в использованной нами в качестве примера задаче о достигающих определениях нам требуется вычислить некоторые наследуемые атрибуты. Оказывается, *in* является наследуемым атрибутом, а *out* — зависящим от него синтезируемым атрибутом. Нам требуется, чтобы $in[S]$ было множеством определений, достигающих начала S , с учетом потока управления во всей программе, в том числе и во внешних по отношению к S инструкциях или таких, по отношению к которым S является вложенной. Аналогично определяется для конца S множество *out*. При этом важно обратить внимание на разницу между множествами *out*[S] и *gen*[S]. Последнее представляет собой множество определений, которые достигают конца S без учета дальнейших путей вне S .

В качестве простого примера такого различия рассмотрим последовательность инструкций на рис. 10.21б. Определение d может быть сгенерировано в S_1 и, таким образом, достичь начала S_2 . Если $d \in kill[S_2]$, d достигает конца S_2 и, таким образом, оказывается в *out*[S_2]. Однако $d \notin gen[S_2]$.

После восходящего вычисления *gen*[S] и *kill*[S] для всех инструкций S мы можем вычислить *in* и *out*, начиная с инструкции S_0 , представляющей программу в целом, с учетом того, что для всей программы $in[S_0] = \emptyset$ (т.е. определений, достигающих начала программы, не существует). Для каждого из четырех типов инструкций, показанных на рис. 10.21, мы считаем, что *in*[S] известно, и должны использовать его для вычисления *in*

каждой из подынструкций S (что тривиально в случаях (б)–(г) и не требуется для случая (а)). Затем мы рекурсивно (сверху вниз) для каждой из подынструкций S_1 или S_2 вычисляем out и используем эти множества для вычисления $out[S]$.

Простейшим случаем является показанный на рис. 10.21 a , где инструкция является присвоением. Полагая, что нам известно $in[S]$, мы можем вычислить $out[S]$ согласно (10.5), т.е.

$$out[S] = gen[S] \cup (in[S] - kill[S]).$$

Иными словами, определение достигает конца S , если оно либо генерируется S (т.е. это определение является инструкцией в S), либо достигает начала S и не уничтожается в нем.

Предположим, что мы вычислили $in[S]$ и S представляет собой последовательность из двух инструкций $S_1 ; S_2$ (второй случай на рис. 10.21). Мы начинаем с того, что $in[S_1] = in[S]$. Затем мы рекурсивно находим $out[S_1]$, что дает нам $in[S_2]$ (поскольку определение достигает начала S_2 тогда и только тогда, когда оно достигает конца S_1). Теперь мы можем рекурсивно вычислить множество $out[S_2]$, которое равно $out[S]$.

Рассмотрим теперь конструкцию if на рис. 10.21 b . Так как, в соответствии со сказанным ранее, мы консервативно предполагаем, что управление может следовать по любому из путей, то определение достигает начала S_1 или S_2 тогда и только тогда, когда достигает начала S , т.е.

$$in[S_1] = in[S_2] = in[S].$$

Кроме того, из диаграммы на рис. 10.21 b следует, что определение достигает конца S тогда и только тогда, когда оно достигает конца одной или обеих подынструкций, т.е.

$$out[S] = out[S_1] \cup out[S_2].$$

Таким образом, мы можем использовать эти уравнения для вычисления $in[S_1]$ и $in[S_2]$ по $in[S]$, рекурсивно вычислить $out[S_1]$ и $out[S_2]$, а затем использовать их для вычисления $out[S]$.

Работа с циклами

Последний случай, показанный на рис. 10.21 c , представляет отдельную проблему. Предположим, что у нас есть $gen[S_1]$ и $kill[S_1]$, вычисленные снизу вверх, и что нам дано множество $in[S]$, как в случае обхода дерева вглубь. В отличие от случаев (б) и (в), мы не можем использовать в качестве $in[S_1]$ множество $in[S]$, поскольку определения внутри S_1 , которые достигают конца S_1 , могут проследовать по обратной дуге в начало S_1 и, таким образом, также оказаться в $in[S_1]$. Так что мы имеем

$$in[S_1] = in[S] \cup out[S_1]. \tag{10.6}$$

Кроме того, у нас есть очевидное уравнение для $out[S]$,

$$out[S] = out[S_1],$$

которое мы сможем использовать после вычисления $out[S_1]$. Однако мы не можем вычислить $in[S_1]$ в соответствии с (10.6) до тех пор, пока не будем знать $out[S_1]$, в то время как наш план состоит в вычислении out по предварительно определенному in .

К счастью, имеется непосредственный способ выразить out через in ; это — уравнение (10.5), или, в нашем конкретном случае,

$$out[S_1] = gen[S_1] \cup (in[S_1] - kill[S_1]). \tag{10.7}$$

Важно понимать, что означает это уравнение. В действительности мы не знаем, справедливо ли (10.7) для произвольной инструкции S_1 ; мы только предполагаем, что оно должно быть истинно, поскольку “имеет смысл”, что определение должно достичь конца инструкции тогда и только тогда, когда оно либо генерируется внутри инструкции, либо достигает начала и не уничтожается. Однако единственный известный нам путь вычисления out — в соответствии с уравнениями, приведенными на рис. 10.21 a – e . Мы принимаем (10.7) и выводим уравнения для in и out на рис. 10.21 g . Затем мы можем использовать уравнения на рис. 10.21 a – g для доказательства того, что (10.7) выполняется для произвольной инструкции S_1 . Затем мы можем объединить эти доказательства для того, чтобы индукцией по размеру инструкции S доказать, что (10.7) и все уравнения, представленные на рис. 10.21, выполняются как для S , так и для всех подинструкций S . Мы не приводим эти доказательства здесь, оставив их в качестве упражнения.

Но даже приняв (10.6) и (10.7), нам не удается выбраться из нашего затруднения. Эти два уравнения определяют рекуррентную зависимость для $in[S_1]$ и $out[S_1]$ друг от друга. Запишем эти уравнения как

$$\begin{aligned} I &= J \cup O \\ O &= G \cup (I - K) \end{aligned} \tag{10.8}$$

где I , O , J , G и K соответствуют $in[S_1]$, $out[S_1]$, $in[S]$, $gen[S_1]$ и $kill[S_1]$. Первые два идентификатора представляют собой переменные, а остальные три — константы.

Для решения (10.8) положим, что $O = \emptyset$. Тогда мы можем использовать первое уравнение в (10.8) для вычисления оценки I , т.е.

$$I^1 = J.$$

Затем мы можем использовать второе уравнение для получения улучшенной оценки O :

$$O^1 = G \cup (I^1 - K) = G \cup (J - K).$$

Применение первого уравнения к этой новой оценке O дает нам

$$I^2 = J \cup O^1 = J \cup G \cup (J - K) = J \cup G.$$

Если мы вновь применим второе уравнение, то получим новую оценку O :

$$O^2 = G \cup (I^2 - K) = G \cup (J \cup G - K) = G \cup (J - K)$$

Заметьте, что $O^2 = O^1$. Следовательно, вычисление новой оценки I даст I^1 , новая оценка O будет равна O^1 и т.д. Таким образом, предельные значения I и O равны приведенным выше I^1 и O^1 , и мы получаем уравнения на рис. 10.21 g :

$$\begin{aligned} in[S_1] &= in[S] \cup gen[S_1] \\ out[S] &= out[S_1] \end{aligned}$$

Первое из этих уравнений следует из приведенных выше выкладок; второе — из рассмотрения графа на рис. 10.21 g .

Осталась одна невыясненная деталь — на каком основании мы начали с оценки $O = \emptyset$? Вспомним, что в нашем обсуждении консервативных оценок мы пришли к выводу о том, что множества типа $out[S_1]$, каковое и обозначено в нашем случае через O , лучше переоценить, чем недооценить. В действительности, если бы мы начали наше рассмотрение с $O = \{d\}$, где d — определение, отсутствующее в J , G или K , то d “раздувало” бы предельные значения I и O .

Здесь мы должны обратиться к предполагаемому значению in и out . Если такое d действительно принадлежат $in[S_1]$, то, где бы это d ни находилось, от него к началу S_1

должен быть путь, показывающий, как d достигает этой точки. Если d находится вне S , то оно должно принадлежать $in[S]$, тогда как если d располагается в S (а следовательно, и в S_1), то оно должно находиться в $gen[S_1]$. В первом случае $d \in J$ и, в соответствии с (10.8), $d \in I$. Во втором случае $d \in G$ и вновь попадает в I через O , согласно (10.8). Отсюда вытекает, что слишком малая начальная оценка и восходящее построение путем присоединения большего числа определений к I и O представляют собой безопасный способ оценки $in[S_1]$.

Представление множеств

Множества определений, такие как $gen[S]$ и $kill[S]$, могут быть компактно представлены с помощью битовых векторов. В графе потока мы присваиваем номер каждому интересующему нас определению, и тогда вектор, представляющий множество, будет иметь 1 в позиции i тогда и только тогда, когда множество содержит определение номер i .

Номер инструкции определения можно получить как индекс инструкции в массиве указателей на них. Однако в процессе анализа глобального потока данных нас могут интересовать не все определения. Например, не требуется присваивать номер временным переменным, используемым только в пределах одного блока (каковыми являются большинство временных переменных, генерируемых для вычисления выражений). Итак, номера интересующих нас определений обычно будут записываться в отдельную таблицу.

Представление множеств в виде битовых векторов позволяет эффективно реализовать операции со множествами. Объединение и пересечение двух множеств реализуется по-битовыми логическими операциями **or** и **and** соответственно, которые в большинстве системно-ориентированных языков программирования являются базовыми. Разность множеств A и B можно реализовать как операцию **and** над вектором A и дополнением множества B : $A - B = A \wedge \neg B$.

Пример 10.13

На рис. 10.22 показана программа с семью определениями, указанными как d_1-d_7 в комментариях слева. Битовые векторы, представляющие множества gen и $kill$ для инструкций на рис. 10.22, показаны слева от узлов синтаксического дерева на рис. 10.23. Эти множества были вычислены с применением уравнений потока данных на рис. 10.21 к инструкциям, представленным узлами синтаксического дерева.

```
/* d1 */ i := m - 1;
/* d2 */ j := n;
/* d3 */ a := u1;
           do
/* d4 */         i := i + 1;
/* d5 */         j := j - 1;
                   if e1 then
                     a := u2
                   else
/* d7 */         i := u3
           while e2
```

Рис. 10.22. Программа для иллюстрации достигающих определений

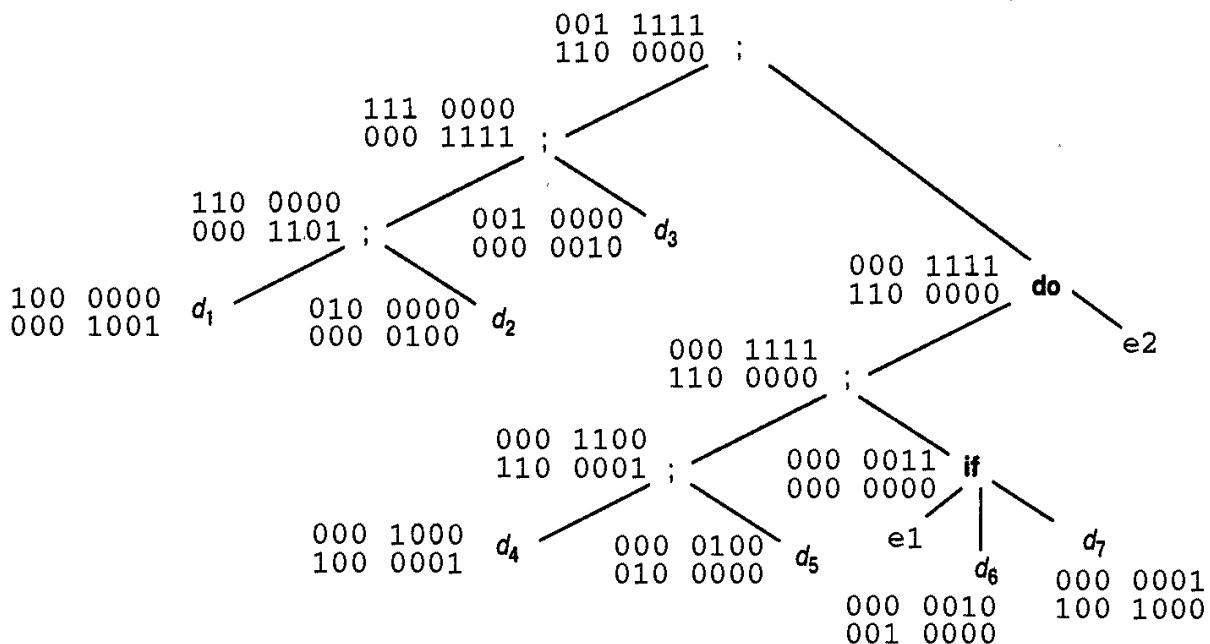


Рис. 10.23. Множества gen и kill в узлах синтаксического дерева

Рассмотрим узел для определения d_7 в нижнем правом углу рис. 10.23. Множество $gen(\{d_7\})$ представлено как 000 0001, а множество $kill(\{d_1, d_4\})$ — как 100 1000. Таким образом, d_7 уничтожает все другие определения своей переменной i . Второй и третий потомки узла if представляют собой соответственно части then и else условной инструкции. Заметим, что множество gen в узле if (000 0011) представляет собой объединение множеств 000 0010 и 000 0001 второго и третьего дочерних узлов. Множество $kill$ пустое, поскольку определения, уничтожаемые частями then и else, не пересекаются.

Уравнения потока данных для последовательных инструкций, примененные к родительскому по отношению к if узлу дают его множество kill:

$$000\ 0000 \cup (110\ 0001 - 000\ 0011) = 110\ 0000$$

Другими словами, конструкция if не уничтожает никакие определения; она же генерирует определение d_7 , которое уничтожается инструкцией d_4 ; следовательно, во множество kill родительского по отношению к if узла входят только d_1 и d_2 .

Теперь мы можем вычислить in и out , начиная от вершины дерева разбора. Мы полагаем, что в корневом узле дерева множество in пустое. Таким образом, out в левом дочернем по отношению к корню узле равно gen , т.е. 111 0000. Это также и значение множества in в узле do. Из уравнений потока данных, связанных с продукцией do на рис. 10.21, множество in для инструкции внутри цикла do получается как объединение множества in (111 0000) в узле do и множества gen (000 1111) инструкции. Операция объединения дает нам 111 1111, так что все определения могут достичь начала тела цикла. Однако в точке непосредственно перед определением d_5 множество in равно 011 1110, поскольку определение d_4 уничтожает d_1 и d_7 . Остаток вычислений in и out предлагается читателю в качестве упражнения. □

Локальные достигающие определения

Затраты пространства для информации о потоке данных можно заменить на затраты времени, сохраняя информацию только для определенных точек и при необходимости вычисляя информацию в промежуточных точках заново. Обычно в процессе анализа по-

тока данных в качестве единиц принимаются базовые блоки; при этом все внимание ограничивается только точками начала блоков. Поскольку точек обычно гораздо больше, чем блоков, ограничение наших усилий блоками дает немалую экономию. При необходимости достигающие определения для всех точек блока могут быть вычислены на основании определений в начале блока.

Для большей ясности рассмотрим последовательность присвоений $S_1 ; S_2 ; \dots ; S_n$ в базовом блоке B . Обозначим точку в начале блока как p_0 , точку между присвоениями S_i и S_{i+1} — как p_i , а точку в конце блока — как p_n . Определения, достигающие точки p_j , могут быть получены из $in[B]$ рассмотрением инструкций $S_1 ; S_2 ; \dots ; S_j$, начиная с S_1 и применяя уравнения потока данных на рис. 10.21 для последовательностей инструкций. Пусть изначально $D = in[B]$. При рассмотрении S_i , мы удаляем из D определения, уничтожаемые S_i и добавляем туда определения, генерируемые им. В конце рассмотрения D состоит из определений, достигающих p_j .

Цепочки определений использований

Зачастую информацию о достигающем определении удобно хранить в виде “цепочек определений использований” (use-definition chain), или, для краткости, ои-цепочек, которые для каждого использования переменной представляют собой списки всех определений, достигающих его. Если перед использованием переменной a в блоке B нет однозначных определений a , то ои-цепочка для этого использования a является множеством определений a в $in[B]$. Если в B есть однозначные определения a , предшествующие рассматриваемому нами использованию, то в ои-цепочку входит только последнее такое определение a и не входит $in[B]$. Кроме того, если имеются неоднозначные определения a , то те из них, для которых между ними и использованием a нет однозначных определений a , попадают в ои-цепочку данного использования a .

Порядок вычислений

Технологии, обеспечивающие экономию памяти при вычислении атрибутов, рассмотренные в главе 5, “Синтаксически управляемая трансляция”, применимы и для вычисления информации о потоке данных с использованием спецификаций, подобных приведенным на рис. 10.21. Единственное ограничение на порядок вычислений множеств gen , $kill$, in и out связано с зависимостями между этими множествами. Выбрав порядок вычислений, мы можем освободить память, выделенную для множества, после его использования.

Уравнения потока данных в этом разделе отличаются от семантических правил для атрибутов из главы 5, “Синтаксически управляемая трансляция”, в одном: циклические зависимости между атрибутами были запрещены, в то время как уравнения потока данных, как мы видели, могут их иметь — например, в (10.8) $in[S_1]$ и $out[S_1]$ зависят друг от друга. Для задачи достигающих определений уравнения потока данных могут быть переписаны в целях устранения цикличности (сравните, например, нециклические уравнения на рис. 10.21 с (10.8)). После этого для эффективного решения уравнений потока данных могут использоваться технологии, описанные в главе 5, “Синтаксически управляемая трансляция”.

Общий поток управления

Анализ потока данных должен принимать во внимание все пути управления. Если пути управления очевидным образом определяются синтаксисом, то уравнения потока данных могут быть записаны и решены синтаксически управляемым способом, как это сделано в данном разделе. Если же программы могут содержать безусловные переходы (и даже более “дисциплинированные” инструкции `break` и `continue`), такой подход должен быть модифицирован с тем, чтобы учесть реальные пути управления.

Для этого можно воспользоваться различными способами. Итеративный метод из следующего раздела работает для произвольных графов потока. Поскольку при наличии инструкций `break` и `continue` получаются приводимые графы потока, такие конструкции могут быть обработаны с помощью метода, основанного на интервалах (который рассматривается в разделе 10.10).

Однако даже при наличии в программе инструкций `break` и `continue` не следует отказываться от синтаксически управляемого подхода. Перед тем как перейти к следующему разделу главы, мы рассмотрим пример, показывающий, каким образом можно приспособить синтаксически управляемый подход к инструкции `break` (эта идея будет развита в разделе 10.10).

Пример 10.14

Инструкция `break` в цикле `do-while` на рис. 10.24 эквивалентна переходу в конец цикла. Как в этом случае мы должны определить `gen` для следующего кода?

```
if e3 then a := u2
else begin i := u3; break end
```

Мы определим `gen` как $\{d_6\}$, где d_6 представляет собой определение `a := u2`, так как это единственное определение, генерируемое на пути управления от начальной до конечной точки рассматриваемой инструкции. Определение d_7 , т.е. `i := u3`, будет принято во внимание при рассмотрении цикла `do-while` в целом.

```
/* d1 */      i := m - 1;
/* d2 */      j := n;
/* d3 */      a := u1;
                do
/* d4 */      i := i + 1;
/* d5 */      j := j - 1;
                if e3 then
/* d6 */      a := u2
                else begin
/* d7 */      i := u3;
                break
            end
        while e2
```

Рис. 10.24. Программа, содержащая инструкцию `break`

Имеется один программный способ игнорировать переход, вызываемый `break` при работе с инструкциями внутри тела цикла. Мы считаем, что для инструкции `break` множества `gen` и `kill` представляют собой соответственно пустое множество \emptyset и всеобщее множество всех определений U , как показано на рис. 10.25. Остальные множества

gen и *kill* на том же рисунке определяются с использованием уравнений потока данных на рис. 10.21 (на рисунке множества *gen* изображены над множествами *kill*). S_1 и S_2 представляют последовательности присвоений. Множества *gen* и *kill* для узла do пока что остаются не определены.

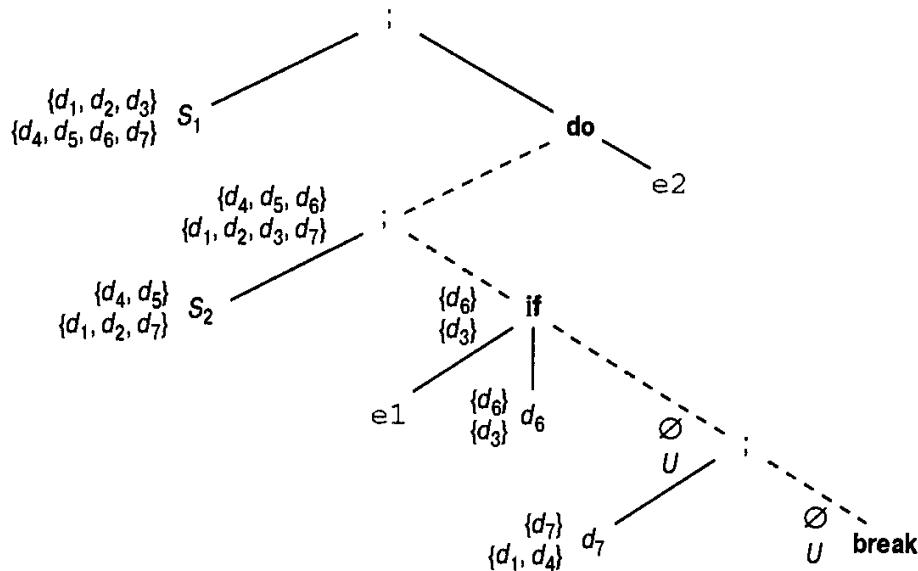


Рис. 10.25. Влияние инструкции break на множества gen и kill

Поскольку конечная точка любой последовательности инструкций, заканчивающейся инструкцией `break`, не может быть достигнута, можно смело считать, что для этой последовательности $gen = \emptyset$ и $kill = U$; в результате мы все равно получим консервативную оценку in и out . Аналогично конечной точки инструкции `if` можно достичь только по пути `then`, поэтому на рис. 10.25 множества gen и $kill$ в узле `if` те же, что и у его второго дочернего узла.

При определении множеств *gen* и *kill* в узле **do** следует принять во внимание все пути от начала до конца конструкции **do**, поскольку на них влияет наличие инструкции **break**. Вычислим два изначально пустых множества *G* и *K* при проходе по пути от узла **do** к узлу **break** (на рис. 10.25 этот путь указан пунктирной линией). Интуитивно множества *G* и *K* представляют определения, генерируемые и уничтожаемые при проходе управления к инструкции **break** от начала тела цикла. После вычисления *G* и *K* множество *gen* для цикла можно найти как объединение *G* и множества *gen* тела цикла, поскольку управление может достичь конца цикла либо посредством инструкции **break**, либо проходом по всему телу цикла. По тем же причинам множество *kill* цикла находится как пересечение *K* и множества *kill* тела цикла.

Непосредственно перед узлом `if` мы имеем $G = \text{gen}[S_2] = \{d_4, d_5\}$ и $K = \text{kill}[S_2] = \{d_1, d_2, d_7\}$. В узле `if` нас интересует случай перехода к инструкции `break`, так что ветвь `then` не влияет на множества G и K . Следующий узел на пунктирном пути соответствует последовательности инструкций, и мы вычисляем здесь новые значения G и K . Обозначая через S_3 инструкцию, представленную левым дочерним узлом с отметкой d_7 , мы получаем

$$\begin{aligned} G &:= \text{gen}[S_3] \cup (G - \text{kill}[S_3]) \\ K &:= \text{kill}[S_3] \cup (K - \text{gen}[S_2]) \end{aligned}$$

Следовательно, значениями G и K при достижении инструкции `break` являются соответственно множества $\{d_5, d_7\}$ и $\{d_1, d_2, d_4\}$. \square

10.6. Итеративное решение уравнений потока данных

Метод, изложенный в предыдущем разделе, прост в применении и эффективен, но для языков программирования типа Fortran или Pascal, допускающих произвольные графы потоков, этот метод недостаточно общ. В разделе 10.10 рассматривается “анализ интервалов”, представляющий собой способ добиться преимуществ синтаксически управляемого подхода к анализу потока данных для графов потоков общего вида ценой существенного концептуального усложнения.

В этом разделе мы рассмотрим другой важный подход к решению задач потока данных. Вместо попыток использовать дерево разбора для управления вычислением множеств *in* и *out* мы сначала строим граф потока, а затем вычисляем *in* и *out*, одновременно для всех узлов. При рассмотрении этого метода мы получим возможность познакомить читателя с рядом различных задач анализа потока данных, показать некоторые из их применений на практике и указать их отличия от других подобных задач.

Уравнения для многих задач потока данных схожи по форме в том, что информация в них “генерируется” и “уничтожается”. Однако имеется два принципиальных различия таких уравнений.

1. Уравнения из предыдущего раздела для достигающих определений представляют собой *прямые* (forward) уравнения в том смысле, что множества *out* вычисляются на основе множеств *in*. Мы познакомимся также с *обратными* (backward) задачами, в которых множества *in* вычисляются на основе множеств *out*.
2. Когда в блок *B* входит более одной дуги, определения, достигающие начала блока *B*, представляют собой объединение определений, достигающих начала блока по каждой из дуг. Таким образом, мы говорим о том, что объединение является *оператором слияния* (confluence operator). Мы встретимся также с задачами типа глобально доступных выражений, в которых оператором слияния является пересечение, поскольку выражение доступно в начале блока *B* только в том случае, когда оно доступно в конце каждого из предшественников *B*. В разделе 10.11 мы рассмотрим и другие примеры операторов слияния.

В этом разделе нами будут рассмотрены прямые и обратные уравнения с объединениями и пересечениями в качестве оператора слияния.

Итеративный алгоритм для достигающих определений

Для каждого базового блока *B* мы можем определить *gen[B]*, *out[B]*, *kill[B]* и *in[B]* как и в предыдущем разделе с примечанием, что каждый блок *B* может рассматриваться как составная инструкция из одного или нескольких присвоений. Полагая, что для каждого блока вычислены *gen* и *kill*, мы можем создать две группы уравнений (10.9), связывающие *in* и *out*. Первая группа уравнений следует из наблюдения, что *in[B]*, представляет собой объединение определений, попадающих в *B* из всех предшественников. Вторая группа представляет собой частный случай общего закона (10.5), который должен выполняться для всех инструкций. Рассмотрим эти группы.

$$\begin{aligned} \textit{in}[B] &= \bigcup_{\substack{P - \text{пред-} \\ \text{шественник } B}} \textit{out}[P] \\ \textit{out}[B] &= \textit{gen}[B] \cup (\textit{in}[B] - \textit{kill}[B]) \end{aligned} \tag{10.9}$$

Если граф потока имеет n базовых блоков, (10.9) дает $2n$ уравнений. Эти $2n$ уравнений могут быть решены, если рассматривать их как рекуррентные соотношения для вычисления множеств in и out , т.е. так же, как были решены уравнения (10.6) и (10.5) для цикла **do-while** в предыдущем разделе. Там мы начинали с пустого множества определений в качестве начального приближения для всех множеств out . Здесь же мы начнем с пустых множеств in , поскольку, если множества out пустые, из первого уравнения (10.9) очевидно, что множества in также пустые. В то время как для решения уравнений (10.6) и (10.7) требовалась только одна итерация, в нашем более сложном случае мы не можем заранее указать требуемое количество итераций.

Алгоритм 10.2. Достигающие определения

Вход. Граф потока, в котором для каждого блока B вычислены множества $kill[B]$ и $gen[B]$.

Выход. Множества $in[B]$ и $out[B]$ для каждого блока B .

Метод. Мы используем итеративный подход с начальной оценкой $in[B] = \emptyset$ для всех B , сходящийся к требуемым значениям in и out . Поскольку итерации должны продолжаться до тех пор, пока не сойдутся множества in (а следовательно, и out), мы используем логическую переменную *change* в качестве индикатора, были ли внесены при данном проходе изменения в какое-либо из множеств in . Набросок алгоритма приведен на рис. 10.26.

```

/* Инициализируем out в предположении, что
   in[B]=∅ для всех B */
(1) for Каждый блок B do out[B] := gen[B];
(2) change := true; /* Переменная цикла while */
(3) while change do begin
(4)     change := false;
(5)     for Каждый блок B do begin
(6)         in[B]:= ∪P-пред-  
шественник B out[P]
(7)         oldout := out[B];
(8)         out[B] := gen[B] ∪ (in[B] - kill[B])
(9)         if out[B] ≠ oldout then change := true
    end
end

```

Рис. 10.26. Алгоритм вычисления in и out

Интуитивно алгоритм 10.2 распространяет определения до тех пор, пока это возможно без их уничтожения, моделируя все возможные пути выполнения программы. В библиографических примечаниях имеются ссылки на формальные доказательства корректности решения этой и других задач анализа потока данных.

Мы можем видеть, что в конечном счете этот алгоритм завершит свою работу, поскольку для любого B множество $out[B]$ не уменьшается в размере; если определение попало в это множество, оно останется там навсегда (доказательство этого факта по индукции оставлено читателю в качестве упражнения). Поскольку множество всех определений конечно, рано или поздно должен произойти проход цикла, при котором для каждого B в строке (9) будет выполнено условие $oldout = out[B]$; при этом переменная *change* останется равной *false* и алгоритм завершит свою работу. Поскольку если out не-

изменно, то и in будет неизменно при следующем проходе (и наоборот), мы можем завершать работу алгоритма по условию неизменности множества out .

Можно показать, что верхняя граница количества итераций в цикле равна количеству узлов в графе потока. Интуитивно понятна причина этого — если определение достигает некоторой точки, то оно может сделать это по пути без циклов, а количество узлов в графе потока и есть верхняя граница количества узлов в пути без циклов.

В действительности имеются эмпирические данные, что при правильном упорядочении блоков в цикле в строке (5) среднее количество итераций для реальных программ не превышает 5 (см. раздел 10.10). Поскольку множества могут быть представлены битовыми векторами, а операции над этими множествами могут быть реализованы побитовыми логическими операциями, на практике алгоритм 10.2 удивительно эффективен.

Пример 10.15

Граф потока на рис. 10.27 построен на основании программы на рис. 10.22 в предыдущем разделе. Мы применим алгоритм 10.2 к этому графу с тем, чтобы сравнить подходы, предложенные в текущем и предыдущем разделах.

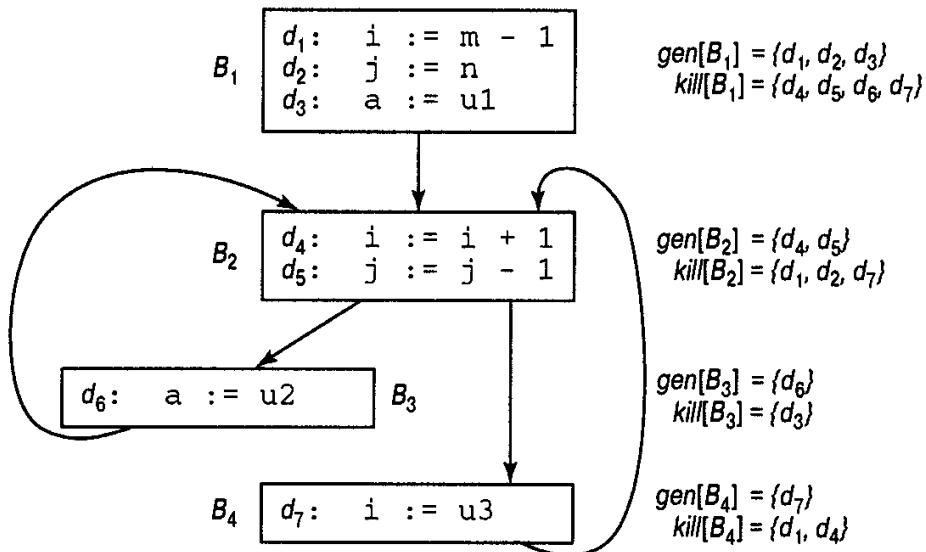


Рис. 10.27. Граф потока для иллюстрации достигающих определений

На рис. 10.27 нас интересуют только определения d_1, d_2, \dots, d_7 , которые определяют i , j и a . Как и в последнем разделе, мы представим множества определений битовыми векторами, где i -й слева бит представляет определение d_i .

Цикл в строке (1) на рис. 10.26 инициализирует значения $out[B] = gen[B]$ для каждого B (эти начальные значения $out[B]$ показаны в таблице на рис. 10.28). Начальные значения каждого $in[B] = \emptyset$ не вычисляются и не используются, но для полноты изложения также приведены в таблице. Предположим, что цикл в строке (5) выполняется с $B = B_1, B_2, B_3, B_4$ (именно в таком порядке). Для $B = B_1$ предшественника нет, так что множество $in[B_1]$ остается пустым, представленным вектором 000 0000; в результате $out[B_1]$ остается равным $gen[B_1]$. Это значение не отличается от $oldout$, вычисленного в строке (7), так что мы пока не изменяем значения переменной $change$.

Затем мы рассматриваем $B = B_2$ и вычисляем

$$\begin{aligned}
 in[B_2] &= out[B_1] \cup out[B_3] \cup out[B_4] \\
 &= 1110000 + 0000010 + 0000001 = 1110011 \\
 out[B_2] &= gen[B_2] \cup (in[B_2] - kill[B_2]) \\
 &= 0001100 + (1110011 - 1100001) = 0011110
 \end{aligned}$$

Эти вычисления резюмированы в таблице на рис. 10.28. В конце первого прохода $out[B_4]=001\ 0111$; это отражает тот факт, что сгенерировано d_7 , а d_3, d_5 и d_6 достигли B_4 и не уничтожены в этом блоке. При втором проходе ни в одном из множеств out нет изменений, а следовательно, алгоритм завершает свою работу. \square

БЛОК В	ИЗНАЧАЛЬНО		ПРОХОД 1		ПРОХОД 2	
	$in[B]$	$out[B]$	$in[B]$	$out[B]$	$in[B]$	$out[B]$
B_1	000 0000	111 0000	000 0000	111 0000	000 0000	111 0000
B_2	000 0000	000 1100	111 0011	001 1110	111 1111	001 1110
B_3	000 0000	000 0010	001 1110	000 1110	001 1110	000 1110
B_4	000 0000	000 0001	001 1110	001 0111	001 1110	001 0111

Рис. 10.28. Вычисление in и out

Доступные выражения

Выражение $x+y$ доступно в точке p , если любой путь (не обязательно без циклов) от начального узла к p вычисляет $x+y$ и после каждого такого вычисления и до достижения p нет последующих присвоений переменным x и y . Когда речь идет о доступных выражениях, мы говорим, что блок *уничтожает* выражение $x+y$, если он присваивает (или может присваивать) x и y и после этого не вычисляет $x+y$ заново. Блок *генерирует* выражение $x+y$, если он вычисляет $x+y$ и не выполняет последующих переопределений x и y .

Заметим, что понятия “уничтожения” и “генерации” доступного выражения не те же, что в случае достигающих определений. Несмотря на это, уничтожение и генерация доступных выражений подчиняются тем же законам, что и уничтожение, и генерация для достигающих определений. Мы можем вычислить их в точности так же, как и в разделе 10.5, модифицировав правила для простой инструкции присвоения на рис. 10.21а.

Основное использование информации о доступных выражениях — поиск общих подвыражений. Например, на рис. 10.29 выражение $4*i$ в блоке B_3 будет общим подвыражением, если $4*i$ доступно во входной точке блока B_3 . Это выражение будет доступно, если i не будет присвоено новое значение в блоке B_2 или если $4*i$ будет заново вычислено после такого присвоения в блоке B_2 , как показано на рис. 10.29б.

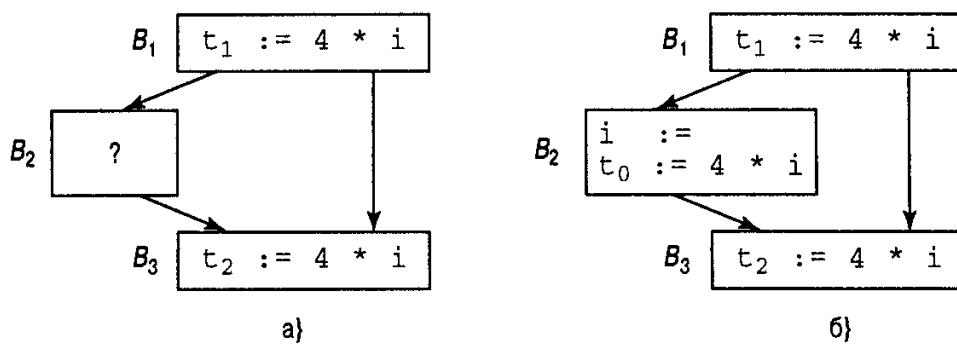


Рис. 10.29. Потенциальные общие подвыражения, пересекающие границы блоков

Мы можем легко вычислить множество генерируемых выражений для каждой точки блока, проходя от начала до конца блока. Предположим, что в точке, предшествующей блоку, доступных выражений нет. Если в точке p доступно множество выражений A , а q представляет собой точку после p с инструкцией $x := y + z$ между ними, то мы образуем множество доступных в q выражений следующим образом.

1. Добавляем к A выражение $y + z$.
2. Удаляем из A все выражения, включающие x .

Заметим, что описанные действия должны выполняться в указанном порядке, так как x может совпадать с y или z . После того как мы достигнем конца блока, A будет представлять собой множество генерируемых блоком выражений. Множество уничтоженных выражений представляет собой множество всех выражений, скажем $y + z$, таких, что y или z определяются в блоке и при этом $y + z$ блоком не генерируется.

Пример 10.16

Рассмотрим четыре инструкции, показанные на рис. 10.30. После первой доступно $b + c$, после второй становится доступным выражение $a - d$, но $b + c$ более недоступно, поскольку при этом переопределяется b . Третья инструкция не делает $b + c$ доступным, поскольку в ней же переопределяется c . После последней инструкции в связи с тем, что она изменяет d , становится недоступным и $a - d$. Итак, здесь не генерируется ни одно доступное выражение, и при этом уничтожаются все выражения, включающие a , b , c или d . \square

ИНСТРУКЦИЯ	ДОСТУПНЫЕ ВЫРАЖЕНИЯ
.....	Нет
$a := b + c$
$b := a - d$	Только $b + c$
$c := b + c$
$d := a - d$	Только $a - d$
.....	Нет

Рис. 10.30. Вычисление доступных выражений

Мы можем найти доступные выражения методом, напоминающим вычисление достижимых определений. Предположим, что U — “универсальное” множество всех выражений, появляющихся в правой части одной или нескольких инструкций программы. Пусть для каждого блока B множество $in[B]$ содержит выражения из U , доступные в точке непосредственно перед началом блока B , а $out[B]$ — такое же множество для точки, следующей за концом блока B . Определим $e_gen[B]$ как множество выражений, генерируемых B , а $e_kill[B]$ — как множество выражений из U , уничтожаемых в B . Заметим, что множества in , out , e_gen и e_kill могут быть представлены в виде битовых векторов. Неизвестные множества in и out связаны друг с другом и с известными e_gen и e_kill следующими соотношениями.

$$\begin{aligned}
 out[B] &= e_gen[B] \cup (in[B] - e_kill[B]) \\
 in[B] &= \bigcap_{\substack{P - \text{пред-} \\ \text{шественник } B}} out[P] \quad \text{для всех } B, \text{ кроме начального} \\
 in[B_1] &= \emptyset \quad \text{где } B_1 \text{ --- начальный блок}
 \end{aligned} \tag{10.10}$$

Уравнения (10.10) выглядят почти идентичными уравнениям (10.9) для достигающих определений. Первое отличие состоит в том, что особый частный случай представляет собой in для начального узла. Это обосновывается тем, что только что начавшейся программе недоступны никакие выражения, несмотря на то что некоторые выражения могут оказаться доступны по всем путям, входящим в начальный блок из других мест в программе. Если мы не примем условие $in[B_1] = \emptyset$, то можем затем прийти к ошибочному выводу о том, что некоторые выражения доступны до начала работы программы.

Второе, более важное отличие состоит в том, что в данном случае оператором слияния служит пересечение, а не объединение множеств. Пересечение множеств используется потому, что выражение доступно в начале блока только в том случае, когда оно доступно в конце всех его предшественников (в отличие от определения, которое достигает начала блока, если достигает конца хотя бы одного его предшественника).

Использование оператора \cap вместо \cup делает поведение уравнений (10.10) отличающимся от (10.9). Пока не получено единственное решение, (10.9) предоставляет наименьшее решение, соответствующее определению достижимости, и мы получаем это решение, начиная с предположения о недостижимости чего бы то ни было в какой угодно точке, а затем “наращиваем” его. При этом подходе мы никогда не предполагаем, что определение d может достичь точки p , до тех пор, пока не будет найден реальный путь распространения d до p . В отличие от этого, для уравнений (10.10) мы хотим найти наибольшее возможное решение, а потому начинаем с максимального приближения и идем по пути его уменьшения.

Может показаться неочевидным, что, начав с предположения “доступно все (т.е. множество U) и везде” и удаляя только те выражения, для которых мы находим пути, по которым они становятся недоступны, мы получим множество истинно доступных выражений. В случае доступных выражений консервативным является получение подмножества точного множества доступных выражений, и это именно то, что мы делаем. Аргументом в пользу консервативности подмножеств является наше предполагаемое использование информации для замены вычисления доступного выражения предварительно вычисленным значением (см. алгоритм 10.5 из следующего раздела), и отсутствие информации о доступности выражения только предотвратит возможное изменение кода.

Пример 10.7

Для иллюстрации влияния начальной аппроксимации $in[B_2]$ на $out[B_2]$ мы ограничимся одним блоком B_2 на рис. 10.31. Пусть G и K обозначают соответственно $gen[B_2]$ и $kill[B_2]$. Уравнения потока данных для блока B_2 следующие:

$$\begin{aligned}
 in[B_2] &= out[B_1] \cap out[B_2] \\
 out[B_2] &= G \cup (in[B_2] - K)
 \end{aligned}$$

На рис. 10.31 эти уравнения записаны в виде рекуррентных соотношений, где I^j и O^j означают j -е приближение соответственно $in[B_2]$ и $out[B_2]$. На рисунке также показано, что начиная с $I^0 = \emptyset$, мы получим $O^1 = O^2 = G$, в то время как начиная с $I^0 = U$, мы получим большее значение O^2 . Так получилось, что в каждом случае $out[B_2] = O^2$, так как итерации сходятся в показанных точках.

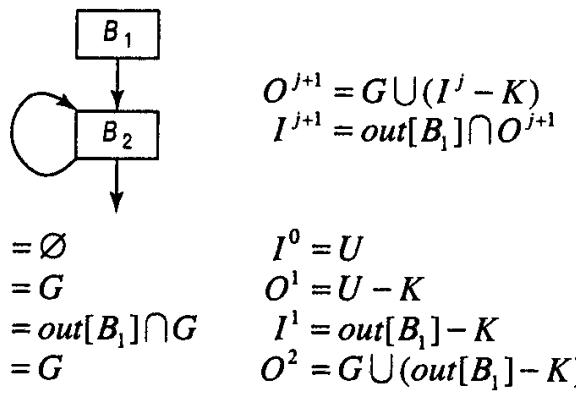


Рис. 10.31. Влияние начального приближения множества in на решение уравнений потока

Интуитивно решение, полученное с начальным приближением $I^0 = U$ с использованием $\text{out}[B_2] = G \cup (\text{out}[B_1] - K)$, более желательно, так как оно точно отражает тот факт, что выражения в $\text{out}[B_1]$, не уничтоженные в B_2 , доступны в конце B_2 , так же как и выражения, сгенерированные в B_2 . \square

Алгоритм 10.3. Доступные выражения

Вход. Граф потока G с $e_kill[B]$ и $e_gen[B]$, вычисленными для каждого блока B . Начальным блоком является блок B_1 .

Выход. Множество $in[B]$ для каждого блока B .

Метод. Выполнить алгоритм, приведенный на рис. 10.32. Пояснения выполняемых шагов аналогичны приведенным на рис. 10.26. \square

```

in[B1] := ∅
out[B1] := e_gen[B1]; /* Для начального узла B1 in и */
/* out никогда не изменяются */
for B ≠ B1 do      /* Большое начальное приближение */
    out[B] := U - e_kill[B];
change := true;
while change do begin
    change := false;
    for B ≠ B1 do begin
        in[B] := ⋂P-предшественник B out[P];
        oldout := out[B];
        out[B] := e_gen[B] ∪ (in[B] - e_kill[B]);
        if out[B] ≠ oldout then change := true
    end
end

```

Рис. 10.32. Вычисление доступных выражений

Анализ активных переменных

Большое количество улучшающих преобразований зависит от информации, вычисляемой в направлении, обратном потоку управления программы. Некоторые из них будут рассмотрены сейчас. Анализ активных переменных состоит в том, что для данной переменной x и точки p мы хотим узнать, может ли значение x в точке p быть использовано

вдоль некоторого, начинающегося в p пути графа потока. Если да, то мы говорим, что переменная x *активная*, или *живая* (live) в точке p ; в противном случае переменная x в точке p *неактивна*, или *мертва* (dead).

Как мы видели в разделе 9.7, информация об активных переменных играет большую роль в процессе генерации объектного кода. После того как значение вычислено и находится в регистре и, предположительно, будет использовано в блоке, нет необходимости сохранять это значение, если при выходе из блока оно неактивно. Кроме того, в ситуации, когда заполнены все регистры и их не хватает для вычислений, мы должны постараться использовать регистры с неактивными значениями, поскольку сохранять их не обязательно.

Определим $in[B]$ как множество переменных, активных в точке непосредственно перед блоком B , а $out[B]$ — как такое же множество в точке, непосредственно следующей за блоком. Обозначим через $def[B]$ множество переменных, которым в блоке B значения присваиваются до их использования, а через $use[B]$ — множество переменных, значения которых могут использоваться в B до их определения. Тогда уравнения, связывающие def и use с неизвестными in и out , выглядят следующим образом.

$$\begin{aligned} in[B] &= use[B] \cup (out[B] - def[B]) \\ out[B] &= \bigcup_{S - \text{пр-} \atop \text{емник } B} in[S] \end{aligned} \quad (10.11)$$

Первая группа уравнений свидетельствует, что переменная активна при входе в блок, если она либо используется до переопределения в блоке, либо она активна на выходе из блока и в нем не переопределена. Вторая группа гласит, что переменная активна на выходе из блока тогда и только тогда, когда она активна при входе хотя бы в один преемник блока B .

Следует отметить связь между (10.11) и уравнениями достигающих определений (10.9). Здесь in и out поменялись ролями, а use и def заменили соответственно gen и $kill$. Как и в случае (10.9), решение не обязательно единственное, и нас интересует наименьшее решение. Алгоритм, используемый для поиска минимального решения, по сути, представляет собой обратную версию алгоритма 10.2. Поскольку механизм обнаружения изменений в любом из множеств in аналогичен обнаружению изменений во множествах out в алгоритмах 10.2 и 10.3, мы опустим детали этого обнаружения для проверки условия завершения алгоритма.

Алгоритм 10.4. Анализ активных переменных

Вход. Граф потока с вычисленными для каждого блока множествами def и use .

Выход. Множество $out[B]$ переменных, активных при выходе из каждого блока B графа потока.

Метод. Выполняем программу, приведенную на рис. 10.33. □

```

for Каждый блок  $B$  do  $in[B] := \emptyset$ 
while Вносятся изменения хотя бы в один из  $in$  do
    for Каждый блок  $B$  do begin
         $out[B] := \bigcup_{S - \text{пр-} \atop \text{емник } B} in[S]$ 
         $in[B] := use[B] \cup (out[B] - def[B])$ 
    end

```

Рис. 10.33. Вычисление активных переменных

Цепочки использований определений

Вычисления, осуществляемые практически так же, как и анализ активных переменных, являются *цепочками использований определений* (definition-use chaining), или ио-цепочками. Мы говорим, что переменная *используется* в инструкции s , если может потребоваться ее r -значение. Например, в каждой из инструкций $a := b+c$ и $a[b] := c$ используются b и c (но не a). Задача ио-цепочек заключается в вычислении для данной точки p множества использований s переменной x , таких, что имеется путь от p до s , который не переопределяет x .

Как и в случае активных переменных, если мы можем вычислить $out[B]$, множество использований, достижимых от конца блока B , то мы можем вычислить и определения, достижимые из любой точки p в блоке B , сканированием части блока B , следующей за p . В частности, если в блоке имеется определение переменной x , мы можем вычислить для такого определения *ио-цепочку*, т.е. список всех возможных использований этого определения. Метод вычислений аналогичен методу, рассмотренному в разделе 10.5 для вычисления ои-цепочек, и его разработку мы оставляем читателю в качестве упражнения.

Уравнения для вычисления информации об ио-цепочках выглядят совершенно так же, как и (10.11), но с заменой def и use . Вместо $use[B]$ мы берем множество *импортирующих использований* (upwards exposed uses) B , т.е. множество пар (s, x) , таких, что s — инструкция B , использующая переменную x , и при этом предшествующих определений x в B нет. Вместо $def[B]$ мы берем множество пар (s, x) , таких, что s — инструкция, использующая x , и при этом $s \notin B$ и B содержит определение x . Эти уравнения решаются с помощью очевидного аналога алгоритма 10.4, и обсуждение вопроса мы на этом прекращаем.

10.7. Преобразования, улучшающие код

Алгоритмы преобразований, совершенствующих код, с которыми мы познакомились в разделе 10.2, основываются на информации о потоках данных. В последних двух разделах мы увидели, каким образом можно собрать эту информацию, а сейчас мы рассмотрим устранение общих подвыражений, распространение копий, перенос инварианта цикла за его пределы и устранение переменных индукции. Во многих языках значительное снижение времени работы программы достигается путем совершенствования кода в циклах. Хотя при реализации таких преобразований в компиляторе некоторые из них могут быть выполнены одновременно, каждая из идей, лежащих в основе указанных преобразований, будет представлена отдельно.

Упор в этом разделе будет сделан на глобальных преобразованиях, использующих информацию о программе в целом. Как мы видели в двух предыдущих разделах, глобальный анализ потока данных обычно не рассматривает точки внутри базовых блоков, и, таким образом, глобальные преобразования не могут заменить собой локальные, и выполняться при оптимизации должны оба вида преобразований. Например, когда мы выполняем глобальное устранение общих подвыражений, должны сосредоточиться только на том, генерируется ли выражение блоком, а не на том, вычисляется ли оно в блоке неоднократно.

Устранение глобальных общих подвыражений

Обсуждавшиеся в предыдущем разделе доступные выражения позволяют нам определить, является ли выражение в точке p графа потока общим подвыражением. Приведенный далее алгоритм формализует интуитивные идеи устранения общих подвыражений, представленные в разделе 10.2.

Алгоритм 10.5. Устранение глобальных общих подвыражений

Вход. Граф потока с информацией о доступных выражениях.

Выход. Преобразованный граф потока.

Метод. Для каждой инструкции s вида $x := y + z$ ⁸, такой, что $y + z$ доступно в начале блока, которому принадлежит s , и ни y , ни z не определены в блоке до инструкции s , выполняем следующее.

1. Для поиска вычислений $y + z$, достигающих блока, которому принадлежит s , мы следуем по дугам графа потока в обратном направлении от рассматриваемого блока. Однако мы останавливаемся на блоках, вычисляющих $y + z$. Последнее вычисление $y + z$ в каждом таком блоке и есть вычисление $y + z$, достигающее s .
2. Создаем новую переменную u .
3. Заменяем каждое выражение $w := y + z$, найденное в (1), инструкциями

$$u := y + z$$

$$w := u$$

4. Заменяем s инструкцией $x := u$. □

Далее следуют некоторые замечания о приведенном алгоритме.

1. Поиск на шаге (1) вычисления $y + z$, достигающего инструкции s , может также быть сформулирован в виде задачи анализа потока данных. Однако не имеет смысла решать ее для всех выражений $y + z$ и всех инструкций в блоке, поскольку при этом накапливается слишком много ненужной информации. Вместо этого мы выполняем поиск по графу потока только для необходимых инструкций и выражений.
2. Не все преобразования, выполняемые алгоритмом 10.5, представляют собой улучшение кода. Вероятно, вы захотите ограничиться преобразованиями кода только для случаев, когда число различных достигающих s вычислений, найденных на шаге (1), равно 1. Однако распространение копий, которое мы рассмотрим немного позже, зачастую дает выигрыш даже в случае нескольких вычислений $y + z$, достигающих s .
3. Алгоритм 10.5 не заметит, что в следующих фрагментах $a * z$ и $c * z$ должны иметь одно и то же значение:

$$\begin{aligned} a &:= x + y \\ b &:= a * z \end{aligned}$$

$$\begin{aligned} c &:= x + y \\ d &:= c * z \end{aligned}$$

поскольку в рассмотренном нами простом подходе к вопросу об общих подвыражениях рассматриваются только символьные представления выражений, а не вычисляемые ими значения. В [249] представлен однопроходный метод поиска таких тождеств; основные идеи такого метода будут рассмотрены нами в разделе 10.11. Однако многопроходное применение алгоритма 10.5 до тех пор, пока не будут найдены все общие подвыражения, позволит справиться с указанным выше случаем. Если a и c представляют собой временные переменные, не используемые вне блока, в котором они появляются, то общее подвыражение $(x+y) * z$ может быть найдено при специальном рассмотрении временных переменных, как показано в следующем примере.

⁸ Вспомним, что мы рассматриваем $+$ как обобщенный оператор.

Пример 10.18

Предположим, что в графе потока, представленном на рис. 10.34 a , отсутствуют присвоения массиву. Тогда мы можем утверждать, что $a[t_2]$ и $a[t_6]$ представляют собой общие подвыражения. Задача состоит в удалении этих общих подвыражений.

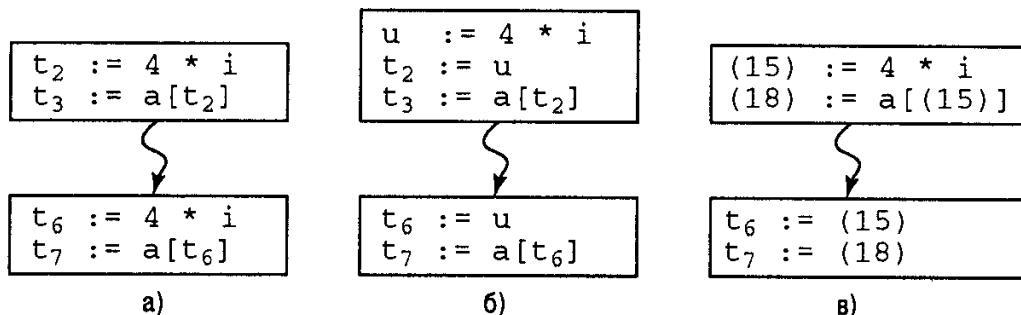


Рис. 10.34. Устранение общего подвыражения $4 * i$

Общее подвыражение $4 * i$ на рис. 10.34 a устранено на рис. 10.34 b . Один из способов определить, что $a[t_2]$ и $a[t_6]$ представляют собой общие подвыражения, — это заменить t_2 и t_6 переменной u с использованием технологии распространения копий (об этом мы поговорим позже). При этом оба выражения принимают вид $a[u]$, к которым может быть повторно применен алгоритм 10.5. Заметим, что одна и та же новая переменная u имеется в обоих блоках на рис. 10.34 b , так что для преобразования $a[t_2]$ и $a[t_6]$ в $a[u]$ достаточно локального распространения копий.

Есть еще один путь, принимающий во внимание тот факт, что временные переменные вносятся компилятором и используются только в пределах своих блоков. Для уяснения того, что различные временные переменные могут представлять одно и то же выражение, мы должны пристальнее рассмотреть способ представления выражений в процессе вычисления доступных выражений. Рекомендуемый способ представления множеств выражений состоит в том, чтобы назначить каждому из них номер и использовать битовый вектор, в котором i -й бит представляет выражение номер i . Для обработки временных переменных некоторым специальным способом в процессе нумерации выражений можно воспользоваться технологиями из раздела 5.2.

Рассматривая вопрос более подробно, предположим, что $4 * i$ имеет значение номер 15. Выражения $a[t_2]$ и $a[t_6]$ получат тот же номер значения, если мы воспользуемся номером значения, а не именами временных переменных t_2 и t_6 . Предположим, что в результате мы получаем номер значения 18. В таком случае бит 18 в процессе анализа потока данных представляет как $a[t_2]$, так и $a[t_6]$, и мы можем установить, что выражение $a[t_6]$ доступно и может быть устранино. Полученный в результате код показан на рис. 10.34 c . Здесь (15) и (18) использованы для представления временных переменных, соответствующих этим номерам значений. В действительности t_6 бесполезно и было бы устранино в процессе локального анализа активных переменных. Кроме того, t_7 , будучи временной переменной, не вычислялось бы — использования t_7 были бы заменены использованием (18). \square

Распространение копирований

Только что представленный алгоритм 10.5, как и другие различные алгоритмы типа устранения переменной индукции, рассматриваемые позже в этой главе, работают с инструкциями копирования вида $x := y$. Инструкции копирования могут быть созданы

непосредственно генератором промежуточного кода, хотя большинство из них работают со временными переменными, локальными по отношению к одному блоку, и могут быть устранины при рассмотренном в разделе 9.8 построении дага. Иногда можно устранить инструкцию копирования $s: x := y$, если мы определим все места, где используется данное определение x . Затем мы можем подставить y вместо переменной x во всех этих местах, при условии, что для каждого такого использования u и переменной x выполняются следующие условия.

1. Инструкция s должна являться единственным определением x , достигающим u (т.е. ои-цепочка для u состоит только из s).
2. Ни на одном пути от s к u , включая пути, проходящие через u несколько раз (но не проходящие через s повторно), нет присвоений переменной y .

Условие (1) можно проверить с помощью ои-цепочек, но как быть с условием (2)? Мы вынуждены поставить новую задачу анализа потока данных, в которой $in[B]$ представляет собой множество копирований $s: x := y$, таких, что каждый путь от начального узла к началу блока B содержит инструкцию s , и после последнего вхождения этой инструкции нет присвоений переменной y . Множество $out[B]$ можно определить аналогично, с заменой начала блока B на его конец. Мы говорим, что инструкция копирования $s: x := y$ генерируется в блоке B , если s находится в блоке B и в пределах B нет последующего присвоения переменной y . Мы говорим, что $s: x := y$ уничтожается в блоке B , если s находится вне блока, а в блоке выполняется присвоение x или y . То, что присвоение переменной x уничтожает копирование $x := y$, хорошо знакомо нам по работе с достигающими определениями, но уничтожение присвоением переменной y — специфично для данной задачи. Обратите внимание на важное следствие из того факта, что различные присвоения $x := y$ уничтожают друг друга: $in[B]$ может содержать только одну инструкцию копирования с x в левой части.

Пусть, как и ранее, U — “всеобщее” множество всех инструкций копирования в программе. Важно отметить, что разные инструкции $x := y$ различаются в этом множестве. Определим $c_gen[B]$ как множество всех инструкций копирования, генерируемых в блоке B , а $c_kill[B]$ — как множество копирований из U , уничтожаемых в B . Тогда мы можем записать следующие уравнения, связывающие только что определенные величины.

$$\begin{aligned} out[B] &= c_gen[B] \cup (in[B] - c_kill[B]) \\ in[B] &= \bigcap_{\substack{P-\text{пред-} \\ \text{шественник } B}} out[P] \quad \text{для всех } B, \text{ кроме начального} \\ in[B_1] &= \emptyset \quad \text{где } B_1 \text{ — начальный блок} \end{aligned} \tag{10.12}$$

Уравнения (10.12) идентичны уравнениям (10.10), если заменить c_kill на e_kill , а c_gen — на e_gen . Следовательно, уравнения (10.12) можно решить с помощью алгоритма 10.3, и на этом мы остановимся, однако приведем пример, который показывает некоторые нюансы оптимизации копирования.

Пример 10.19

Рассмотрим граф потока на рис. 10.35. Здесь $c_gen[B_1]=\{x := y\}$, а $c_gen[B_3]=\{x := z\}$. Кроме того, $c_kill[B_2]=\{x := y\}$, поскольку в B_2 выполняется присвоение y . И наконец, $c_kill[B_1]=\{x := z\}$, поскольку в B_1 происходит присвоение x , и по той же причине $c_kill[B_3]=\{x := y\}$.

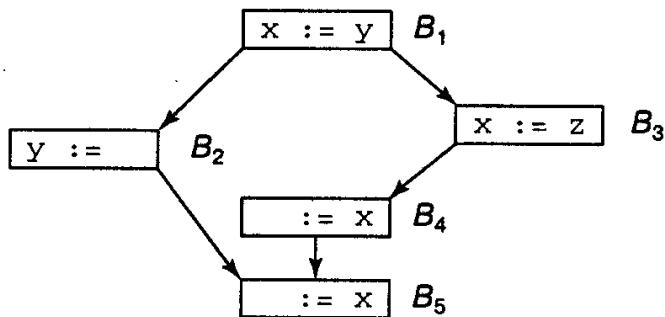


Рис. 10.35. Пример графа потока

Остальные множества c_gen и c_kill равны \emptyset . Кроме того, в соответствии с (10.12) $in[B_1] = \emptyset$. Один проход алгоритма 10.3 дает нам

$$in[B_2] = in[B_3] = out[B_1] = \{x := y\}$$

Аналогично, $out[B_2] = \emptyset$ и

$$out[B_3] = in[B_4] = out[B_4] = \{x := z\}$$

И наконец, $in[B_5] = out[B_2] \cap out[B_4] = \emptyset$.

Мы видим, что ни копирование $x := y$, ни $x := z$ не “достигают” использования x в B_5 в смысле алгоритма 10.5 (правда, оба эти определения x “достигают” B_5 в смысле достигающих определений, но это не имеет отношения к нашему рассмотрению). Следовательно, ни одно из копирований не может быть распространено, и во всех использованиях x , которых достигает определение $x := y$ (соответственно $x := z$), замена переменной x на y (соответственно z) невозможна. Мы можем подставить z вместо x в B_4 , но это не приведет к улучшению кода. \square

Теперь мы определим детали алгоритма, удаляющего инструкции копирования.

Алгоритм 10.6. Распространение копирований

Вход. Граф потока G с ои-цепочкой, дающей определения, достигающие блока B , а также множество $c_in[B]$, представляющее решение уравнений (10.12), т.е. множество копирований $x := y$, достигающих блока B по каждому из возможных путей, причем после последнего встреченного по пути копирования $x := y$ нет присвоений переменным x или y . Нам, кроме того, требуются ио-цепочки, дающие использование каждого определения.

Выход. Исправленный граф потока.

Метод. Для каждого копирования $s: x := y$ выполняем следующие действия.

1. Находим те использования x , которые достигаются рассматриваемым определением x , а именно $s: x := y$.
2. Для каждого использования x , найденного в (1), определяем, находится ли s в $c_in[B]$, где B является блоком этого конкретного использования, и кроме того, нет ли определений x или y в блоке B до рассматриваемого использования x . Вспомните, что если $s \in c_in[B]$, то s является единственным определением x , достигающим B .
3. Если s удовлетворяет условиям (2), удаляем s и заменяем все использования x , найденные в (1), переменной y . \square

Поиск вычислений, инвариантных относительно цикла

Для поиска в цикле вычислений, являющихся *инвариантами цикла* (*loop-invariant*), т.е. таких, значение которых не изменяется, пока управление остается внутри цикла, мы воспользуемся ои-цепочками. Как обсуждалось в разделе 10.4, цикл представляет собой область, состоящую из множества блоков с заголовком, доминирующим над всеми другими блоками (так что единственный путь входа в цикл проходит через заголовок). Кроме того, для каждого блока должен существовать по крайней мере один путь от него к заголовку.

Если присвоение $x := y+z$ находится в цикле, а все возможные определения y и z вне его (включая частный случай, когда y и/или z являются константами), то $y+z$ представляет собой инвариант цикла, поскольку значение этого выражения остается одним и тем же всякий раз, когда мы встречаем инструкцию $x := y+z$ в цикле. Все такие инструкции могут быть найдены с помощью ои-цепочек, т.е. списка всех точек определений y и z , достигающих присвоения $x := y+z$.

Выяснив, что значение x , вычисляемое в инструкции $x := y+z$, остается неизменным в цикле, предположим, что у нас есть другая инструкция $v := x+w$, где w определяется только вне цикла. В таком случае $x+w$ также является инвариантом цикла.

Мы можем использовать изложенные здесь идеи для выполнения неоднократных проходов по циклу, в процессе которых будет устанавливаться все больше и больше инвариантов. Если же у нас имеются как ои-, так и ио-цепочки, то нам не потребуются повторные проходы по коду. Выяснить, где используется значение x из рассматриваемого определения $x := y+z$, мы можем с помощью его ио-цепочки; после этого нам следует проверить только те использования x в пределах цикла, которые не используют других определений. Такие инвариантные относительно цикла присвоения могут быть вынесены в предзаголовок в соответствии с приведенным ниже алгоритмом, при условии, что их операнды, кроме x , также инвариантны относительно цикла.

Алгоритм 10.7. Поиск вычислений, инвариантных относительно цикла

Вход. Цикл L , состоящий из множества базовых блоков, каждый из которых содержит последовательность трехадресных инструкций. Мы полагаем, что для отдельных инструкций имеются ои-цепочки, вычисленные, как описано в разделе 10.5.

Выход. Множество трехадресных инструкций, которые вычисляют всякий раз одно и то же значение, пока управление находится в цикле L .

Метод. Мы приводим неформальное описание алгоритма с тем, чтобы были ясны принципы его работы.

1. Пометить как “инвариант” те инструкции, все операнды которых либо являются константами, либо все их достигающие определения располагаются вне цикла L .
2. Повторять шаг (3) до тех пор, пока в нем будут выявляться новые инструкции, помечаемые как “инвариант”.
3. Пометить как “инвариант” те ранее не помеченные инструкции, все операнды которых либо являются константами, либо все их достигающие определения располагаются вне цикла L , либо имеют ровно одно достигающее определение, которое представляет собой инструкцию в L , помеченную как инвариант. \square

Выполнение перемещения кода

После того как мы найдем инвариантные относительно цикла инструкции, можно применить к некоторым из них оптимизацию, известную как *перемещение кода* (code motion), при которой инструкции перемещаются в предзаголовок цикла. Три приведенных ниже условия гарантируют, что такое перемещение не изменит вычисления программы. Ни одно из этих условий не является абсолютно необходимым; мы выбрали их потому, что они просты в проверке и применении в ситуациях, возникающих в реальных программах. Позже мы обсудим возможность смягчения этих условий.

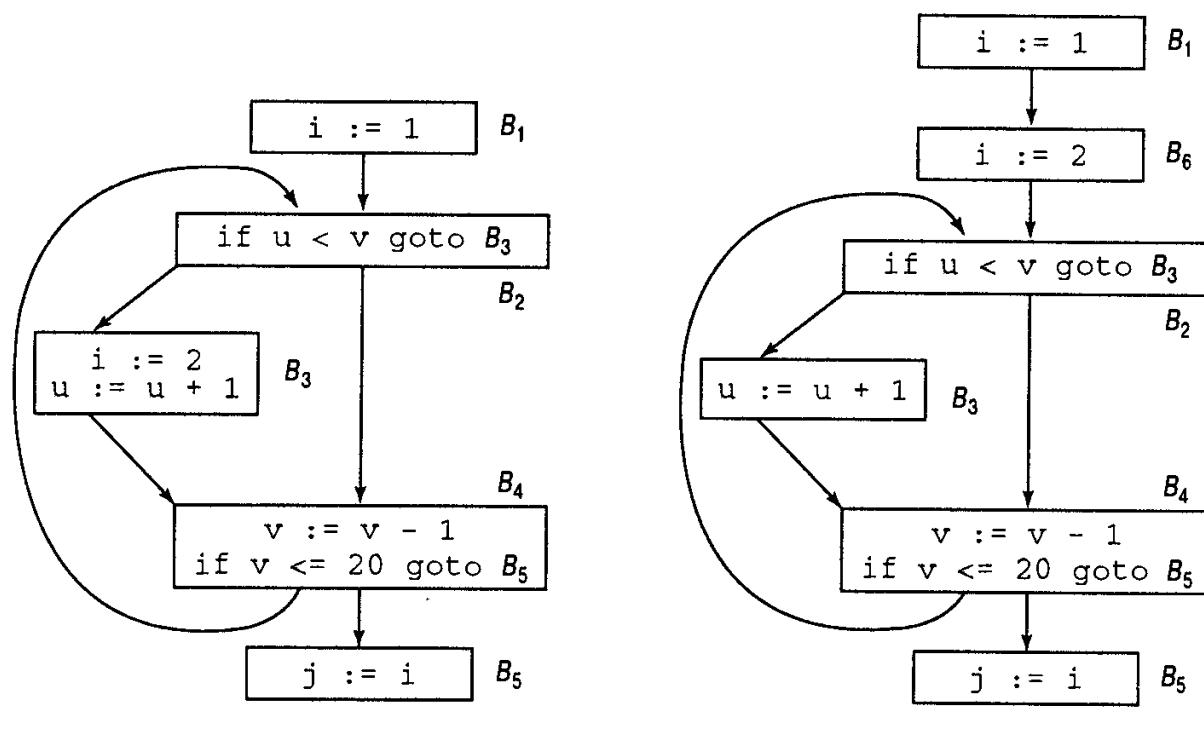
Условия для инструкции s : $x := y + z$ выглядят следующим образом.

1. Блок, содержащий s , доминирует над всеми *выходами* из цикла, где выход из цикла представляет собой узел, преемник которого находится вне цикла.
2. В цикле нет других инструкций, выполняющих присвоение переменной x . Если x представляет собой временную переменную, присвоение которой выполняется только один раз, это условие выполняется и его можно не проверять.
3. Никакое использование x в цикле не достигается никаким другим, кроме s , определением x . Если x — временная переменная, это условие обычно выполняется.

Следующие три примера поясняют перечисленные выше условия.

Пример 10.20

Перемещение инструкции, которая может не выполняться в цикле, в позицию вне цикла может изменить вычисления программы, как показано на рис. 10.36. Это наблюдение обосновывает условие (1), поскольку инструкция, доминирующая над всеми выходами, не может оказаться невыполненной при условии, что рано или поздно выполнение цикла заканчивается.



а) До преобразования

б) После преобразования

Рис. 10.36. Пример неверного перемещения кода

Рассмотрим граф потока, показанный на рис. 10.36а. Блоки B_2 , B_3 и B_4 образуют цикл с заголовком B_2 . Инструкция $i := 2$ в блоке B_3 является инвариантной по отношению к циклу. Однако B_3 не доминирует над B_4 , который является единственным выходом из цикла. Если мы переместим инструкцию $i := 2$ во вновь созданный предзаголовок B_6 , как показано на рис. 10.36б, то тем самым можем изменить значение, присваиваемое j в блоке B_5 в том случае, когда блок B_3 при выполнении цикла ни разу не выполнялся. Например, если при входе в блок B_2 $u = 30$ и $v = 25$, то на рис. 10.36а в блоке B_5 j будет присвоено значение 1, так как управление не попадет в блок B_3 . В то же время на рис. 10.36б j будет присвоено значение 2. \square

Пример 10.21

Условие (2) требуется при наличии в цикле более одного присвоения переменной x . Например, структура графа потока на рис. 10.37 та же, что и на рис. 10.36а, и мы должны создать предзаголовок B_6 , как на рис. 10.36б.

Поскольку B_2 на рис. 10.37 доминирует над выходом из цикла B_4 , условие (1) не препятствует переносу присвоения $i := 3$ в предзаголовок B_6 . Однако если мы перенесем это присвоение в подзаголовок, присвоение $i := 2$ в блоке B_3 может привести к тому, что в блоке B_5 i будет иметь значение 2, даже если мы будем следовать по пути $B_2 \rightarrow B_3 \rightarrow B_4 \rightarrow B_2 \rightarrow B_4 \rightarrow B_5$. Рассмотрим, например, значения $v = 22$ и $u = 21$ при первом входе в блок B_2 . Если присвоение $i := 3$ находится в блоке B_2 , то в блоке B_5 j получит значение 3, но если присвоение $i := 3$ вынесено в предзаголовок, то j станет равным 2. \square

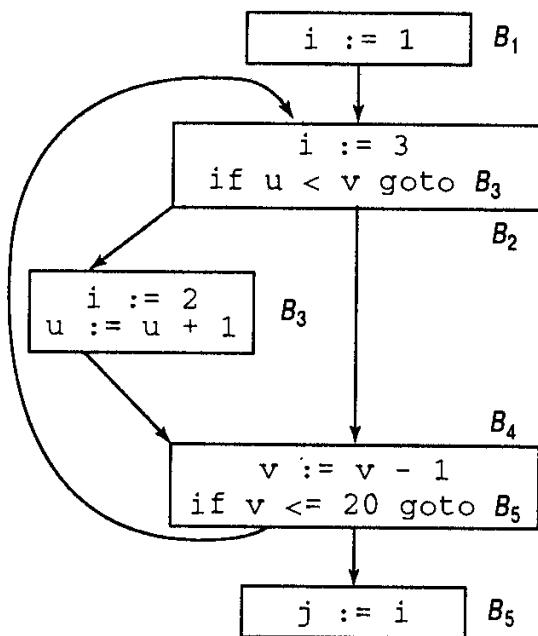


Рис. 10.37. Иллюстрация условия (2)

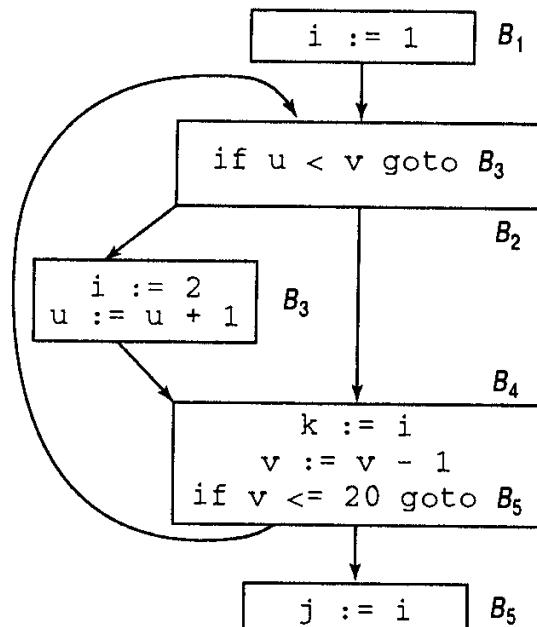


Рис. 10.38. Иллюстрация условия (3)

Пример 10.22

Рассмотрим теперь правило (3). Использование $k := i$ в блоке B_4 на рис. 10.38 достигается как определением $i := 1$ в блоке B_1 , так и определением $i := 2$ из блока B_3 . Таким образом, мы не можем переносить присвоение $i := 2$ в предзаголовок, потому что значение k , достигающее B_5 , может измениться при $u \geq v$. Например, если $u = v = 0$, то k становится равным 1 в графе потока на рис. 10.38, но если присвоение $i := 2$ выносится в предзаголовок, то k устанавливается равным 2 раз и навсегда. \square

Алгоритм 10.8. Перемещение кода

Вход. Цикл L с информацией о доминаторах и ои-цепочках.

Выход. Преобразованная версия цикла с предзаголовком и (возможно) некоторыми инструкциями, перемещенными в предзаголовок.

Метод.

1. Используем алгоритм 10.7 для поиска инвариантных по отношению к циклу инструкций.
2. Для каждой найденной на шаге (1) инструкции s , определяющей x , проверяем:
 - i) что она находится в блоке, доминирующем над всеми выходами из L ;
 - ii) что x не определяется в другом месте цикла L ;
 - iii) что все использования x в L могут быть достигнуты только определением x в инструкции s .
3. Переместим в порядке, определенном в алгоритме 10.7, каждую инструкцию s , найденную в (1) и соответствующую условиям (2i), (2ii) и (2iii), во вновь созданный предзаголовок при условии, что для любого операнда s , определенного в цикле L (в случае инструкции s , найденной на шаге (3) алгоритма 10.7), его определение уже вынесено в предзаголовок. \square

Чтобы понять, почему изменения вычислений программы невозможны, рассмотрим условия (2i) и (2ii) алгоритма 10.8, которые гарантируют, что значение x , вычисленное в s , должно быть значением x после любого выходного блока L . Когда мы переносим инструкцию s в предзаголовок, s остается определением x , достигающим конца любого выходного блока L . Условие (2iii) гарантирует, что любое использование x в цикле L использовало и продолжает использовать значение x , вычисленное в s .

Чтобы увидеть, почему преобразование не может увеличить время работы программы, заметим только, что условие (2i) гарантирует, что s выполняется как минимум один раз, когда управление достигает цикла L . После перемещения кода при достижении управлением цикла L она будет выполняться ровно один раз — в предзаголовке и нигде более.

Альтернативные стратегии перемещения кода

Мы можем до некоторой степени ослабить условие (1), если готовы пойти на риск увеличить время работы программы; естественно, условие неизменности вычислений программы при этом не отменяется. Ослабленная версия условия перемещения кода (1) (т.е. пункт (2i) алгоритма 10.8) гласит, что мы можем переместить инструкцию s присвоения переменной x , только если выполняется следующее условие.

- 1'. Блок, содержащий s , доминирует над всеми выходами из цикла или x не используется вне цикла. Например, если x — временная переменная, во многих компиляторах мы можем быть уверены, что это значение будет использоваться только в его собственном блоке. В общем случае для того, чтобы выяснить, является ли x активной в каком-либо из выходов цикла, требуется проведение анализа активных переменных.

Если алгоритм 10.8 изменен и использует условие (1'), изредка время работы программы может несколько увеличиваться, но в среднем следует ожидать повышения производительности программы. Модифицированный алгоритм может переместить в предзаголовок некоторые вычисления, которые могут не вычисляться в цикле ни разу. При этом мы рискуем не только снизить производительность программы, но и при некоторых условиях можем получить ошибку. Например, вычислению деления x/y в цикле может

предшествовать проверка равенства у нулю $y = 0$. Если мы переместим x/y в предзаголовок, возможно выполнение деления на 0. По этой причине неразумно использовать условие (1'), если такая оптимизация не может быть запрещена программистом (либо для деления надо использовать более строгое условие (1)).

Даже если присвоение $x := y+z$ не удовлетворяет ни одному из условий (2i), (2ii) и (2iii) алгоритма 10.8, мы можем вынести вычисление $y+z$ за пределы цикла. Создав новую временную переменную t , поместим в предзаголовок присвоение $t := y+z$, а затем заменим присвоение $x := y+z$ в цикле присвоением $x := t$. Во многих случаях после этого мы можем воспользоваться распространением инструкции копирования $x := t$, что уже обсуждалось ранее в этом разделе. Заметим, что если условие (2iii) алгоритма 10.8 выполнено, т.е. если все использования x в цикле L определены присвоением $x := y+z$ (теперь $x := t$), то мы, несомненно, можем удалить инструкцию $x := t$, заменив все использования x в L на t и разместив инструкции $x := t$ после каждого выхода из цикла.

Поддержание информации о потоке данных после перемещения кода

Преобразования, выполняемые алгоритмом 10.8, не изменяют информацию ои-цепочек, поскольку по условиям (2i), (2ii) и (2iii) все использования переменной, присваиваемой перемещенной инструкцией s , которые достигались последней, остаются достижимы инструкцией s и из ее нового положения. Определения переменных, используемых в s , находятся либо вне цикла L (и в этом случае достигают предзаголовка), либо в L , но тогда они также выносятся в предзаголовок перед s .

Если ои-цепочка представлена в виде списков указателей на указатели на инструкции (а не списками указателей непосредственно на инструкции), поддерживать ои-цепочки при перемещении инструкции s мы можем простым изменением указателя на s . Другими словами, для каждой инструкции s мы создаем указатель p_s , который всегда указывает на s , и помещаем его в каждую ои-цепочку, содержащую s . Тогда, куда бы мы не переместили s , мы должны только изменить p_s , независимо от того, как много ои-цепочек используется нами. Естественно, за дополнительный уровень косвенности придется платить временем работы компилятора и некоторым количеством памяти.

Если мы представляем ои-цепочки списком адресов инструкций (указателей на инструкции), мы можем поддерживать корректность информации и в этом случае. Но тогда для эффективности нам нужны ио-цепочки. При перемещении s мы можем пройти по его ио-цепочке и изменить ои-цепочки всех использований, ссылающихся на s .

Информация о доминаторах при перемещении кода меняется слабо. Предзаголовок является непосредственным доминатором заголовка, а непосредственный доминатор предзаголовка — узел, который ранее был непосредственным доминатором заголовка. Таким образом, предзаголовок вставляется в дерево доминаторов в качестве родителя заголовка.

Устранение переменных индукции

Переменная x называется *переменной индукции*, или *индуктивной переменной* (induction variable) цикла L , если каждый раз при изменении ее значения она уменьшается или увеличивается на некоторую константу. Зачастую переменная индукции увеличивается на одну и ту же константу при каждом проходе цикла, как, например, i в цикле с заголовком `for i:=1 to 10`. Однако наш метод работает с переменными, которые увеличиваются или уменьшаются нуль, один или несколько раз при проходе по циклу. Более того, число изменений может отличаться от итерации к итерации.

Обычна ситуация, когда переменная индукции, например i , служит индексом массива, а некоторая другая переменная индукции, например t , значение которой линейно зависит от i , представляет собой действительное смещение, используемое для доступа к элементам массива. Зачастую единственное применение индуктивной переменной — проверка условия окончания работы цикла. Таким образом, мы можем освободиться от i , заменив его проверку проверкой переменной t .

Приведенный далее алгоритм для упрощения его представления работает с ограниченным классом переменных индукции. Посредством добавления большего числа случаев алгоритм может быть несколько расширен, но другие расширения требуют доказательства теорем о выражениях, включающих обычные арифметические операторы.

Мы будем искать *базовые переменные индукции* (basic induction variables), представляющие собой такие переменные i , присвоения которым внутри цикла L имеют вид $i := i \pm c$, где c — константа⁹. Затем мы разыщем дополнительные переменные индукции j , которые однократно определяются в L и имеют при этом значение, представляющее собой линейную функцию от некоторой базовой переменной индукции i .

Алгоритм 10.9. Обнаружение переменных индукции

Вход. Цикл L с информацией о достигающих определениях и информацией о вычислениях, инвариантных относительно цикла (из алгоритма 10.7).

Выход. Множество переменных индукции. С каждой переменной индукции связана тройка (i, c, d) , где i — базовая переменная индукции, а c и d представляют собой константы, такие, что в точке, где определяется переменная j , ее значение равно $c * i + d$. Мы говорим, что j принадлежит семейству i ; базовая переменная индукции i принадлежит своему собственному семейству.

Метод.

1. Находим все базовые переменные индукции путем сканирования инструкций L , используя при этом информацию о вычислениях, инвариантных относительно цикла. С каждой базовой переменной индукции связывается тройка $(i, 1, 0)$.
2. Находим переменные k с единственным присвоением в L одного из следующих типов:
 $k := j * b$, $k := b * j$, $k := j / b$, $k := j \pm b$, $k := b \pm j$
где b — константа, а j — переменная индукции (может быть как базовой, так и небазовой).

Если j — базовая переменная индукции, то k принадлежит семейству j . Тройка k зависит от определяющей эту переменную инструкции. Например, если k определено как $k := j * b$, то тройка, связанная с k , — $(j, b, 0)$. Тройки для других случаев определяются аналогично.

Если j — небазовая переменная, то пусть j принадлежит семейству i . Тогда наши дополнительные требования состоят в следующем:

- a) между единственной точкой присвоения j в L и присвоением k нет присвоения i ,
- b) ни одно определение j вне L не достигает k .

Типичный, простой для проверки случай — когда определения k и j представляют собой определения временных переменных в одном и том же блоке. В общем случае

⁹ В нашем рассмотрении переменных индукции “+” означает оператор сложения, а не обобщенный оператор (то же относится и к прочим стандартным арифметическим операторам).

информация о достигающих определениях обеспечит проверку, необходимую нам при анализе графа потока цикла L в поисках таких блоков (а следовательно, и таких определений) на путях между присвоением j и присвоением k .

Мы вычисляем тройку для k по тройке (i, c, d) для j и по инструкции, определяющей k . Например, определение $k := b * j$ приводит к тройке $(i, b * c, b * d)$ для k . Заметим, что умножения в $b * c$ и $b * d$ могут быть выполнены в процессе анализа, поскольку b , c и d являются константами. \square

После того как найдены семейства переменных индукции, мы изменяем инструкции, вычисляющие переменные индукции с тем, чтобы они использовали сложение и вычитание вместо умножения. Замещение дорогостоящих инструкций более дешевыми называется *снижением стоимости* (strength reduction).

Пример 10.23

Цикл, состоящий из блока B_2 на рис. 10.39a, имеет базовую переменную индукции i , поскольку единственное присвоение i в цикле увеличивает значение этой переменной на 1. Семейство i содержит t_2 , поскольку имеется единственное присвоение t_2 , содержащее в правой части $4 * i$. Таким образом, тройка для t_2 представляет собой $(i, 4, 0)$. Аналогично j является единственной базовой переменной индукции цикла, состоящего из блока B_3 , а t_4 — членом семейства j с тройкой $(j, 4, 0)$.

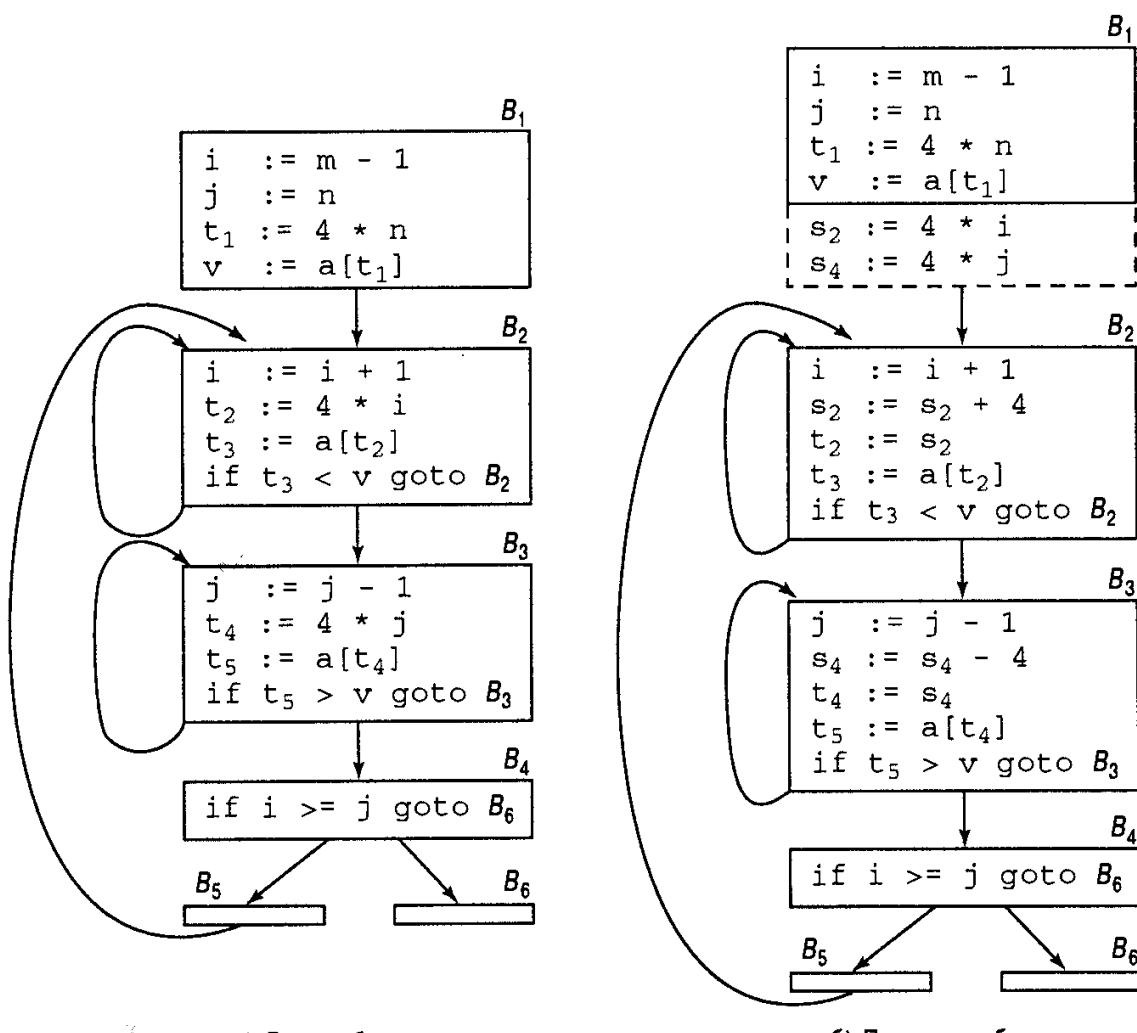


Рис. 10.39. Снижение стоимости

Мы можем также рассмотреть переменные индукции во внешнем цикле, с заголовком B_2 и блоками B_2, B_3, B_4, B_5 . В большем цикле переменными индукции являются и i , и j , а t_2 и t_4 представляют собой переменные индукции с тройками $(i, 4, 0)$ и $(j, 4, 0)$ соответственно.

Граф потока на рис. 10.39б получен из графа потока на рис. 10.39а с помощью рассмотренного далее алгоритма. Мы обсудим сделанные преобразования ниже. \square

Алгоритм 10.10. Применение снижения стоимости к переменным индукции

Вход. Цикл L с информацией о достигающих определениях и семействах переменных индукции, вычисленных в соответствии с алгоритмом 10.9.

Выход. Преобразованный цикл.

Метод. Рассмотрим каждую из базовых переменных индукции i . Для каждой переменной индукции j из семейства i с тройкой (i, c, d) выполним следующее.

1. Создадим новую переменную s (но если две переменные j_1 и j_2 имеют одинаковые тройки, то для обоих создается только одна новая переменная).
2. Заменим присвоение переменной j присвоением $j := s$.
3. Непосредственно после каждого присвоения $i := i + n$ в L , где n — константа, добавим

$s := s + c * n$

где выражение $c * n$ представляет собой константу, поскольку константами являются и c , и n . Поместим s в семейство i с тройкой (i, c, d) .

4. Остается убедиться, что s инициализируется значением $c * i + d$ при входе в цикл. Инициализация может быть размещена в конце предзаголовка. Инициализация состоит из

```
s := c*i      /* либо просто s := i, если c равно 1 */
s := s+d      /* опускаем, если d равно 0 */
```

Заметим, что s является переменной индукции в семействе i . \square

Пример 10.24

Предположим, что мы рассматриваем циклы на рис. 10.39а от внутреннего к внешнему. Поскольку рассмотрение внутренних циклов B_2 и B_3 очень похоже одно на другое, мы будем говорить только о цикле вокруг B_3 . В примере 10.23 мы видели, что базовая переменная индукции в цикле вокруг B_3 — j , а вторая переменная индукции — t_4 с тройкой $(j, 4, 0)$. На шаге (1) алгоритма 10.10 создается новая переменная s_4 . На шаге (2) присвоение $t_4 := 4 * j$ заменяется присвоением $t_4 := s_4$. На шаге (4) после присвоения $j := j - 1$ вставляется присвоение $s_4 := s_4 - 4$, где -4 получено перемножением -1 в присвоении j и 4 из тройки $(j, 4, 0)$ для t_4 .

Поскольку B_1 служит в качестве предзаголовка цикла, мы можем поместить инициализацию s_4 в конце блока B_1 , содержащего определение j . Добавленные инструкции показаны на рис. 10.39б в пунктирном прямоугольнике, дополняющем блок B_1 .

При рассмотрении внешнего блока граф потока выглядит так, как показано на рис. 10.39б. Имеются четыре переменные i , s_2 , j и s_4 , которые могут рассматриваться как переменные индукции. Однако на шаге (3) алгоритм 10.10 размещает вновь создан-

ные переменные в семействах i и j соответственно для упрощения устранения i и j с помощью приведенного далее алгоритма. \square

После снижения стоимости мы можем обнаружить, что единственное использование некоторых переменных индукции состоит в выполнении проверки завершения выполнения цикла. Такую проверку переменной индукции мы можем заменить проверкой другой переменной индукции. Например, если i и t представляют собой переменные индукции, такие, что значение t всегда равно учетверенному значению i , то проверка $i \geq j$ эквивалентна проверке $t \geq 4 * j$. После такой замены может оказаться возможным удаление i . Заметим, однако, что если $t = -4 * i$, то следует заменить и оператор сравнения, так как в этом случае $i \geq j$ эквивалентно $t \leq -4 * j$. В следующем алгоритме мы рассматриваем случай, когда постоянный сомножитель положителен, оставляя случай отрицательной константы читателю в качестве упражнения.

Алгоритм 10.11. Устранение переменной индукции

Вход. Цикл L с информацией о достигающих определениях и об инвариантных относительно цикла вычислениях (от алгоритма 10.7), а также с информацией об активных переменных.

Выход. Преобразованный цикл.

Метод.

1. Рассмотрим каждую базовую переменную индукции i , единственное использование которой — вычисление других переменных индукции в своем семействе, а также выполнение проверок в условном ветвлении. Возьмем некоторую переменную j из семейства i , предпочтительно такую, чтобы c и d в ее тройке (i, c, d) были как можно проще (например, $c = 1$ и $d = 0$), и изменим каждую проверку $c \neq i$ так, чтобы она использовала значение j . Далее мы предполагаем, что значение c положительно. Проверка вида $if i \neq 0 \text{ goto } B$, где x не является индуктивной переменной, заменяется кодом

```
r := c*x /* r := x, если c равно 1 */  
r := r+d /* опускаем при d, равном 0 */  
if j  $\neq 0$  r goto B
```

где r — новая временная переменная. Случай $if x \neq 0 \text{ goto } B$ обрабатывается аналогично. Если в проверке $if i_1 \neq 0 \text{ goto } B$ принимают участие две переменные индукции, то мы проверяем возможность замены как i_1 , так и i_2 . Простейший случай осуществляется при наличии j_1 с тройкой (i_1, c_1, d_1) и j_2 с тройкой (i_2, c_2, d_2) , если выполняются условия $c_1 = c_2$ и $d_1 = d_2$ — тогда $i_1 \neq 0$ эквивалентно $j_1 \neq 0$. В более сложных случаях замена проверки может не дать результата, поскольку может оказаться необходимым внесение двух операций умножения и одного сложения, в то время как устранение i_1 и i_2 экономит всего лишь две инструкций.

И наконец, удаляем все присвоения устраниенным переменным индукции из цикла L , поскольку они становятся бесполезными.

2. Теперь рассмотрим каждую переменную индукции j , для которой алгоритм 10.10 добавил присвоение $j := s$. Вначале проверим, что между внесенной инструкцией $j := s$ и любым использованием j нет присвоений переменной s . Обычно j ис-

пользуется в том же блоке, где и определяется, что упрощает данную проверку. В противном случае для выполнения проверки нам потребуется информация о достигающих определениях и дополнительный анализ графа. После этого мы заменяем все использования j использованием s и удаляем инструкцию $j := s$. \square

Пример 10.25

Рассмотрим граф потока на рис. 10.39б. Внутренний цикл вокруг B_2 содержит две переменные индукции — i и s_2 , однако ни одна из них не может быть устранина, поскольку s_2 используется в качестве индекса массива a , а i используется в проверке вне цикла. Аналогично цикл вокруг B_3 содержит переменные индукции j и s_4 , которые также не могут быть устранины.

Применим алгоритм 10.11 ко внешнему циклу. При создании в соответствии с алгоритмом 10.10 новых переменных s_2 и s_4 , как обсуждалось в примере 10.24, s_2 помещается в семейство i , а s_4 — в семейство j . Рассмотрим семейство i . Единственное использование i состоит в проверке прекращения работы цикла в блоке B_4 , так что переменная i является кандидатом на устранение на шаге (1) алгоритма 10.11. Проверка в блоке B_4 включает две переменные индукции i и j . К счастью, семейства i и j содержат s_2 и s_4 с одинаковыми константами в их тройках ($(i, 4, 0)$ и $(j, 4, 0)$ соответственно). Таким образом, проверка $i \geq j$ может быть заменена проверкой $s_2 \geq s_4$, что обеспечивает возможность устранения переменных i и j .

Шаг (2) алгоритма 10.11 применяет распространение копий к вновь созданным переменным, заменяя t_2 и t_4 соответственно переменными s_2 и s_4 . \square

Переменные индукции и выражения, инвариантные относительно цикла

В алгоритмах 10.9 и 10.10 мы можем допустить использование выражений, инвариантных относительно цикла, вместо констант. Однако в таком случае тройка (i, c, d) для переменной индукции j может вместо констант содержать инвариантные выражения. Вычисление этих выражений должно выполняться вне цикла L , в предзаголовке. Кроме того, поскольку промежуточный код требует, чтобы на одну инструкцию приходилось не более одного оператора, для вычисления выражений мы должны быть готовы генерировать инструкции промежуточного кода. Замена проверок в алгоритме 10.11 требует, чтобы был известен знак сомножителя c — по этой причине может быть разумным ограничиться только случаями, когда c — известная константа.

10.8. Работа с альтернативными именами

Если два или несколько выражений означают один и тот же адрес памяти, мы говорим, что эти выражения представляют собой *альтернативные имена*, или *псевдонимы* (aliases) друг друга. В этом разделе мы рассмотрим анализ потока данных при наличии указателей и процедур, которые вносят в рассмотрение псевдонимы.

Наличие указателей делает анализ потока данных более сложным, так как они вызывают неопределенность в отношении того, что именно используется и определяется. Если нам ничего не известно о том, куда может указывать указатель p , единственным безопасным предположением является предположение о том, что косвенное присвоение с помощью указателя может потенциально привести к изменению (т.е. определению) любой переменной. Мы также должны предположить, что любое использование данных по-

средством указателя, например `x := *p`, потенциально может использовать любую переменную. Такие предположения дают в результате больше активных переменных и достигающих определений, чем есть на самом деле, и уменьшают количество доступных выражений. К счастью, мы можем использовать анализ потока данных для фиксации того, на что может указывать данный указатель, что позволяет нам получить более точную информацию из выполняемых нами других видов анализа потока данных.

Так же, как и при присвоении с использованием указателей, когда мы встречаемся с вызовом процедуры, мы можем вычислить множество переменных, которые могут изменяться в результате этого вызова, не прибегая к наихудшему предположению о том, что измениться в результате вызова процедуры может любая переменная. Как и в случае любой оптимизации кода, мы можем допускать ошибки только в безопасном, консервативном направлении — т.е. множества переменных, значения которых “могут быть” изменены или использованы, могут включать не только переменные, которые реально изменяются или используются при некоторых выполнениях программы. Мы, как обычно, будем пытаться по возможности приблизить вычисляемые множества к истинным множествам изменяемых и используемых переменных, стараясь не затрачивать излишних усилий и, естественно, не допуская изменений вычислений программы.

Простой язык ~~указателей~~

Для определенности рассмотрим язык, в котором имеются элементарные типы данных (т.е. целые и действительные числа), требующие по одному слову каждый, и массивы этих типов. Пусть также имеются указатели на эти элементы и массивы, но не на другие указатели. Мы будем довольствоваться знанием о том, что указатель `p` указывает на некоторый элемент массива `a`, не определяя, на какой именно. Такая группировка всех элементов массива в единое целое при рассмотрении указателей имеет определенный смысл. Обычно указатель используется в качестве курсора для прохода по всему массиву, так что более детальный анализ потока данных, если мы вообще можем довести его до конца, зачастую сообщает в результате, что указатель `p` может указывать на любой из элементов массива `a`.

Мы должны также принять определенные предположения о том, какие арифметические операторы над указателями семантически имеют смысл. Если указатель `p` указывает на примитивный (“однословный”) элемент данных, то любая арифметическая операция над `p` дает значение, которое может быть целым, но не указателем. Если же `p` указывает на массив, то прибавление или вычитание целого числа оставляет `p` указывающим на элемент массива, в то время как остальные арифметические операции дают значение, не являющееся указателем. Хотя не все языки запрещают, например, перемещение указателя из одного массива `a` в другой массив `b` путем прибавления к указателю, такое действие будет зависеть от конкретной реализации, поскольку нужно обеспечить следование массива `b` в памяти за массивом `a`. Наша точка зрения заключается в том, что оптимизирующий компилятор в решении вопроса об используемой оптимизации должен полагаться только на определение языка. Каждый создатель компилятора, однако, должен сам решить, какую именно оптимизацию может выполнять компилятор.

Воздействие присвоений указателей

При этих предположениях единственными переменными, которые могут использоваться в качестве указателей, являются переменные, объявленные как указатели, и временные переменные, которые получают значение, являющееся указателем, плюс или ми-

нус константа. Мы будем говорить обо всех этих переменных как об указателях. Наши правила для определения того, на что может указывать указатель p , следующие.

1. При наличии инструкции присвоения $s: p := \&a$ непосредственно за s указатель p может указывать только на a . Если a является массивом, то после присвоения вида $p := \&a + c$ (где c — константа) p может указывать только на a^{10} . Как обычно, $\&a$ означает положение первого элемента массива a в памяти.
2. Если имеется оператор присвоения вида $s: p := q + c$, где c — целое число, отличное от 0, а p и q — указатели, то непосредственно после s p может указывать на любой массив, на который указывает q перед s , и никуда более.
3. Если имеется присвоение $s: p := q$, то непосредственно после s p может указывать на то, на что q указывал перед s .
4. После любого другого присвоения p нет объекта, на который может указывать p ; такое присвоение, вероятно (в зависимости от семантики языка), бессмысленно.
5. После любого присвоения переменной, отличающейся от p , p указывает на то же, что и до присвоения. Заметим, что это правило предполагает, что указатель не может указывать на указатель. Ослабление этого предположения не делает рассмотрение существенно более сложным, и мы оставляем читателю обобщение в качестве упражнения.

Мы определим $in[B]$ для блока B как функцию, дающую для каждого указателя p множество переменных, на которые может указывать p в начале блока B . Формально $in[B]$ представляет собой множество пар вида (p, a) , где p — указатель, а a — переменная, означающих, что p может указывать на a . На практике $in[B]$ может быть представлено в виде списка для каждого указателя; список для p дает множество всех a , таких, что $(p, a) \in in[B]$. $out[B]$ определяется для конца блока B аналогично.

Мы определим функцию передачи $trans_B$, которая определяет влияние блока B , т.е. функция $trans_B$ принимает в качестве аргумента множество S пар вида (p, a) для указателя p и переменной a , не являющейся указателем, и возвращает еще одно множество T . Множество, к которому применяется функция $trans_B$, — $in[B]$; в результате этого применения мы получаем множество $out[B]$. Нам необходимо только указать, каким образом $trans$ вычисляется для отдельной инструкции; $trans_B$ в таком случае будет представлять собой композицию $trans_s$ для каждой инструкции s блока B . Правила для вычисления $trans$ следующие.

1. Если s представляет собой $p := \&a$ или $p := \&a + c$ в случае, когда a является массивом, то

$$trans_s(S) = (S - \{(p, b) \mid b - \text{любая переменная}\}) \cup \{(p, a)\}$$

2. Если s имеет вид $p := q + c$ для указателя q и ненулевого целого c , то

$$trans_s = (S - \{(p, b) \mid b - \text{любая переменная}\}) \\ \cup \{(p, b) \mid (q, b) \in S \text{ и } b - \text{переменная массива}\}$$

Заметим, что это правило имеет смысл, даже если $p = q$.

¹⁰ В этом разделе $+$ означает оператор сложения, а не обобщенный оператор.

3. Если s имеет вид $p := q$, то
 $trans_s = (S - \{(p, b) \mid b - \text{любая переменная}\}) \cup \{(p, b) \mid (q, b) \in S\}$
4. Если s присваивает указателю p любое другое выражение, то
 $trans_s(S) = S - \{(p, b) \mid b - \text{любая переменная}\}$
5. Если s не является присвоением указателю, то $trans_s(S) = S$.

Теперь мы можем записать уравнения, связывающие in , out и $trans$:

$$\begin{aligned} out[B] &= trans_B(in[B]) \\ in[B] &= \bigcup_{\substack{P-\text{пред-} \\ \text{шественник } B}} out[P] \end{aligned} \quad (10.13)$$

где в случае, если B состоит из инструкций s_1, s_2, \dots, s_k , то

$$trans_B(S) = trans_{s_k} \left(\dots \left(trans_{s_2} \left(trans_{s_1}(S) \right) \right) \dots \right).$$

Уравнения (10.13) могут быть решены, по сути, так же, как и достигающие определения в алгоритме 10.2. Поэтому мы не будем подробно рассматривать соответствующий алгоритм, удовлетворившись примером.

Пример 10.26

Рассмотрим граф потока, приведенный на рис. 10.40. Мы считаем, что a представляет собой массив, а c — целое число; p и q являются указателями. Изначально мы устанавливаем $in[B_1] = \emptyset$. Тогда $trans_{B_1}$ представляет собой результат удаления любых пар с первым компонентом q , а затем добавления пары (q, c) ; тем самым утверждается, что q указывает на c . Следовательно,

$$out[B_1] = trans_{B_1}(\emptyset) = \{(q, c)\}$$

Тогда $in[B_2] = out[B_1]$. Воздействие $p := \&c$ состоит в замене всех пар с первым компонентом p парой (p, c) . Влияние $q := \&(a[2])$ заключается в замене пар с первым компонентом q парой (q, a) . Обратите внимание, что присвоение $q := \&(a[2])$ в действительности является присвоением вида $q := \&a + c$, где c — константа. Теперь мы можем вычислить

$$out[B_2] = trans_{B_2}(\{(q, c)\}) = \{(p, c), (q, a)\}$$

Аналогично $in[B_3] = \{(q, c)\}$ и $out[B_3] = \{(p, a), (q, c)\}$.

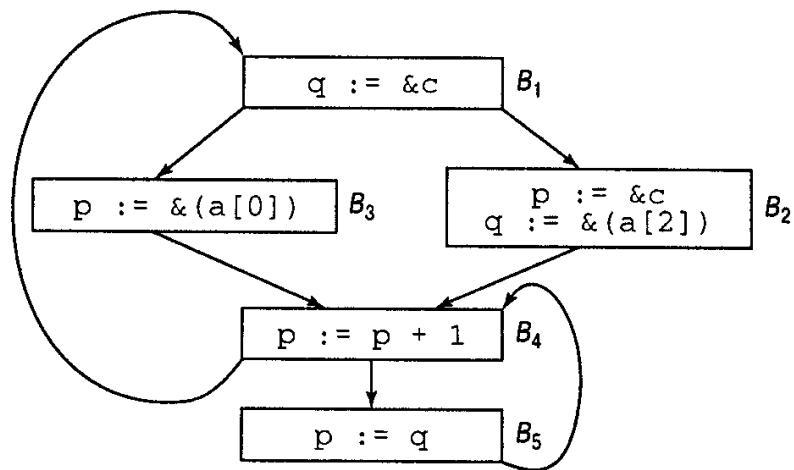


Рис. 10.40. Граф потока с показанными операциями с указателями

Далее мы находим $in[B_4] = out[B_2] \cup out[B_3] \cup out[B_5]$. Предварительно $out[B_5]$ проинициализировано значением \emptyset и не изменяется при этом проходе. Однако $out[B_2] = \{(p,c), (q,a)\}$ и $out[B_3] = \{(p,a), (q,c)\}$, так что

$$in[B_4] = \{(p,a), (p,c), (q,a), (q,c)\}$$

Влияние присвоения $p := p+1$ в B_4 состоит в отбрасывании возможности, что p указывает не на массив, т.е.

$$in[B_4] = trans_{B_4}(in[B_4]) = \{(p,a), (q,a), (q,c)\}$$

Заметим, что если выполняется B_2 , то p указывает на c , что делает семантически бессмысленным действием косвенное использование p после присвоения $p := p+1$ в блоке B_4 . Следовательно, приведенный на рис. 10.40 граф потока не является реалистичным (зато выполняет свою задачу, иллюстрируя выводы, которые мы можем сделать при рассмотрении указателей).

Продолжая работу, получаем $in[B_5] = out[B_4]$, и $trans_{B_5}$ копирует объекты указателя q и передает их указателю p . Поскольку в $in[B_5]$ q может указывать на a или c , получаем

$$out[B_5] = \{(p,a), (p,c), (q,a), (q,c)\}$$

При следующем проходе мы находим, что $in[B_1] = out[B_4]$, так что $out[B_1] = \{(p,a), (q,c)\}$. Это значение является также новым значением $in[B_2]$ и $in[B_3]$, но эти новые значения не приводят к изменению значений $out[B_2]$, $out[B_3]$ или $out[B_4]$. Следовательно, мы нашли искомый ответ. \square

Использование информации об указателях

Предположим, что $in[B]$ является множеством переменных, на которые указывает каждый указатель в начале блока B , и что у нас есть ссылка на указатель p внутри блока B . Начиная с $in[B]$, применим $trans_s$ к каждой инструкции s блока B , которая предшествует ссылке на p . Такое вычисление говорит нам, на что может указывать p в некоторой инструкции, где эта информация представляет интерес.

Предположим теперь, что мы определили, на что каждый указатель при его использовании в качестве косвенной ссылки может указывать, находясь либо справа, либо слева от символа присвоения. Как нам использовать эту информацию для получения более точных решений обычных задач анализа потока данных? В любом случае мы должны рассмотреть, в каком направлении ошибка является консервативной, и использовать имеющуюся информацию так, чтобы могли быть допущены только консервативные ошибки. Рассмотрим для иллюстрации два примера анализа: достигающих определений и активных переменных.

Для вычисления достигающих определений мы можем использовать алгоритм 10.2, однако мы должны знать значения множеств $kill$ и gen для этого блока. Эти величины вычисляются, как обычно для инструкций, в которых отсутствует косвенное присвоение посредством указателей. Мы полагаем, что косвенное присвоение $*p := a$ генерирует определение каждой переменной b , такой, что p может указывать на b . Это предположение консервативно, поскольку, как обсуждалось в разделе 10.5, в общем случае предположение, что определение может достигать некоторой точки, в то время как на самом деле оно не достигает ее, консервативно.

При вычислении множества $kill$ предположим, что присвоение $*p := a$ уничтожает определения b только в том случае, если b — не массив, и переменная p может указывать только на b . Если p может указывать на две и более переменных, то мы не считаем,

что определения всех этих переменных уничтожаются. Тем самым мы вновь выбираем консервативный путь, поскольку разрешаем определениям b проходить через присвоение $*p := a$, кроме случая, когда мы можем доказать, что это присвоение переопределяет b . Другими словами, при наличии сомнений мы полагаем, что определение достигает интересующей нас точки.

Для активных переменных мы можем использовать алгоритм 10.4, но при этом мы должны пересмотреть определение множеств *def* и *use* для инструкций вида $*p := a$ и $a := *p$. Инструкция $*p := a$ использует только a и p . Мы говорим, что она определяет b только в том случае, когда b — единственная переменная, на которую может указывать p . Такое предположение позволяет использованием b проходить через присвоения посредством указателя, если только нет гарантии, что они блокируются этим присвоением. Таким образом, мы никогда не потребуем, чтобы b была неактивна, если на самом деле это не так. Инструкция $a := *p$, несомненно, представляет собой определение a . Она также представляет использование p и всех переменных, на которые p может указывать. Максимизируя количество возможных использований, мы максимизируем нашу оценку количества активных переменных, а обычно это — консервативный путь. Например, мы можем генерировать код для хранения неактивной переменной, но мы ни в коем случае не должны оставить несохраненной активную переменную.

Анализ межпроцедурного потока данных

До сих пор мы говорили о “программах”, которые представляли собой отдельные процедуры и, следовательно, единые графы потоков. Сейчас мы познакомимся с тем, каким образом собирать информацию из нескольких взаимодействующих процедур. Базовая идея состоит в определении, каким образом каждая процедура влияет на множества *gen*, *kill*, *use* или *def* других процедур, с последующим вычислением интересующей нас информации о потоке данных для каждой процедуры отдельно, как и ранее.

В процессе анализа потока данных мы будем иметь дело с альтернативными именами, определяемыми параметрами в вызовах процедур. Поскольку две глобальные переменные не могут означать один и тот же адрес памяти, как минимум один из пары псевдонимов должен быть формальным параметром. Поскольку формальные параметры могут передаваться в процедуры, два формальных параметра могут быть псевдонимами.

Пример 10.27

Предположим, что у нас есть процедура p с двумя формальными параметрами x и y , переданными по ссылке. На рис. 10.41 мы видим ситуацию, в которой $b+x$ вычисляется в блоках B_1 и B_3 . Предположим, что единственный путь из блока B_1 в блок B_3 проходит через блок B_2 и что вдоль этого пути нет присвоений переменным b или x . Будет ли тогда $b+x$ доступно в блоке B_3 ? Ответ зависит от того, могут ли x и y означать один и тот же адрес памяти. Например, это может быть вызов $p(z, z)$ или, возможно, вызов $p(u, v)$, где u и v — формальные параметры другой процедуры $q(u, v)$, и при этом возможен вызов $q(z, z)$.

Аналогично x и y могут быть псевдонимами, если x — формальный параметр, например, в $p(x, w)$, а y — переменная с областью видимости, доступной для некоторой процедуры q , вызывающей p , например, как $p(y, t)$. Могут быть еще более сложные ситуации, делающие x и y псевдонимами друг друга, и мы разработаем некоторые общие правила, которые дадут нам возможность находить все пары псевдонимов такого типа. \square

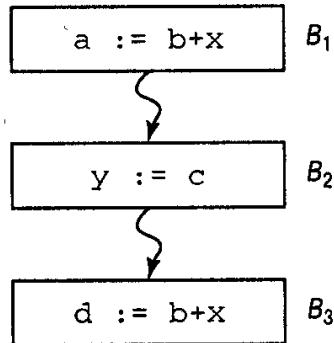


Рис. 10.41. Иллюстрация проблемы альтернативных имен

В ряде ситуаций оказывается консервативным отказ от рассмотрения переменных как псевдонимов друг друга. Например, в случае достигающих определений, если мы утверждаем, что определение a уничтожается определением b , мы должны убедиться в том, что a и b являются псевдонимами в момент определения b . В других случаях консервативным является рассмотрение, в случае сомнений, переменных как псевдонимов друг друга. Пример 10.27 является одним из таких случаев. Если доступное выражение $b+x$ не должно уничтожаться определением y , нам следует убедиться, что ни b , ни x не могут быть псевдонимами y .

Модель кода с вызовами процедур

Для иллюстрации, каким образом можно работать с псевдонимами, рассмотрим язык, допускающий рекурсивные процедуры, любая из которых может обращаться как к локальным, так и глобальным переменным. Данные, доступные процедуре, состоят из глобальных данных и данных, локальных по отношению к данной процедуре; т.е. язык не имеет блочной структуры. Параметры передаются в процедуры по ссылке. Мы требуем, чтобы все процедуры имели графы потоков с единственным *входом* (начальным узлом) и *выходом* (узлом, который возвращает управление вызываемой процедуре). Для удобства предположим, что каждый узел лежит на некотором пути от входа к выходу.

Теперь предположим, что мы находимся в процедуре p и встретились с вызовом процедуры $q(u, v)$. Если нас интересует вычисление достигающих определений, доступных выражений или любого из других видов анализа потока данных, мы должны знать, может ли вызов $q(u, v)$ изменить значение некоторой переменной. Мы можем только найти множество, включающее все переменные, значения которых изменяются, и, возможно, некоторые из тех, что остаются неизменными. Будучи внимательными, мы можем уменьшить последний класс переменных, получая хорошее приближение к истинному значению и допуская ошибки только в консервативную сторону.

Единственными переменными, значения которых может определить вызов $q(u, v)$, являются глобальные переменные и переменные u и v , которые могут быть локальными по отношению к p . Определения локальных переменных q после возврата из вызова не приводят ни к каким последствиям. Даже если $p = q$, изменяться будет другая копия локальных по отношению к q переменных, и после возврата эти копии уничтожаются. Очень просто определить, какие глобальные переменные явным образом определяются процедурой q — для этого достаточно посмотреть, какие из них определяются в q или в процедуре, вызываемой q . Кроме того, u и/или v , которые могут быть глобальными, изменяются, если q содержит определения соответственно своего первого и/или второго формального параметра или если эти формальные параметры процедурой q передаются

в качестве фактических параметров другой процедуре, которая и определяет их. Однако не каждая переменная, изменяемая вызовом `q`, определяется процедурой `q` или одной из вызываемых ею процедур явно, поскольку переменные могут иметь псевдонимы.

Вычисление псевдонимов

Перед тем как мы сможем ответить на вопрос, какие переменные могут быть изменены данной процедурой, мы должны разработать алгоритм для поиска псевдонимов. Мы воспользуемся простейшим подходом для вычисления отношения \equiv над переменными, которое формализует понятие “может быть псевдонимом для”. При таком подходе мы не сможем отличить вхождения переменной в различные вызовы одной и той же процедуры, хотя различим переменные, локальные по отношению к разным процедурам, но имеющие один и тот же идентификатор.

Для простоты мы не будем пытаться различать множества псевдонимов в разных точках программы. Вместо этого мы будем считать, что если одна переменная может быть псевдонимом другой, то она может быть им всегда. И наконец, мы примем консервативное предположение о транзитивности отношения \equiv , т.е. переменные группируются в классы эквивалентности, и одна переменная может быть псевдонимом другой тогда и только тогда, когда они находятся в одном и том же классе.

Алгоритм 10.12. Простое вычисление псевдонимов

Вход. Набор процедур и глобальных переменных.

Выход. Отношение эквивалентности \equiv , обладающее тем свойством, что если в некоторой точке программы x и y являются псевдонимами друг друга, то $x \equiv y$; обратное не обязательно справедливо.

Метод.

1. При необходимости переименуем переменные так, чтобы никакие две процедуры не использовали одни и те же формальные параметры или локальные переменные и чтобы никакие локальные, формальные и глобальные переменные не имели одинаковых имен.
2. Если есть процедура $p(x_1, x_2, \dots, x_n)$ и вызов $p(y_1, y_2, \dots, y_n)$ этой процедуры, установим $x_i \equiv y_i$ для всех i , т.е. каждый формальный параметр может быть псевдонимом любого соответствующего действительного параметра.
3. Получим рефлексивное и транзитивное замыкания соответствий фактических и формальных параметров добавлением отношений
 - a) $x \equiv y$, если $y \equiv x$.
 - b) $x \equiv z$, если для некоторого y выполняется $x \equiv y$ и $y \equiv z$.

□

Пример 10.28

Рассмотрим набросок из трех процедур, показанный на рис. 10.42; мы считаем, что параметры в процедуры передаются по ссылке. Имеются две глобальные (`g` и `h`) и две локальные (`i` для процедуры `main` и `k` для процедуры `two`) переменные. Процедура `one` имеет формальные параметры `w` и `x`, процедура `two` — `y` и `z`, у процедуры `main` параметров нет. Следовательно, в переименовании переменных нет необходимости. Приступим к вычислению псевдонимов, основанных на соответствии формальных и фактических параметров.

Вызов процедуры `one` процедурой `main` дает нам $h \equiv w$ и $i \equiv x$. Первый вызов процедуры `two` процедурой `one` дает $w \equiv y$ и $w \equiv z$, а второй — $g \equiv y$ и $x \equiv z$.

Вызов процедуры `one` процедурой `two` дает $k \equiv w$ и $y \equiv x$. При получении транзитивного замыкания отношения альтернативности имен, мы найдем, что в этом примере все переменные могут быть псевдонимами друг друга. \square

```
global g, h;
procedure main();
    local i;
    g := ...;
    one(h, i)
end;
procedure one(w, x);
    x := ...;
    two(w, w);
    two(g, x)
end;
procedure two(y, z);
    local k;
    h := ...;
    one(k, y)
end;
```

Рис. 10.42. Пример процедур

Вычисление псевдонимов в соответствии с алгоритмом 10.12 не часто приводит к столь интенсивному использованию псевдонимов, как в рассмотренном нами примере. Интуитивно понятно, что не так уж часто приходится ожидать, что переменные двух разных типов окажутся псевдонимами. Кроме того, программист использует для своих переменных типы, соответствующие смыслу используемых переменных. Например, если первый формальный параметр процедуры `p` представляет собой скорость, следует ожидать, что везде первым фактическим параметром этой процедуры будет скорость. Таким образом, в реальных программах следует ожидать небольшие группы возможных псевдонимов.

Анализ потока данных при наличии вызовов процедур

Рассмотрим в качестве примера, как могут быть вычислены доступные выражения при наличии вызовов процедур, параметры в которые передаются по ссылке. Как и в разделе 10.6, мы должны определить, когда может быть определена переменная (уничтожая тем самым выражение), а также когда генерируется (т.е. вычисляется) выражение.

Для каждой процедуры `p` мы можем определить множество $change[p]$, значение которого представляет собой множество глобальных переменных и формальных параметров `p`, которые могут изменяться в процессе выполнения `p`. При этом мы не можем считать переменную неизменной, если изменяется хотя бы один член ее класса эквивалентности.

Пусть $def[p]$ — множество формальных параметров и глобальных переменных, явно определяемых в `p` (не включая переменные, определенные в процедурах, вызываемых процедурой `p`). Для того чтобы записать уравнения для $change[p]$, нам надо только свя-

зать глобальные переменные и формальные параметры p , которые используются в качестве фактических параметров в вызовах, выполняемых процедурой p , с соответствующими формальными параметрами вызываемых процедур. Мы можем записать:

$$change[p] = def[p] \cup A \cup G \quad (10.14)$$

где

1. $A = \{a \mid a \text{ — глобальная переменная или формальный параметр } p, \text{ такой, что для некоторой процедуры } q \text{ и целого } i \text{ } p \text{ вызывает } q \text{ с } a \text{ в качестве } i\text{-го действительного параметра и } i\text{-й формальный параметр } q \text{ содержитя в } change[q]\}$.
2. $G = \{g \mid g \text{ — глобальная переменная из } change[q] \text{ и } p \text{ вызывает } q\}$.

Для читателя не должно быть неожиданностью, что уравнение (10.14) может быть решено для множества процедур с применением итеративной методики. Хотя решение не единственное, нас интересует только наименьшее из возможных решений. Обеспечить сходимость к этому решению можно итерациями с выбором заведомо малого начального приближения. Таким очевидным заведомо малым приближением является $change[p] = def[p]$. Детальное рассмотрение процесса итерирования остается читателю в качестве упражнения.

Стоит рассмотреть и порядок, в котором должны обрабатываться различные процедуры в процессе описанных выше итераций. Например, если процедуры не взаимно рекурсивны, то сначала мы можем рассмотреть процедуры, которые не вызывают друг друга (должна быть как минимум одна такая процедура). Для этих процедур $change = def$. Далее мы можем вычислить $change$ для тех процедур, которые вызывают только процедуры, не вызывающие в свою очередь других процедур. Для этой группы процедур мы можем применить (10.14) непосредственно, поскольку $change[q]$ будет известно для любого q из (10.14).

Эта идея может быть уточнена следующим образом. Рассмотрим *граф вызовов* (calling graph), узлы которого представляют процедуры, с дугами от p к q , если p вызывает q ¹¹. Набор процедур, не являющихся взаимно рекурсивными, дает ациклический граф вызовов, и в этом случае мы можем посетить каждый узел один раз.

Теперь мы приведем алгоритм для вычисления множества $change$.

Алгоритм 10.13. Межпроцедурный анализ изменяемых переменных

Вход. Набор процедур p_1, p_2, \dots, p_n . Если граф вызовов ациклический, мы полагаем, что p_i вызывает p_j , только если $j < i$. В противном случае мы не делаем предположений о том, какие процедуры вызывают другие.

Выход. Для каждой процедуры p мы получаем множество $change[p]$ глобальных переменных и формальных параметров p , которые могут быть явно изменены процедурой p без использования псевдонимов.

Метод.

1. Вычислим множество $def[p]$ путем просмотра для каждой процедуры p .
2. Выполним программу, приведенную на рис. 10.43, для вычисления $change$. □

¹¹ Мы полагаем, что у нас нет процедурных переменных. Допущение таких переменных усложнит построение графа вызовов, поскольку одновременно с построением дуг графа вызовов нам придется определить возможные фактические значения, соответствующие формальным параметрам процедурного типа.

```

/* Инициализация множеств */
(1) for Каждая процедура p do  $change[p] := def[p]$ ;
(2) while Имеются изменения,
    вносимые в любой из  $change[p]$  do
(3)   for i := 1 to n do
(4)     for Каждая процедура q, вызываемая  $p_i$  do begin
(5)       Добавить все глобальные переменные из
             $change[q]$  в  $change[p_i]$ ;
(6)       for Каждый j-й формальный параметр x
            процедуры q do
(7)         if  $x \in change[q]$  then
(8)           for Каждый вызов q процедурой  $p_i$  do
(9)             if a, фактический j-й
                параметр вызова является
                глобальной переменной или
                формальным параметром  $p_i$ 
                then Добавить a к  $change[p_i]$ 
(10)
        end

```

Рис. 10.43. Итеративный алгоритм вычисления множества change

Пример 10.29

Рассмотрим рис. 10.42 еще раз. Путем просмотра определяем, что $def[main] = \{g\}$, $def[one] = \{x\}$ и $def[two] = \{h\}$. Это и есть начальные значения множества $change$. Граф вызовов процедур представлен на рис. 10.44. Мы будем рассматривать two, one, main в том порядке, в котором мы посещаем эти процедуры.

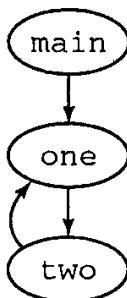


Рис. 10.44. Граф вызовов

Рассмотрим $p_i = two$ в программе на рис. 10.42. Тогда в строке (4) q может быть только процедурой one. Поскольку изначально $change[one] = \{x\}$, в строке (5) к $change[two]$ ничего не добавляется. В строках (6) и (7) мы должны рассмотреть только второй формальный параметр процедуры one, поскольку первый фактический параметр локален по отношению к two. В единственном вызове процедуры one процедурой two второй фактический параметр у и соответствующий ему формальный параметр x изменяются, так что в строке (10) множество $change[two]$ становится равным $\{h, y\}$.

Теперь рассмотрим $p_i = one$. В строке (4) q может быть только процедурой two. В строке (5) h в $change[two]$ представляет собой глобальную переменную, так что мы определяем $change[one] = \{h, x\}$. В строках (6) и (7) в $change[two]$ находится только первый формальный параметр two, так что в строке (10) мы должны добавить к

change[one] переменные *g* и *w*, которые являются первыми фактическими параметрами в вызовах процедуры *two*. Следовательно, *change[one]* = {*g*, *h*, *w*, *x*}.

Рассмотрим теперь процедуру *main*. Процедура *one* изменяет оба свои формальные параметры, так что и *h*, и *i* будут изменены в результате вызова процедуры *one* процедурой *main*. Однако переменная *i* является локальной и не должна рассматриваться. Таким образом, *change[main]* = {*g*, *h*}. И наконец, мы повторяем цикл *while* в строке (2). При повторном рассмотрении *two* мы находим, что *one* изменяет глобальную переменную *g*. Таким образом, вызов *one(k, y)* изменяет *g*, и *change[two]* = {*g, h, y*} . Поступающие итерации не вносят никаких новых изменений. □

Использование информации об изменениях

В качестве примера способа использования множества *change* рассмотрим вычисление глобальных общих подвыражений. Предположим, что мы вычисляем доступные выражения для процедуры *p* и хотим вычислить *a_kill[B]* для блока *B*. Следует считать, что определение переменной *a* уничтожает любое выражение, содержащее *a* или некоторое *x*, которое может быть псевдонимом *a*. Однако вызов в блоке *B* процедуры *q* не может уничтожить выражение, включающее *a*, если только *a* не является псевдонимом (напомним, что *a* является псевдонимом самого себя) некоторой переменной из *change[q]*. Таким образом, информация, вычисленная по алгоритмам 10.12 и 10.13, может использоваться для построения безопасного приближения множества уничтоженных выражений.

Для вычисления доступных выражений в программах с вызовами процедур мы также должны использовать консервативную оценку множества выражений, генерируемых вызовом процедуры. Для консервативности можно считать, что *a+b* генерируется вызовом процедуры *q* тогда и только тогда, когда на каждом пути от входа в *q* до выхода из нее обнаруживается выражение *a+b* без последующего переопределения *a* или *b*. При поиске вхождений выражения *a+b* мы не должны воспринимать выражение *x+y* как такое вхождение, если только мы не уверены, что при любом вызове *q* *x* является псевдонимом *a*, а *y* — псевдонимом *b*.

Описанное требование вытекает из того, что уменьшение множества доступных выражений является консервативной ошибкой, т.е. допустимо не включить в это множество некоторое действительно доступное выражение, но не наоборот. По той же причине мы должны считать, что выражение *a+b* уничтожается определением любой переменной *z*, которая может быть псевдонимом *a* или *b*. Таким образом, простейший путь вычисления доступных выражений для всех узлов всех процедур состоит в том, чтобы полагать, что вызовы не генерируют никаких доступных выражений и что *a_kill[B]* для всех блоков *B* вычисляется так, как показано выше. Поскольку не приходится ожидать генерации большого числа выражений в типичной процедуре, такой подход вполне применим для большинства случаев.

Более сложный и более точный альтернативный подход к вычислению допустимых выражений состоит в итеративном вычислении *gen[p]* для каждой процедуры *p*. Мы можем инициализировать *gen[p]* множеством выражений, доступных в конце узла выхода из *p*, вычисленным в соответствии с описанным выше методом, т.е. для генерируемых выражений запрещено использование псевдонимов; *a+b* представляет только себя, даже если другие переменные могут быть псевдонимами *a* и *b*.

Теперь заново вычислим доступные выражения для всех узлов всех процедур. Однако следует учесть, что вызов $q(a,b)$ генерирует такие выражения в $gen[q]$ с a и b , подставленными вместо соответствующих формальных параметров q . Множество a_kill остается тем же, что и ранее. Новое значение $gen[p]$ для каждой процедуры p может быть найдено путем поиска выражений, доступных в конце блока выхода из p . Эта итерация может повторяться до тех пор, пока мы не получим все доступные выражения во всех узлах.

10.9. Анализ потока данных структурированных графов потока

Программы, не использующие операторы безусловного перехода, имеют приводимые графы потока; написание таких программ приветствуется рядом методологий программирования. Исследования больших классов программ показали, что в сущности все программы, написанные людьми, имеют приводимые графы потока¹². Это наблюдение может использоваться при оптимизации, поскольку для приводимых графов потока могут быть найдены существенно более быстрые алгоритмы оптимизации. В этом разделе мы рассмотрим ряд концепций, связанных с графиками потока, например “анализ интервалов”, который в первую очередь относится к структурированным графикам потока. В сущности, мы применим синтаксически управляемые технологии, разработанные в разделе 10.5, к более общему случаю, когда синтаксис не обязательно обеспечивает структуру, но это делает график потока.

Поиск вглубь

Имеется практический способ упорядочения узлов графа потока, известный как упорядочение вглубь (depth-first ordering), являющийся обобщением обхода дерева вглубь, рассмотренного в разделе 2.3. Упорядочение вглубь может использоваться для поиска циклов в любом графике потока, а кроме того, помогает ускорить итеративные алгоритмы потока данных, типа рассмотренных в разделе 10.6. Упорядочение вглубь начинается в начальном узле и выполняет исследование всего графа, пытаясь посетить узлы, наиболее удаленные от начального, как можно раньше (вглубь). Маршрут такого исследования образует дерево. Перед тем как привести формальный алгоритм, рассмотрим пример.

Пример 10.30

Один из возможных способов обхода вглубь графа потока, показанного на рис. 10.45, приведен на рис. 10.46. Дуги, показанные сплошными линиями, образуют дерево; пунктирные линии соответствуют прочим дугам графа потока. Просмотр вглубь графа потока соответствует обходу дерева в прямом порядке $1 \rightarrow 3 \rightarrow 4 \rightarrow 6 \rightarrow 7 \rightarrow 8 \rightarrow 10$, затем обратно в 8, после чего — в 9. Затем мы еще раз возвращаемся в узел 8, проходим назад по узлам 7, 6 и 4 и входим в узел 5. Из узла 5 мы возвращаемся в узлы 4, 3 и 1. После этого мы заходим в узел 2, возвращаемся в узел 1, и тем самым дерево оказывается пройденным. Правда, пока мы не пояснили, как это дерево выбирается на основе графа потока. □

¹² “Написанные людьми” — отнюдь не лишнее дополнение, так как известен ряд программ, генерирующих код, напичканный безусловными переходами. Ничего ошибочного в этом нет, структура содержится во входных данных для такой программы.

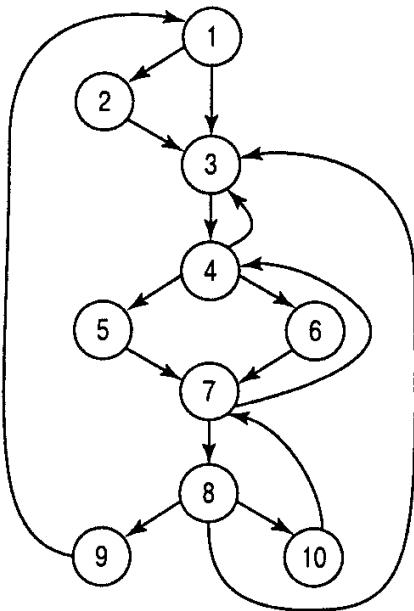


Рис. 10.45. Граф потока

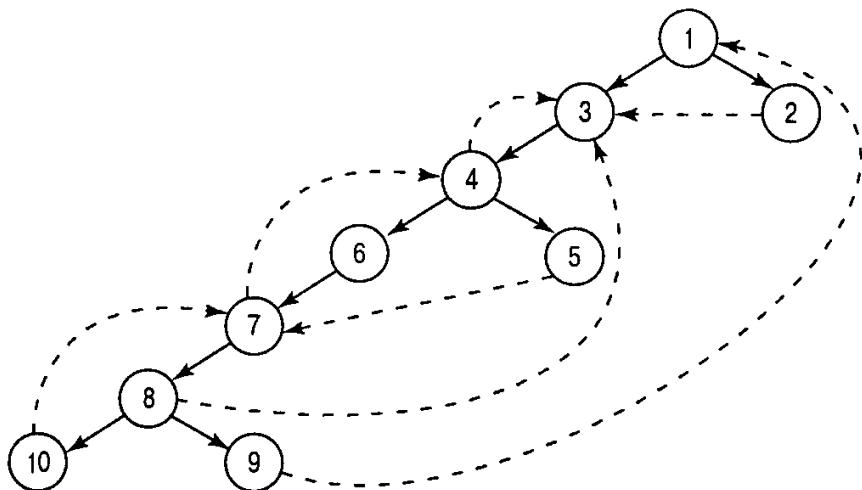


Рис. 10.46. Представление вглубь

Упорядочение вглубь представляет собой обращение порядка, в котором мы посещаем узлы при обходе дерева в прямом порядке.

Пример 10.31

В примере 10.30 полная последовательность узлов, посещенных при обходе дерева, имеет вид

1, 3, 4, 6, 7, 8, 10, 8, 9, 8, 7, 6, 4, 5, 4, 3, 1, 2, 1.

Пометим в этом списке последние появления каждого из узлов.

1, 3, 4, 6, 7, 8, 10, 8, 9, 8, 7, 6, 4, 5, 4, 3, 1, 2, 1

Упорядочение вглубь представляет собой последовательность помеченных номеров в обратном порядке. В нашем случае это последовательность 1, 2, ..., 10, т.е. узлы были изначально пронумерованы в порядке вглубь. □

Приведем теперь алгоритм, который вычисляет упорядочение вглубь графа потока путем построения и обхода дерева с корнем в начальном узле, пытаясь сделать пути в дереве по возможности более длинными. Такое дерево называется *охватывающим*.

вглубь, или *остовным вглубь деревом* (depth-first spanning tree, сокращенно *dfs*). Именно этот алгоритм был использован для построения дерева на рис. 10.46 по графу на рис. 10.45.

Алгоритм 10.14. Охватывающее вглубь дерево и упорядочение вглубь

Вход. Граф потока G .

Выход. Охватывающее вглубь дерево T графа потока G , а также упорядочение узлов графа G .

Метод. Мы используем рекурсивную процедуру $search(n)$, представленную на рис. 10.47. Алгоритм сначала инициализирует все узлы G как “непосещенные”, а затем вызывает $search(n_0)$, где n_0 — начальный узел. При вызове $search(n)$ мы сначала помечаем узел n как “посещенный”, чтобы избежать повторного добавления узла n в дерево. Переменную i мы используем как счетчик, принимающий значения от количества узлов G до 1, назначая при обходе вглубь номера $dfn[n]$ узлам n . Множество дуг T образует охватывающее вглубь дерево графа G , и эти дуги называются *дугами дерева*. \square

```

procedure search(n);
begin
(1)      Пометить  $n$  как посещенный узел
(2)      for Все преемники  $s$  узла  $n$  do
(3)          if Узел  $s$  непосещенный then begin
(4)              Добавить в  $T$  дугу  $n \rightarrow s$ 
(5)              search(s)
(6)          end;
(7)      dfn[n] := i;
(8)      i := i - 1
end;
/* Далее следует основная программа */
(9)      T :=  $\emptyset$  /* Множество дуг */
(10)     for Каждый узел  $n \in G$  do
            Пометить  $n$  как "непосещенный"
(11)     i := Количество узлов в  $G$ ;
search( $n_0$ )

```

Рис. 10.47. Алгоритм просмотра вглубь

Пример 10.32

Рассмотрим рис. 10.47. Мы устанавливаем i равным 10 и вызываем $search(1)$. В строке (2) процедуры $search$ мы должны рассмотреть всех потомков узла 1. Предположим, что первым мы рассматриваем узел $s = 3$. Тогда мы добавляем в дерево T дугу $1 \rightarrow 3$ и вызываем $search(3)$. В $search(3)$ к дереву T мы добавляем дугу $3 \rightarrow 4$ и вызываем $search(4)$.

Предположим, что в $search(4)$ мы сначала выбрали $s = 6$. В таком случае мы добавляем к дереву T дугу $4 \rightarrow 6$ и вызываем $search(6)$. Это приводит нас к добавлению в дерево дуги $6 \rightarrow 7$ и вызову $search(7)$. У узла 7 два потомка — 4 и 8. Однако узел 4 уже помечен как “посещенный” вызовом $search(4)$, так что при $s = 4$ мы не выполняем никаких действий. При $s = 8$ мы добавляем в дерево дугу $7 \rightarrow 8$ и вызываем $search(8)$. Предположим, что после этого нами выбран узел $s = 10$. Мы добавляем к дереву дугу $8 \rightarrow 10$ и вызываем $search(10)$.

Потомок узла 10 — узел 7 уже посещен, так что мы переходим к строке (6) на рис. 10.47 и выполняем присвоения $dfn[10] = 10$ и $i = 9$. Этим завершается вызов

$\text{search}(10)$, и мы возвращаемся в $\text{search}(8)$. Здесь мы рассматриваем $s = 9$, добавляя при этом к дереву T дугу $8 \rightarrow 9$ и вызывая $\text{search}(9)$. Так как единственный потомок узла 9 — узел 1 уже посещен, мы присваиваем $\text{dfn}[9] = 9$ и $i = 8$, после чего возвращаемся в $\text{search}(8)$. Последний потомок узла 8 , узел 3 уже помечен как посещенный, так что для $s = 3$ мы не предпринимаем никаких действий. Теперь, когда мы рассмотрели всех потомков 8 , присвоим $\text{dfn}[8] = 8$ и $i = 7$ и вернемся в $\text{search}(7)$.

Все потомки узла 7 уже рассмотрены, так что мы присваиваем $\text{dfn}[7] = 7$ и $i = 6$ и возвращаемся в $\text{search}(6)$. Аналогично, поскольку рассмотрены все потомки узла 6 , присвоим $\text{dfn}[6] = 6$ и $i = 5$ и вернемся в $\text{search}(4)$. Потомок 3 узла 4 уже посещен, но потомок 5 — еще нет, так что к дереву добавляется дуга $4 \rightarrow 5$ и вызывается $\text{search}(5)$. Поскольку потомок узла 5 узел 7 уже посещен, вызов сводится к присвоениям $\text{dfn}[5] = 5$ и $i = 4$, и мы возвращаемся в $\text{search}(4)$. Так как рассмотрение потомков узла 4 выполнено полностью, мы присваиваем $\text{dfn}[4] = 4$ и $i = 3$ и возвращаемся в $\text{search}(3)$, где присваиваем $\text{dfn}[3] = 3$ и $i = 2$ и возвращаемся в $\text{search}(1)$.

Последние шаги алгоритма состоят в вызове $\text{search}(2)$ из $\text{search}(1)$, присвоении $\text{dfn}[2] = 2$ и $i = 1$, возврате в $\text{search}(1)$ и присвоении $\text{dfn}[1] = 1$ и $i = 0$. Заметим, что мы выбрали нумерацию узлов такую, что выполняется соотношение $\text{dfn}[i] = i$, но для произвольного графа это соотношение выполнятся не обязано (как не будет выполняться и при другом упорядочении вглубь графа на рис. 10.45).

Дуги представления графа потока вглубь

При построении охватывающего вглубь дерева для графа потока дуги этого графа разбиваются на три категории.

1. Дуги, которые идут от узла m к предшественнику узла m в дереве (возможно, к самому узлу m). Такие дуги мы будем называть *отступающими* (retreating). Например, отступающими на рис. 10.46 являются дуги $7 \rightarrow 4$ и $9 \rightarrow 1$. Интересен и имеет практическую ценность тот факт, что если график потока приводим, то отступающие дуги в точности представляют собой обратные дуги графа потока¹³, независимо от порядка, в котором на шаге (2) на рис. 10.47 посещаются потомки узла. Для любого графа потока каждая обратная дуга является отступающей, хотя если график неприводим, в нем могут быть некоторые отступающие дуги, не являющиеся обратными.
2. *Наступающими* (advancing) называются дуги, идущие от узла m к истинному потомку m в дереве (не совпадающему с m). Все дуги охватывающего вглубь дерева являются наступающими. На рис. 10.46 других наступающих дуг нет, хотя, например, если бы имелась дуга $4 \rightarrow 8$, то она принадлежала бы этой категории.
3. Существуют дуги $m \rightarrow n$, такие, что ни m , ни n не являются родительскими по отношению друг к другу в охватывающем вглубь дереве. Примерами таких дуг на рис. 10.46 являются $2 \rightarrow 3$ и $5 \rightarrow 7$. Мы называем их *поперечными* (cross) дугами. Важное свойство поперечных дуг заключается в том, что если мы изобразим охватывающее вглубь дерево так, чтобы потомки узла изображались слева направо в порядке их добавления в дерево, то все поперечные дуги будут идти справа налево.

Следует отметить, что дуга $m \rightarrow n$ является отступающей тогда и только тогда, когда $\text{dfn}[m] \geq \text{dfn}[n]$. Чтобы увидеть, почему это так, заметим, что если m является потомком n в охватывающем вглубь дереве, то $\text{search}(m)$ прекращает работу до $\text{search}(n)$, так что

¹³ Напомним, что обратными дугами графа потока являются дуги, головы которых доминируют над их хвостами.

$dfn[m] \geq dfn[n]$. И наоборот, если $dfn[m] \geq dfn[n]$, то $search(m)$ прекращает работу до $search(n)$ либо $m = n$. Однако если существует дуга $m \rightarrow n$, $search(n)$ должно начать работу до $search(m)$, иначе тот факт, что n является преемником m , должен привести к тому, что n будет потомком m в охватывающем вглубь дереве. Следовательно, время активности $search(m)$ является подынтервалом времени активности $search(n)$, откуда вытекает, что n является предком узла m в охватывающем вглубь дереве.

Глубина графа потока

Важным параметром графа потока является его *глубина*. Для данного охватывающего вглубь дерева для графа глубина представляет собой наибольшее количество отступающих дуг на любом ациклическом пути.

Пример 10.33

На рис. 10.46 глубина равна 3, поскольку есть путь $10 \rightarrow 7 \rightarrow 4 \rightarrow 3$ с тремя отступающими дугами, но нет ациклического пути с четырьмя или более отступающими дугами. То, что “глубочайший” путь содержит только отступающие дуги, не более чем случайное совпадение; в общем случае на наиболее глубоком пути у нас может быть смесь отступающих, наступающих и поперечных дуг. \square

Можно доказать, что глубина никогда не превышает того, что можно интуитивно назвать глубиной вложенности циклов графа потока. Если график потока приводим, мы можем заменить “отступающие” на “обратные” в определении глубины, поскольку отступающие дуги любого охватывающего вглубь дерева являются в точности обратными дугами. Понятие глубины при этом становится независимым от реально выбранного охватывающего вглубь дерева.

Интервалы

Разделение графа потока на интервалы служит для придания ему иерархической структуры. Такая структура, в свою очередь, позволяет нам применять правила для синтаксически управляемого анализа потока данных (разработка правил началась в разделе 10.5).

Интуитивно интервал в графике потока представляет собой естественный цикл плюс ациклическая структура, “свисающая” из узлов этого цикла. Важным свойством интервалов является то, что они имеют *заголовочные* (header) узлы, доминирующие над всеми узлами интервала, так что каждый интервал представляет собой область. Формально для данного графа потока G с начальным узлом n_0 и узла $n \in G$ *интервал с заголовком* n , обозначаемый как $I(n)$, определяется следующим образом.

1. $n \in I(n)$
2. Если все предшественники некоторого узла $m \neq n_0$ принадлежат $I(n)$, то $m \in I(n)$.
3. Кроме этого, в $I(n)$ нет других узлов.

Таким образом, мы можем построить $I(n)$, начиная с n и добавляя узлы m в соответствии с правилом (2). Порядок, в котором мы добавляем два кандидата m , значения не имеет, поскольку когда предшественники узла попадают в $I(n)$, они остаются в интервале, и каждый кандидат в конечном счете оказывается добавленным согласно правилу (2). Когда в конце концов новых кандидатов для добавления в $I(n)$ больше нет, полученное множество представляет собой интервал с заголовком n .

Разбиение на интервалы

Для данного графа потока G мы можем провести процедуру разбивки его на непересекающиеся интервалы с помощью следующего алгоритма.

Алгоритм 10.15. Анализ интервалов графа потока

Вход. Граф потока G с начальным узлом n_0 .

Выход. Разбиение G на множество непересекающихся интервалов.

Метод. Для любого узла n мы вычисляем $I(n)$ с помощью метода, набросок которого приведен ниже.

```
I(n) := { n };  
while Существует узел  $m \neq n_0$ , все предшественники  
которого содержатся в  $I(n)$  do  
    I(n) := I(n) ∪ { m }
```

Узлы, представляющие собой заголовки интервалов разбиения, выбираются следующим образом (изначально "избранных" узлов нет).

Построить $I(n_0)$ и "избрать" все узлы в этом интервале;

while Имеется узел m , который еще не "избран", но имеет
"избранного" предшественника do

Построить $I(m)$ и "избрать" все узлы в этом интервале.

□

Как только кандидат m имеет избранного предшественника p , m никогда не будет добавлен к некоторому интервалу, не содержащему p . Таким образом, узлы-кандидаты m остаются кандидатами до тех пор, пока они не будут избраны для того, чтобы возглавить свои собственные интервалы. Следовательно, порядок, в котором выбираются заголовки интервалов в алгоритме 10.15, не влияет на конечную разбивку графа на интервалы. Кроме того, при условии, что все узлы достижимы из n_0 , индукцией по длине пути от n_0 к узлу n можно показать, что n в итоге либо будет помещен в интервал какого-то другого узла, либо возглавит собственный интервал (но не то и другое одновременно). Таким образом, множество построенных по алгоритму 10.15 интервалов представляет собой истинное разбиение G .

Пример 10.34

Найдем разбиение на интервалы графа, представленного на рис. 10.45. Мы начинаем с построения $I(1)$, поскольку узел 1 является начальным. Мы можем добавить к $I(1)$ узел 2, поскольку единственным предшественником узла 2 является узел 1. Однако мы не можем добавить к $I(1)$ узел 3, поскольку его предшественники 4 и 8 еще не вошли в $I(1)$. Аналогично все прочие узлы, кроме 1 и 2, имеют предшественников, которые пока что не вошли в $I(1)$. Следовательно, $I(1) = \{1, 2\}$.

Теперь мы можем приступить к вычислению $I(3)$, так как узел 3 имеет "избранных" предшественников 1 и 2, но сам в интервал не входит. Однако к интервалу узла 3 не получается добавить ни один узел, так что $I(3) = \{3\}$. Теперь заголовком становится узел 4, так как его предшественник 3 находится в интервале. К $I(4)$ можно добавить узлы 5 и 6, так как их предшественником является только узел 4, принадлежащий интервалу. Другие узлы к $I(4)$ добавить невозможно, так как они имеют и других, не вошедших в интервал предшественников (например, таким предшественником узла 7 является узел 10).

Далее заголовком становится узел 7, к которому можно добавить узел 8, а затем и 9 и 10, так как их единственным предшественником является узел 8. Следовательно, интервалами в разбиении графа на рис. 10.45 являются следующие множества:

$$\begin{array}{ll} I(1) = \{1, 2\} & I(4) = \{4, 5, 6\} \\ I(3) = \{3\} & I(7) = \{7, 8, 9, 10\} \end{array}$$

Графы интервалов

На основе интервалов одного графа потока G мы можем построить новый граф потока $I(G)$ согласно следующим правилам.

1. Узлы $I(G)$ соответствуют интервалам, полученным в результате разбиения G .
2. Начальный узел $I(G)$ представляет собой интервал, содержащий начальный узел G .
3. Дуга от интервала I к некоторому другому интервалу J ведет тогда и только тогда, когда в G есть дуга от некоторого узла из I к заголовку J . Заметим, что дуги, входящей в узел i интервала J , отличающейся от его заголовка, не может быть, поскольку в таком случае i не мог бы быть добавлен в интервал J в алгоритме 10.15.

Мы можем поочередно применять алгоритм 10.15 и построение графа интервалов, получая последовательность графов G , $I(G)$, $I(I(G))$, $I(I(I(G)))$, В конечном счете мы приедем к графу, узлы которого сами по себе будут являться интервалами. Такой граф называется *пределным графом потока* (*limit flow graph*) G . Интересен тот факт, что граф потока приводим тогда и только тогда, когда его предельный граф потока представляет собой единственный узел¹⁴.

Пример 10.35

На рис. 10.48 показан результат неоднократного применения построения интервалов к графу на рис. 10.45. Интервалы этого графа были найдены в примере 10.34, и построенный на их основе граф интервалов показан на рис. 10.48a. Обратите внимание на то, что дуга $10 \rightarrow 7$ на рис. 10.45 не приводит к появлению дуги из узла, представляющего $\{7, 8, 9, 10\}$ в себя на рис. 10.48a, поскольку построение графа интервалов явно исключает такие циклы. Заметим также, что граф потока на рис. 10.45 приводим, поскольку его предельный граф потока представляет собой единичный узел. \square

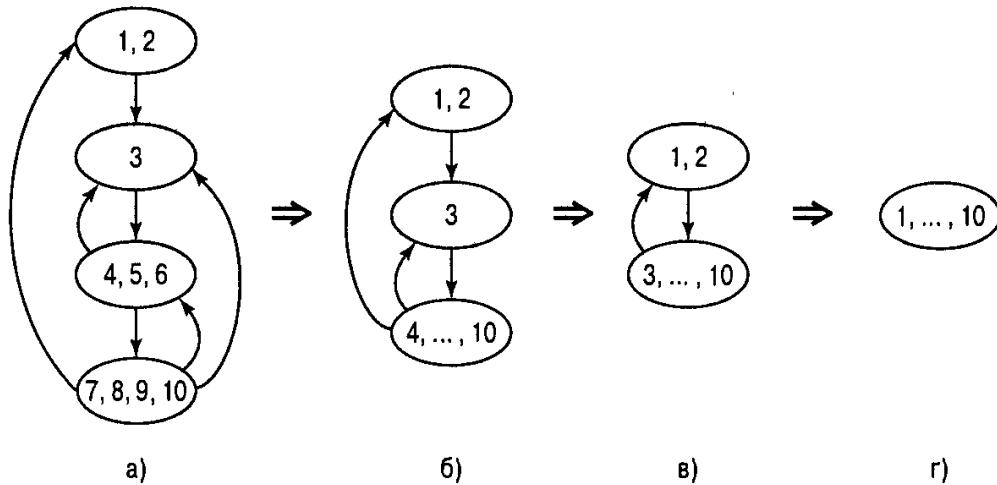


Рис. 10.48. Последовательность графов интервалов

¹⁴ В действительности исторически это и было исходным определением приводимого графа.

Разделение узлов

Если мы достигли предельного графа потока, состоящего более чем из одного узла, мы можем продолжать далее, только если разделим один или несколько узлов. Если узел n имеет k предшественников, мы можем заменить его k узлами n_1, n_2, \dots, n_k . i -й предшественник узла n становится предшественником только n_i , в то время как все преемники n становятся преемниками всех n_i .

Если мы применим алгоритм 10.15 к полученному графу, в котором каждый n_i имеет единственного предшественника, то он гарантированно становится частью интервала этого предшественника. Таким образом, разделение одного узла и один цикл разбиения на интервалы приводят к графу с меньшим количеством узлов. Как следствие, построение графа интервалов с использованием (при необходимости) разделения узлов должно привести в конечном счете к графу, состоящему из одного узла. Значение этого наблюдения станет ясным в следующем разделе, где мы рассмотрим алгоритмы анализа потока данных, которые управляются этими двумя операциями над графиками.

Пример 10.36

Рассмотрим граф потока на рис. 10.49 a , который представляет собственный предельный график потока. Мы можем разбить узел 2 на $2a$ и $2b$ с предшественниками соответственно 1 и 3. Этот график показан на рис. 10.49 b . Если мы дважды применим разбиение на интервалы, то получим последовательность графов, показанную на рис. 10.49 c и рис. 10.49 d , сходящуюся к единственному узлу. \square

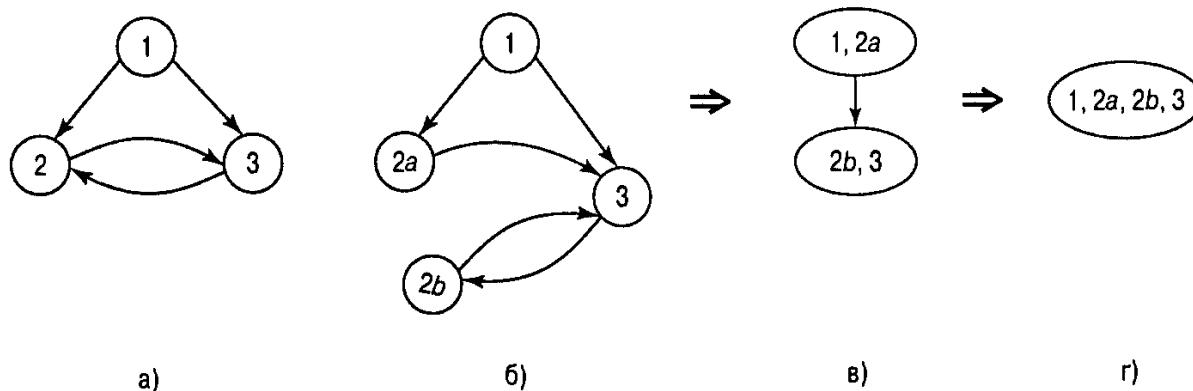


Рис. 10.49. Разделение узлов с последующим разбиением на интервалы

T_1 - T_2 -анализ

Удобным способом получить тот же эффект, что и при анализе интервалов, является применение двух простейших преобразований к графу потока.

- T_1 : Если n — узел с циклом, т.е. дугой $n \rightarrow n$, удалить эту дугу.
- T_2 : Если существует (не начальный) узел n , имеющий единственного предшественника m , то m может *поглотить* (consume) n , удаляя его и делая всех преемников n (включая, возможно, m) преемниками m .

Вот некоторые интересные факты о преобразованиях T_1 и T_2 .

1. Если мы применим T_1 и T_2 к графу потока G в произвольном порядке, пока не получим график потока, к которому невозможно применение этих преобразований,

то в результате будет получен единственный результирующий граф (не зависящий от порядка применения преобразований). Причина этого в том, что кандидат для устранения цикла преобразованием T_1 или поглощения преобразованием T_2 остается кандидатом, даже если сначала выполняются некоторые другие преобразования T_1 и T_2 .

- Граф потока, полученный исчерпывающим применением T_1 и T_2 к графу потока G , дает предельный граф потока G . Доказательство этого факта оставлено читателю в качестве упражнения. Как следствие, другое определение “приводимого графа потока” состоит в том, что он может быть приведен к единственному узлу с помощью преобразований T_1 и T_2 .

Пример 10.37

На рис. 10.50 представлена последовательность преобразований T_1 и T_2 , примененная к графу, представляющему собой переименованный граф на рис. 10.49б. На рис. 10.50б с поглощает d . Заметьте, что цикл вокруг cd на рис. 10.50б получается из-за наличия на рис. 10.50а дуги $d \rightarrow c$. Этот цикл удаляется путем преобразования T_1 на рис. 10.50в. Обратите внимание, что при поглощении узлом a узла b на рис. 10.50г дуги из узлов a и b к узлу cd становятся единой дугой.

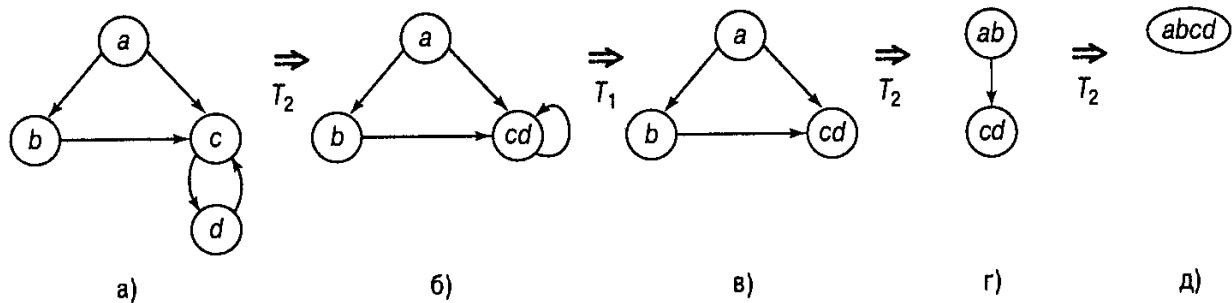


Рис. 10.50. Приведение путем применения преобразований T_1 и T_2

Области

Вспомним, что в разделе 10.5 область в графе потока определялась как множество узлов N , включающее заголовок, доминирующий над всеми другими узлами области. Все дуги между узлами из N находятся в области, за исключением (возможно) некоторых из тех, что входят в заголовок. Например, каждый интервал представляет собой область, однако имеются области, не являющиеся интервалами, поскольку, например, они могут не включать узлы, которые входят в интервал, или не содержать некоторые обратные дуги, ведущие в заголовок. Кроме того, как мы увидим, существуют области, гораздо большие любого интервала.

При приведении графа потока G с помощью преобразований T_1 и T_2 всегда остаются верными следующие условия.

- Узел представляет область в G .
- Дуга от a к b представляет множество дуг. Каждая такая дуга является дугой, исходящей из некоторого узла области, представленной a , в заголовок области, представленной узлом b .
- Каждый узел и дуга в G представлены ровно одним узлом или дугой текущего графа.

Чтобы увидеть, почему выполняются эти условия, заметим вначале, что их выполнение в G тривиально. Каждый узел сам по себе представляет область, и каждая дуга представляет только себя. Предположим, что мы применили преобразование T_1 к некоторому узлу n , представляющему область R , где дуга $n \rightarrow n$ представляет некоторое множество дуг E , в котором все дуги входят в заголовок R . Если мы добавим дуги E в область R , она останется областью, так что после удаления дуги $n \rightarrow n$ узел n представляет область R и дуги из E , что сохраняет выполнимость условий (1)–(3).

Рассмотрим преобразование T_2 , поглощающее узел b узлом a , где a и b представляют соответственно области R и S . Кроме того, пусть E — множество дуг, представленное дугой $a \rightarrow b$. Мы утверждаем, что R, S и E вместе образуют область, заголовком которой является заголовок области R . Для доказательства этого мы должны убедиться, что заголовок R доминирует над всеми узлами S . Если это не так, то должен существовать некоторый путь в заголовок S , который не оканчивается дугой из E . Тогда последняя дуга этого пути должна быть представлена некоторой другой дугой, входящей в b . Однако таких дуг быть не может, иначе мы не могли бы применить преобразование T_2 для поглощения b .

Пример 10.38

Узел, помеченный на рис. 10.50б как cd , представляет область, показанную на рис. 10.51а, которая образована узлами c и d . Обратите внимание, что дуга $d \rightarrow c$ не является частью области; на рис. 10.50б эта дуга представлена циклом вокруг cd . Однако на рис. 10.50в дуга $cd \rightarrow cd$ удалена, и теперь узел cd представляет область, показанную на рис. 10.51б.

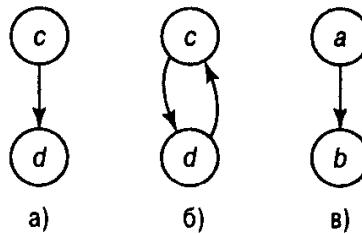


Рис. 10.51. Некоторые области

На рис. 10.50г узел cd все еще представляет область на рис. 10.51б, в то время как узел ab представляет область на рис. 10.51в. Дуга $ab \rightarrow cd$ на рис. 10.50г представляет дуги $a \rightarrow c$ и $b \rightarrow c$ исходного графа потока на рис. 10.50а. После применения преобразования T_2 для получения рис. 10.50д, оставшийся узел будет представлять весь исходный граф потока, представленный на рис. 10.50а. \square

Мы должны заметить, что упомянутое выше свойство приведения T_1 и T_2 сохраняется и для анализа интервалов. Мы оставляем в качестве упражнения тот факт, что при построении $I(G)$, $I(I(G))$, ... каждый узел каждого из этих графов представляет собой область, а каждая дуга — множество дуг, удовлетворяющих приведенному выше свойству (2).

Поиск доминаторов

Мы завершим этот раздел эффективным алгоритмом для этого столь часто используемого нами понятия и продолжим использовать его в дальнейшем развитии теории графов потоков и анализа потока данных. Мы приведем простой алгоритм для вычисле-

я доминаторов каждого узла n графа потока, основанный на том, что если p_1, p_2, \dots, p_k являются предшественниками n и $d \neq n$, то $d \text{ dom } n$ тогда и только тогда, когда $d \text{ dom } p_i$ для каждого i . Этот метод родственен методу прямого анализа потока данных с пересечением в качестве оператора слияния (например, для доступных выражений), при котором мы получаем приближение к множеству доминаторов n и уточняем его неоднократным посещением всех узлов по очереди.

В этом случае выбираемое начальное приближение состоит из начального узла, доминирующего только над начальным узлом, и всех узлов, доминирующих над всеми узлами, кроме начального. Интуитивно обоснование работоспособности такого подхода заключается в том, что кандидаты в доминаторы исключаются только тогда, когда мы находим путь, доказывающий, например, что $m \text{ dom } n$ ложно. Если мы не можем найти такой путь от начального узла в n , обходящий m , то m действительно является доминатором n .

Алгоритм 10.16. Поиск доминаторов

Вход. Граф потока G с множеством узлов N , множество дуг E и начальный узел n_0 .

Выход. Отношение dom .

Метод. Мы вычисляем множество $D(n)$ доминаторов узла n итеративно с применением процедуры, приведенной на рис. 10.52. В конце концов $d \in D(n)$ тогда и только тогда, когда $d \text{ dom } n$. Читатель может самостоятельно добавить детали проверки внесения изменений в множество $D(n)$; в качестве модели может служить алгоритм 10.2.

Можно показать, что $D(n)$, вычисленное в строке (5) на рис. 10.52, всегда представляется собой подмножество текущего $D(n)$. Поскольку $D(n)$ не может уменьшаться бесконечно, в конечном счете выполнение цикла прекратится. Доказательство того, что после сходжения $D(n)$ есть множество доминаторов n , мы оставляем заинтересованному читателю. Алгоритм, приведенный на рис. 10.52, весьма эффективен, поскольку $D(n)$ может быть представлено в виде битового вектора, а операции над множествами в строке (5) могут выполняться с помощью побитовых операций *and* и *or*. \square

```
(1)  $D(n_0) := \{n_0\};$ 
(2) for  $n \in N - \{n_0\}$  do  $D(n) := N;$ 
   /* Окончание инициализации */
(3) while Имеются изменения в  $D(n)$  do
(4)   for  $n \in N - \{n_0\}$  do
(5)      $D(n) := \{n\} \cup \bigcap_{\substack{p-\text{пред-} \\ \text{шественник } n}} D(p);$ 
```

Рис. 10.52. Алгоритм вычисления доминаторов

Пример 10.39

Вернемся к графу потока на рис. 10.45 и предположим, что цикл в строке (4) обходит узлы в порядке их нумерации. Узел 2 имеет в качестве предшественника только узел 1, так что $D(2) := \{2\} \cup D(1)$. Поскольку узел 1 — начальный, в строке (1) $D(1) := \{1\}$ и, таким образом, в строке (5) $D(2) := \{1, 2\}$.

Затем рассматривается узел 3, имеющий предшественников 1, 2, 4 и 8. Страна (5) дает нам $D(3) = \{3\} \cup (\{1\} \cap \{1, 2\} \cap \{1, 2, \dots, 10\}) = \{1, 3\}$. Остальные вычисления дают нам

$$D(4) = \{4\} \cup (D(3) \cap D(7)) = \{4\} \cup (\{1, 3\} \cap \{1, 2, \dots, 10\}) = \{1, 3, 4\}$$

$$\begin{aligned}
 D(5) &= \{5\} \cup D(4) = \{5\} \cup \{1, 3, 4\} = \{1, 3, 4, 5\} \\
 D(6) &= \{6\} \cup D(4) = \{6\} \cup \{1, 3, 4\} = \{1, 3, 4, 6\} \\
 D(7) &= \{7\} \cup (D(5) \cap D(6) \cap D(10)) \\
 &= \{7\} \cup (\{1, 3, 4, 5\} \cap \{1, 3, 4, 6\} \cap \{1, 2, \dots, 10\}) = \{1, 3, 4, 7\} \\
 D(8) &= \{8\} \cup D(7) = \{8\} \cup \{1, 3, 4, 7\} = \{1, 3, 4, 7, 8\} \\
 D(9) &= \{9\} \cup D(8) = \{9\} \cup \{1, 3, 4, 7, 8\} = \{1, 3, 4, 7, 8, 9\} \\
 D(10) &= \{10\} \cup D(8) = \{10\} \cup \{1, 3, 4, 7, 8\} = \{1, 3, 4, 7, 8, 10\}
 \end{aligned}$$

Второй проход по циклу не вносит никаких изменений в полученные множества, так что приведенные выше значения дают нам отношение *dom*. \square

10.10. Эффективные алгоритмы потоков данных

В этом разделе мы рассмотрим два способа использования теории графов потоков для ускорения анализа потока данных. Первый способ представляет собой применение упорядочения вглубь для уменьшения количества проходов итеративных алгоритмов из раздела 10.6, а второй — использование интервалов или преобразований T_1 и T_2 для обобщения синтаксически управляемого подхода из раздела 10.5.

Упорядочение вглубь в итеративных алгоритмах

Во всех рассмотренных задачах, таких как достигающие определения, доступные выражения или активные переменные, любое важное событие в узле распространяется к нему по нециклическим путям. Например, если определение d принадлежит $in[B]$, то имеется некоторый нециклический путь от блока, содержащего d , к блоку B , такой, что d содержится во всех множествах in и out вдоль этого пути. Аналогично, если выражение $x+y$ не доступно при входе в блок B , то имеется некоторый нециклический путь, который обнаруживает этот факт; либо путь идет от начального узла и не включает инструкции, которые уничтожают или генерируют $x+y$, либо путь выходит из блока, который уничтожает $x+y$, и вдоль этого пути нет последующих генераций $x+y$. И наконец, для активных переменных, если x активна на выходе из блока B , то имеется нециклический путь от B к использованию x , на котором нет определений x .

В каждом из этих случаев читатель может убедиться, что пути с циклами ничего не привносят. Например, если использование x достигнуто от конца блока B вдоль пути с циклом, мы можем устраниТЬ этот цикл для поиска более короткого пути, вдоль которого использование x достигается из B .

Если вся полезная информация распространяется вдоль нециклического пути, у нас имеется возможность в итеративных алгоритмах потока данных изменить порядок посещения узлов таким образом, что после относительно меньшего количества проходов по узлам мы можем быть уверены, что информация прошла по всем нециклическим узлам. В частности, статистика, собранная в [256], показывает, что типичные графы потоков имеют очень малую *интервальную глубину*, которая представляет собой количество применений разбиения на интервалы до получения предельного графа потока; найдено, что среднее количество разбиений — 2.75. Кроме того, можно показать, что интервальная глубина графа потока никогда не меньше того, что мы называем “глубиной”, т.е. макси-

мального числа отступающих дуг на любом нециклическом пути (если граф потока неприводим, то глубина может зависеть от того, какое охватывающее вглубь дерево выбрано).

Вспоминая наше обсуждение охватывающих вглубь деревьев в предыдущем разделе, заметим, что если $a \rightarrow b$ — дуга, то номер b “вглубь” меньше номера a , только если дуга является отступающей. Итак, заменим строку (5) на рис. 10.26, предписывающую нам посетить каждый блок B в графе потока, для которого мы вычисляем достигающие определения, следующей строкой:

```
for Каждый блок  $B$  в порядке “вглубь” do
```

Предположим, что у нас есть путь распространения определения d , такой, как

$$3 \rightarrow 5 \rightarrow 19 \rightarrow 35 \rightarrow 16 \rightarrow 23 \rightarrow 45 \rightarrow 4 \rightarrow 10 \rightarrow 17$$

где целые числа представляют номера “вглубь” блоков вдоль пути. Тогда при первом проходе цикла в строках (5)–(9) на рис. 10.26 d будет распространяться от $out[3]$ к $in[5]$, затем к $out[5]$ и так до $out[35]$. В этом проходе определение не достигнет $in[16]$, поскольку 16 предшествует 35, и мы уже вычислили $in[16]$ к тому времени, когда d было помещено в $out[35]$. Однако в следующем выполнении цикла (5)–(9), когда мы вычисляем $in[16]$, d будет включено в него, так как уже есть в $out[35]$. Определение d распространится также в $out[16]$, $in[23]$ и так до $out[45]$, где оно должно подождать следующего прохода, поскольку $in[4]$ уже вычислено. При третьем проходе d доходит до $in[4]$, $out[4]$, $in[10]$, $out[10]$ и $in[17]$, так что после трех проходов мы определили, что d достигает блока 17¹⁵.

Из этого примера несложно выделить общий принцип. Если на рис. 10.26 мы используем порядок “вглубь”, то количество проходов, необходимое для распространения любого достигающего определения вдоль любого нециклического пути, не более чем на единицу превышает число дуг пути, идущих от блока с большим номером к блоку с меньшим. Эти дуги являются отступающими, так что необходимое количество проходов на единицу больше глубины. Конечно, алгоритму 10.2, для того чтобы определить достижение всех определений, требуется один дополнительный проход, так что верхняя граница количества проходов, необходимых алгоритму с упорядочением блоков вглубь, в действительности на 2 превышает глубину, так что, если верить [256], типичное значение равно 5.

Порядок “вглубь” выгоден и для работы с доступными выражениями (алгоритм 10.3), и в случае любых задач потока данных, которые мы решали распространением в прямом направлении. Для задач типа активных переменных, где распространение происходит в обратном направлении, то же среднее значение, равное пяти проходам, достигается при обращении нумерации вглубь. Таким образом, мы можем распространить использование переменной в блоке 17 в обратном направлении по пути

$$3 \rightarrow 5 \rightarrow 19 \rightarrow 35 \rightarrow 16 \rightarrow 23 \rightarrow 45 \rightarrow 4 \rightarrow 10 \rightarrow 17$$

за один проход до $in[4]$, где мы должны дождаться следующего прохода, чтобы достичь $out[45]$. При втором проходе мы достигаем $in[16]$, а на третьем проходим от $out[35]$ до $out[3]$. В общем случае, если мы выберем порядок посещения узлов при проходе, обратный нумерации вглубь, то в связи с тем, что распространение в данной ситуации по уменьшающейся последовательности номеров выполняется за один проход, нам будет достаточно количества проходов, на единицу превышающее глубину.

¹⁵ Определение d достигает также $out[17]$, но это не имеет отношения к рассматриваемому пути.

Анализ потока данных, основанный на структуре

При несколько больших усилиях мы можем реализовать алгоритмы потока данных, которые посещают узлы (и используют уравнения потока данных) не большее количество раз, чем значение интервальной глубины графа потока, а зачастую позволяют обойтись еще меньшим количеством посещений. Хотя не доказано, что дополнительные усилия обязательно приведут к экономии времени работы, эти технологии, основанные на анализе интервалов, используются в ряде компиляторов. Кроме того, идеи, приведенные здесь, применимы к синтаксически управляемым алгоритмам потоков данных для всех типов структурированных управляющих конструкций, а не только для `if...then` и `while...do`, рассмотренных в разделе 10.5, и также присутствуют в некоторых компиляторах.

Наш алгоритм будет основываться на структурах, порожденных из графа потока преобразованиями T_1 и T_2 . Как и в разделе 10.5, мы имеем дело с определениями, которые генерируются и уничтожаются при прохождении управления по области. В отличие от областей, определенных инструкциями `if` или `while`, область в общем случае имеет несколько выходов, так что для каждого блока B в области R мы вычислим множества $gen_{R,B}$ и $kill_{R,B}$ соответственно генерируемых и уничтожаемых определений вдоль путей в области от заголовка до конца блока B . Эти множества будут использованы для определения функции *передачи* $trans_{R,B}(S)$, которая для любого множества определений S говорит о том, какое множество определений достигает конца блока B по путям, полностью лежащим в R , когда все определения из S (и только они) достигают заголовка R .

Как мы видели в разделах 10.5 и 10.6, определения, достигающие конца блока B , разделяются на два класса.

1. Определения, которые генерируются в R и распространяются до конца B независимо от S .
2. Определения, не генерируемые в R , но и не уничтожаемые по некоторому пути от заголовка R к концу B , и, следовательно, находящиеся в $trans_{R,B}(S)$ тогда и только тогда, когда они находятся в S .

Таким образом, мы можем записать *trans* в виде

$$trans_{R,B}(S) = gen_{R,B} \cup (S - kill_{R,B})$$

Сердце алгоритма представляет собой способ вычисления $trans_{R,B}$ для постепенно возрастающих областей, определяемых некоторой (T_1, T_2) -декомпозицией графа потока. В настоящий момент мы полагаем, что график потока приводим, хотя простая модификация позволяет алгоритму работать и с неприводимыми графиками.

Основой является область, состоящая из одного блока B . Функция передачи области в этом случае представляет собой функцию передачи блока, поскольку определение достигает конца блока тогда и только тогда, когда оно либо генерируется блоком, либо находится во множестве S и не уничтожается. Таким образом,

$$\begin{aligned} gen_{B,B} &= gen[B] \\ kill_{B,B} &= kill[B] \end{aligned}$$

Теперь рассмотрим построение области R путем преобразования T_2 , т.е. R образуется, когда R_1 поглощает R_2 , как предлагается на рис. 10.53. Заметим, что в области R нет дуг от R_2 к R_1 , поскольку любая дуга от R_2 к заголовку R_1 не является частью R . Таким образом, любой путь, полностью находящийся в R , проходит (необязательно) через R_2 ,

но не может затем вернуться в R_1 . Заметим также, что заголовок R представляет собой заголовок R_1 . Мы можем заключить, что внутри R область R_2 не влияет на функцию передачи узлов в R_1 , т.е.

$$\begin{aligned} \text{gen}_{R,B} &= \text{gen}_{R_1,B} \\ \text{kill}_{R,B} &= \text{kill}_{R_1,B} \end{aligned}$$

для всех $B \in R_1$.

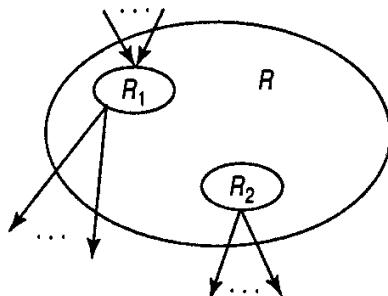


Рис. 10.53. Построение области путем преобразования T_2

Для $B \in R_2$ определение может достичь конца B , если выполняется любое из следующих условий.

1. Определение генерируется в R_2 .
2. Определение генерируется в R_1 , достигает конца некоторого предшественника заголовка R_2 и не уничтожается при переходе от заголовка R_2 к B .
3. Определение принадлежит множеству S , доступному в заголовке R_1 , и не уничтожается ни при переходе к некоторому предшественнику заголовка R_2 , ни при переходе от заголовка R_2 к B .

Следовательно, определения, достигающие концов тех блоков в R_1 , которые являются предшественниками заголовка R_2 , играют особую роль. По сути, мы видим, что происходит с множеством S , входящим в заголовок R_1 , когда его определения пытаются достичь заголовка R_2 через один из его предшественников. Множество определений, достигающих одного из предшественников заголовка R_2 , становится входным для R_2 , и мы применяем к нему функцию передачи для R_2 .

Таким образом, пусть G представляет собой объединение $\text{gen}_{R_1,P}$ для всех предшественников P заголовка R_2 и пусть K — пересечение $\text{kill}_{R_1,P}$ для всех этих предшественников. Тогда если S является множеством определений, достигающим заголовка R_1 , то множество определений, которые достигают заголовка R_2 по путям, полностью расположенным в R , равно $G \cup (S - K)$. Следовательно, функция передачи в R для блоков B из R_2 может быть вычислена с помощью следующих уравнений.

$$\begin{aligned} \text{gen}_{R,B} &= \text{gen}_{R_2,B} \cup (G - \text{kill}_{R_2,B}) \\ \text{kill}_{R,B} &= \text{kill}_{R_2,B} \cup (K - \text{gen}_{R_2,B}) \end{aligned}$$

Рассмотрим теперь, что происходит, когда область R строится из R_1 путем преобразования T_1 . Общая ситуация показана на рис. 10.54; заметим, что R состоит из R_1 и некоторого количества обратных дуг, идущих к заголовку R_1 (который, конечно, является одновременно и заголовком R). Путь, проходящий через заголовок дважды, является циклическим и, как мы уже выяснили, не должен рассматриваться. Таким образом, все определения, генерируемые в конце блока B , генерируются одним из двух способов.

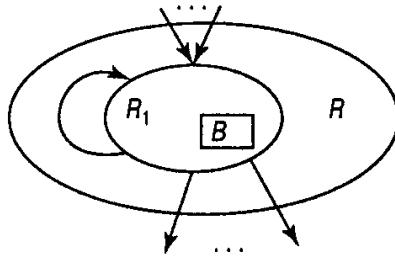


Рис. 10.54. Построение области путем преобразования T_1

1. Определение генерируется в R_1 и не требует для достижения B обратных дуг из R .
2. Определение генерируется в R_1 , достигает предшественника заголовка, следует по обратной дуге и не уничтожается при переходе от заголовка к B .

Если мы обозначим через G объединение $gen_{R_1, P}$ для всех предшественников заголовка в R , то

$$gen_{R,B} = gen_{R_1,B} \cup (G - kill_{R_1,B})$$

Определение уничтожается при переходе от заголовка к B тогда и только тогда, когда оно уничтожается вдоль всех нециклических путей, так что обратные дуги, включенные в R , не вызывают уничтожения большего числа определений, т.е.

$$kill_{R,B} = kill_{R_1,B}$$

Пример 10.40

Рассмотрим граф потока на рис. 10.50, (T_1, T_2) -декомпозиция которого показана на рис. 10.55, с именованными областями декомпозиции. Кроме того, на рис. 10.56 показаны гипотетические битовые векторы, представляющие три определения и информацию о том, генерируются ли они и уничтожаются ли в каждом из блоков на рис. 10.55.

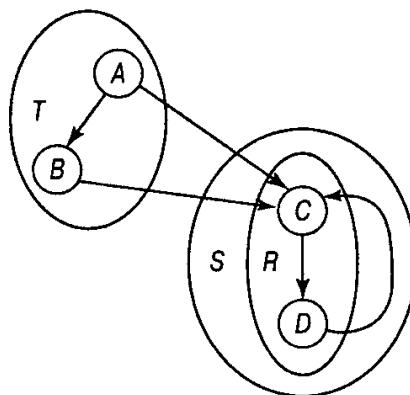


Рис. 10.55. Декомпозиция графа потока

БЛОК	<i>gen</i>	<i>kill</i>
A	100	010
B	010	101
C	000	010
D	001	000

Рис. 10.56. Информация о множествах *gen* и *kill* для блоков на рис. 10.55

Начиная работу изнутри наружу, заметим, что для областей, состоящих из одного узла, названных A, B, C и D , множества gen и $kill$ приведены в таблице на рис. 10.56. После этого мы можем продолжить работу с областью R , образованной при поглощении D блоком C путем преобразования T_2 . Следуя приведенным ранее для T_2 правилам, заметим, что gen и $kill$ для C не изменяются.

$$\begin{array}{lll} gen_{R,C} & = gen_{C,C} & = 000 \\ kill_{R,C} & = kill_{C,C} & = 010 \end{array}$$

Для узла D мы должны найти в области C объединение gen для всех предшественников заголовка области D . Конечно, заголовком области D является сам узел D , и имеется только один предшественник этого узла в области C , а именно — узел C . Таким образом,

$$\begin{array}{l} gen_{R,D} = gen_{D,D} \cup (gen_{C,C} - kill_{D,D}) = 001 + (000 - 000) = 001 \\ kill_{R,D} = kill_{D,D} \cup (kill_{C,C} - gen_{D,D}) = 000 + (010 - 001) = 010 \end{array}$$

Теперь мы строим область S из области R путем преобразования T_1 . Множества $kill$ не изменяются, так что мы имеем

$$\begin{array}{lll} kill_{S,C} & = kill_{R,C} & = 010 \\ kill_{S,D} & = kill_{R,D} & = 010 \end{array}$$

При вычислении множеств gen для S мы заметим, что единственной обратной дугой в заголовок S , которая включается при переходе от R к S , является дуга $D \rightarrow C$. Таким образом,

$$\begin{array}{l} gen_{S,C} = gen_{R,C} \cup (gen_{R,D} - kill_{R,C}) = 000 + (001 - 010) = 001 \\ gen_{S,D} = gen_{R,D} \cup (gen_{R,C} - kill_{R,D}) = 001 + (001 - 010) = 001 \end{array}$$

Вычисления для области T аналогичны вычислениям для области R , и мы получаем

$$\begin{array}{ll} gen_{T,A} & = 100 \\ kill_{T,A} & = 010 \\ gen_{T,B} & = 010 \\ kill_{T,B} & = 101 \end{array}$$

И наконец, мы вычисляем gen и $kill$ для области U — всего графа потока. Поскольку U строится путем поглощения областью T области S посредством преобразования T_2 , значения gen и $kill$ для узлов A и B не изменяются по сравнению с только что приведенными. Для C и D заметим, что заголовок S — узел C — имеет двух предшественников в области T , а именно — A и B . Следовательно, мы вычисляем

$$\begin{array}{l} G = gen_{T,A} \cup gen_{T,B} = 110 \\ K = kill_{T,A} \cap kill_{T,B} = 000 \end{array}$$

после чего можем найти

$$\begin{array}{l} gen_{U,C} = gen_{S,C} \cup (G - kill_{S,C}) = 101 \\ kill_{U,C} = kill_{S,C} \cup (K - gen_{S,C}) = 010 \\ gen_{U,D} = gen_{S,D} \cup (G - kill_{S,D}) = 101 \\ kill_{U,D} = kill_{S,D} \cup (K - gen_{S,D}) = 010 \end{array}$$

□

Имея вычисленные $gen_{U,B}$ и $kill_{U,B}$ для каждого блока B , где U является областью, состоящей из графа потока в целом, мы, по сути, вычислили $out[B]$ для каждого блока B . Иными словами, если мы рассмотрим определение $trans_{U,B}(S) = gen_{U,B} \cup (S - kill_{U,B})$, то заметим, что $trans_{U,B}(\emptyset) = out[B]$. Однако $trans_{U,B}(\emptyset) = gen_{U,B}$. Таким образом, заверше-

ние алгоритма достигающих определений на основе структур состоит в использовании множеств gen в качестве out и вычислении множеств in путем объединения множеств out предшественников. Эти шаги собраны в следующем алгоритме.

Алгоритм 10.1.7. Достигающие определения, основанные на структуре

Вход. Приводимый граф потока G и множества определений $gen[B]$ и $kill[B]$ для каждого блока B из G .

Выход. $in[B]$ для каждого блока B .

Метод.

1. Найти (T_1, T_2) -декомпозицию G .
2. Для каждой области R декомпозиции изнутри наружу вычислить $gen_{R,B}$ и $kill_{R,B}$ для каждого блока $B \in R$.
3. Если U — имя области, состоящей из всего графа, то для каждого блока B установить $in[B]$ равным объединению по всем предшественникам P блока B множеств $gen_{U,P}$. \square

Некоторые ускорения структурного алгоритма

Сначала заметим, что если у нас есть функция передачи $G \cup (S - K)$, то она не изменяется при удалении из K некоторых членов G . Таким образом, при применении T_2 в формулах

$$\begin{aligned} gen_{R,B} &= gen_{R_2,B} \cup (G - kill_{R_2,B}) \\ kill_{R,B} &= kill_{R_2,B} \cup (K - gen_{R_2,B}) \end{aligned}$$

вторую из них можно заменить следующей:

$$kill_{R,B} = kill_{R_2,B} \cup K$$

тем самым сэкономив по операции для каждого блока в области R_2 .

Еще одна идея основана на наблюдении о том, что преобразование T_1 мы применяем только после того, как сначала некоторая область R_2 поглощается областью R_1 и при этом имеются некоторые обратные дуги от R_2 к заголовку R_1 . Вместо того чтобы сначала вносить изменения в R_2 путем применения преобразования T_2 , а затем изменять R_1 и R_2 с помощью T_1 , мы можем скомбинировать эти два множества изменений следующим образом.

1. Используя правило T_2 , вычислим новую функцию передачи для тех узлов в R_2 , которые являются предшественниками заголовка R_1 .
2. Используя правило T_1 , вычислим новую функцию передачи для всех узлов R_1 .
3. Используя правило T_2 , вычислим новую функцию передачи для всех узлов R_2 . Заметим, что обратная связь, благодаря применению T_1 , достигает предшественников R_2 и передается всем узлам R_2 по правилу T_2 ; поэтому нет необходимости в применении правила T_1 к R_2 .

Обработка неприводимых графов потока

Если (T_1, T_2) -приведение графа потока останавливается на предельном графе потока, который не является единственным узлом, мы должны выполнить разделение узлов. Разделение узла предельного графа потока соответствует созданию копии области, представленной этим узлом. Например, на рис. 10.57 показан возможный результат разделения узла по отношению к исходному графу потока, состоящему из девяти узлов, который был разделен преобразованиями T_1 и T_2 на три области, соединенные дугами.

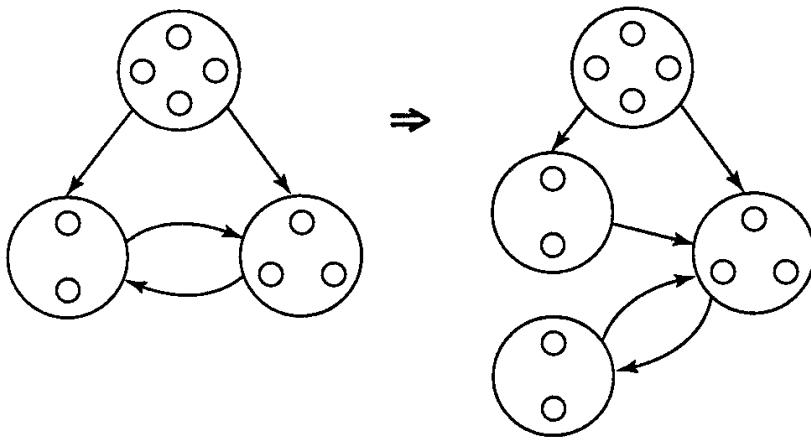


Рис. 10.57. Разделение неприводимого графа потока

Как упоминалось в предыдущем разделе, чередуя разделение узлов с последовательностями приведений, мы гарантированно приводим граф потока к единому узлу. Результатом разделения является появление копий узлов исходного графа в области, представленной графиком, состоящим из одного узла. Мы можем применить к такой области немного измененный алгоритм 10.17. Единственное его отличие состоит в том, что когда мы разделяем узел, множества *gen* и *kill* для узлов исходного графа в области, представленной разделяемым узлом, должны быть дублированы. Например, любые значения множеств *gen* и *kill* в двухузельной области слева на рис. 10.57 становятся множествами *gen* и *kill* для каждого из соответствующих узлов в обеих двухузельных областях справа. На последнем этапе при вычислении множеств *in* для всех узлов, те узлы исходного графа, которые содержат несколько представителей в окончательной области, имеют значения *in*, полученные объединениями *in* всех своих представителей.

В наихудшем случае разделение узлов может привести к экспоненциальному росту количества узлов, представленных всеми областями. Таким образом, если мы ожидаем большого количества неприводимых графов потока, вероятно, не стоит использовать структурные методы. К счастью, неприводимые графы потока — достаточно большая редкость, что обычно позволяет нам игнорировать стоимость разделения узлов.

10.11. Средства для анализа потока данных

Как мы указывали ранее, различные изученные нами задачи потока данных во многом сходны. Уравнения потока данных из раздела 10.6 различаются следующим.

1. Используемой функцией передачи, которая в каждом рассмотренном нами случае имеет вид $f(X) = A \cup (X - B)$. Например, для достигающих определений $A = kill$ и $B = gen$.
2. Оператором слияния, который во всех рассмотренных нами случаях представлял собой либо объединение, либо пересечение.
3. Направлением распространения информации — вперед или назад.

Поскольку эти различия не столь значительны, неудивительно, что все эти задачи можно рассматривать одинаково. Такой подход был описан в работе [249], а соответствующий способ упрощения реализации задач потока данных, разработанный Килдаллом (G. Kildall), был использован им в ряде компиляторов. Эта методика не получила широкого распространения, видимо, по той причине, что экономия труда при ее применении

не так велика, как при использовании инструментария типа генераторов синтаксических анализаторов. Однако нам следует рассмотреть и этот инструментарий, и не только потому, что он позволяет упростить создание оптимизирующих компиляторов, но и главным образом потому, что он помогает унифицировать различные идеи, с которыми мы встречались в этой главе. Далее в этом разделе будет рассказано, каким образом могут быть разработаны более мощные стратегии анализа потока данных, обеспечивающие более точную, по сравнению с рассмотренными алгоритмами, информацию.

Схема анализа потока данных

Здесь мы опишем схемы, моделирующие задачи прямого распространения информации. Если мы рассматриваем только итеративные решения задач потока данных, то направление не имеет значения — мы можем просто обратить направления дуг и внести минимальные изменения для учета начального узла, после чего рассматривать обратную задачу так же, как и прямую. Алгоритмы, основанные на структуре, несколько отличны — прямая и обратная задачи не могут быть решены одинаково в силу того, что обращение приводимого графа потока может не быть приводимо. Однако мы ограничимся рассмотрением только прямых задач, оставив рассмотрение обратных задач в качестве упражнения.

Схема анализа потока данных состоит из следующих компонентов.

1. Множество V распространяемых значений. Значения из множеств in и out являются элементами V ¹⁶.
2. Множество F функций передачи из V в V .
3. Бинарная операция слияния (meet operation) \wedge на множестве V , представляющая оператор слияния.

Пример 10.41

Для достигающих определений V состоит из всех подмножеств множества определений в программе. Множество F является множеством всех функций вида $f(X) = A \cup (X - B)$, где A и B — множества определений, т.е. члены V ; A и B — множества, которые мы именуем соответственно *gen* и *kill*. И наконец, операция \wedge является объединением.

Для доступных выражений V состоит из всех подмножеств множества выражений, вычисляемых программой, а F — множество выражений того же вида, что и ранее; однако A и B в данном случае являются множествами выражений. Оператор слияния в данном случае представляет собой пересечение. \square

Пример 10.42

Подход Килдалла не ограничивается простыми примерами, с которыми мы имели дело, несмотря на растущую при усложнении задач сложность метода как с точки зрения времени вычислений, так и с точки зрения интеллектуальных усилий. В упражнениях вы найдете очень сложный и мощный пример, в котором вычисленная информация о потоках данных говорит нам, по сути, о всех парах выражений, имеющих одно и то же значение в некоторой точке. Однако часть этого примера составляет метод, обеспечивающий нас знаниями о том, какие из переменных содержат постоянные значения, и дающий

¹⁶ Элементы этого множества далее называются членами. — Прим. ред.

больше информации, чем достигающие определения. Наша новая схема понимает, например, что когда x определяется инструкцией $d: x := x + 1$ и переменная x до присвоения имела постоянное значение, то ее значение остается константой и после присвоения.

Использование достигающих определений для распространения констант, напротив, полагает d возможным определением x и на этом основании считает, что x не имеет постоянного значения. Конечно, при одном проходе правая часть инструкции $d: x := x + 1$ будет заменена константой, и второй цикл распространения констант сможет определить, что использование x , определенного в d , на самом деле представляет собой использование константы.

В новой схеме множество V является множеством всех отображений переменных программы на определенное множество значений. Это множество значений состоит из следующих элементов.

1. Все константы.
2. Значение *nonconst*, которое означает, что рассматриваемая переменная определена как не имеющая постоянного значения. Значение *nonconst* присваивается переменной x , если, например, в процессе анализа потока данных мы нашли два пути, на которых переменной x присваиваются значения 2 и 3, или нашли путь, для которого определением x является считающая x инструкция.
3. Значение *undef*, которое означает, что о рассматриваемой переменной пока нельзя сказать ничего определенного, поскольку, например, при анализе потока данных до этого момента не найдено ни одно определение этой переменной.

Заметим, что *nonconst* и *undef* — это два разных значения, по сути, противоположные друг другу. Первое значение говорит нам о том, что мы встретили много путей определения этой переменной, так что она не может быть константой; второе значение гласит, что у нас так мало знаний о переменной, что мы не можем сказать о ней ничего.

Операция слияния определена с помощью следующей таблицы. Пусть μ и ν — два члена V , т.е. и μ , и ν отображают каждую переменную на константу, *undef* или *nonconst*. Тогда функция $\rho = \mu \wedge \nu$ определена так, как показано на рис. 10.58, где мы выразили значение $\rho(x)$ через $\nu(x)$ и $\mu(x)$ для каждой переменной x . В этой таблице c — произвольная константа, а d — некоторая другая константа, не равная c . Например, если $\mu(x)=c$ и $\nu(x)=d$, то x , вероятно, принимает значения c и d на различных путях, и при слиянии этих путей значение x не является константой; следовательно, в этом случае $\rho(x)=\text{nonconst}$. Рассмотрим еще один пример. Если вдоль одного пути об x не известно ничего, т.е. $\mu(x)=\text{undef}$, а вдоль другого x принимает значение c , то после слияния этих путей мы можем утверждать, что x принимает значение c . Конечно, позже может появиться дополнительная информация о первом пути и поведении на нем x , и тогда значение x после слияния этих путей примет, например, значение *nonconst*.

$\nu(x)$	$\mu(x)$			
	<i>nonconst</i>	c	$d (\neq c)$	<i>undef</i>
<i>nonconst</i>	<i>nonconst</i>	<i>nonconst</i>	<i>nonconst</i>	<i>nonconst</i>
c	<i>nonconst</i>	c	<i>nonconst</i>	c
<i>undef</i>	<i>nonconst</i>	c	d	<i>undef</i>

Рис. 10.58. $\rho(x)$, выраженное через $\nu(x)$ и $\mu(x)$

И наконец, мы должны построить множество функций F , которые отображают передачу информации от начала к концу каждого блока. Описание этого множества функций сложное, хотя идеи при этом достаточно просты. Поэтому мы представим “базис” этого множества описанием функций, которые представляют единичные инструкции определений, после чего все множество функций может быть построено путем композиции функций из этого базисного множества отображения блоков и более чем одной определяющей инструкции.

1. В F входит тождественная функция. Эта функция отражает блоки, не содержащие определяющих инструкций. Если I — тождественная функция, а μ представляет собой любое отображение переменных в значения, то $I(\mu) = \mu$. Заметим, что само по себе μ произвольно и не обязано быть тождественным.
2. Для каждой переменной x и константы c имеется функция $f \in F$, такая, что для любого отображения $\mu \in V$ мы имеем $f(\mu) = v$, где $v(w) = \mu(w)$ для всех w , отличающихся от x , и $v(x) = c$. Эти функции отражают влияние инструкции присвоения $x := c$.
3. Для каждого трех (не обязательно различных) переменных x , y и z существует функция $f \in F$, такая, что для каждого отображения $\mu \in V$ мы имеем $f(\mu) = v$. Отображение v определяется следующим образом. Для каждого w , кроме x , мы имеем $v(w) = \mu(w)$, и $v(x) = \mu(y) + \mu(z)$. Если либо $\mu(y)$, либо $\mu(z)$ — *nonconst*, сумма равна *nonconst*. Если либо $\mu(y)$, либо $\mu(z)$ — *undef*, то и сумма равна *undef*. Эта функция выражает результат присвоения $x := y + z$. Как принято в этой главе, оператор $+$ означает обобщенный оператор. В случае унарного, тернарного или операторов более высокого порядка требуется внесение очевидных изменений; столь же очевидны и изменения, требуемые для учета инструкций копирования $x := y$.
4. Для каждой переменной x имеется функция $f \in F$, такая, что $f(\mu) = v$ для каждого μ , где $v(w) = \mu(w)$ для всех w , отличающихся от x , и $v(x) = \text{nonconst}$. Эта функция отражает определение x чтением, поскольку после инструкции чтения x должно считаться не принимающим ни одного определенного постоянного значения. \square

Аксиомы схем анализа потока данных

Для того чтобы сделать рассмотренные ранее алгоритмы потоков данных работающими с произвольными схемами, нам требуются некоторые исходные предположения о множестве V , множестве функций F и операторе слияния \wedge . Наши основные предположения перечислены ниже, хотя некоторые из алгоритмов требуют дополнительных предположений.

1. F содержит тождественную функцию, такую, что $I(\mu) = \mu$ для всех $\mu \in V$.
2. F замкнуто относительно композиции функций, т.е. для любых двух функций f и g из F функция h , определенная как $h(\mu) = g(f(\mu))$, также принадлежит F .
3. Оператор \wedge является ассоциативным, коммутативным и идемпотентным. Эти свойства алгебраически выражаются как

$$\begin{aligned}\mu \wedge (v \wedge \rho) &= (\mu \wedge v) \wedge \rho \\ \mu \wedge v &= v \wedge \mu \\ \mu \wedge \mu &= \mu\end{aligned}$$

для всех μ , v и ρ из V .

4. В V имеется наибольший элемент T , такой, что для всех $\mu \in V$ выполняется $T \wedge \mu = \mu$.

Пример 10.43

Рассмотрим достигающие определения. В этом случае F , конечно же, содержит тождественную функцию, когда gen и $kill$ представляют собой пустые множества. Чтобы показать замкнутость относительно композиции функций, предположим, что у нас есть две функции

$$\begin{aligned}f_1(X) &= G_1 \cup (X - K_1) \\f_2(X) &= G_2 \cup (X - K_2)\end{aligned}$$

Тогда

$$f_2(f_1(X)) = G_2 \cup ((G_1 \cup (X - K_1)) - K_2)$$

Мы можем убедиться, что правая часть алгебраически эквивалентна

$$(G_2 \cup (G_1 - K_2)) \cup (X - (K_1 \cup K_2))$$

Если положить $K = K_1 \cup K_2$ и $G = (G_2 \cup (G_1 - K_2))$, то мы показали, что композиция f_1 и f_2 , которая представляет собой $f(X) = G \cup (X - K)$, имеет вид, делающий ее членом F .

Что касается оператора слияния, который в данном случае является объединением, то не представляет трудности убедиться в том, что он ассоциативен, коммутативен и идемпотентен. Наибольший элемент в этом случае представляет собой пустое множество, поскольку для любого множества X выполняется $\emptyset \cup X = X$.

При рассмотрении доступных выражений мы обнаружим, что показать наличие тождественного элемента и замкнутости относительно композиции функций можно так же, как и в случае достигающих определений. Оператор слияния в данном случае — пересечение, которое также является ассоциативным, коммутативным и идемпотентным. Наибольший элемент в данном случае более соответствует своему названию, представляя собой множество E всех выражений программы, поскольку для любого множества выражений X выполняется соотношение $E \cap X = X$. \square

Пример 10.44

Рассмотрим схему вычисления констант, приведенную в примере 10.42. Множество функций F сконструировано так, чтобы включать тождественную функцию и быть замкнутым относительно композиции функций. При проверке алгебраических законов для оператора \wedge достаточно показать, что они применимы для каждой переменной x . В качестве примера проверим идемпотентность оператора. Пусть $v = \mu \wedge \mu$, т.е. для всех x $v(x) = \mu(x) \wedge \mu(x)$. Рассмотрев различные варианты x , очень легко убедиться, что $v(x) = \mu(x)$. Например, если $\mu(x) = \text{nonconst}$, то и $v(x) = \text{nonconst}$, поскольку в соответствии с рис. 10.58 результат операции над двумя nonconst равен nonconst .

И наконец, наибольшим элементом является отображение τ , определенное как $\tau(x) = \text{undef}$ для всех x . Из таблицы на рис. 10.58 мы можем убедиться, что для любого отображения μ и любой переменной x , если v — функция $\tau \wedge \mu$, то $v(x) = \mu(x)$, поскольку результат операции над undef и любым другим значением дает это значение. \square

Монотонность и дистрибутивность

Для работы итеративного алгоритма анализа потока данных нам требуется другое условие, именуемое монотонностью. Неформально оно означает, что если мы возьмем любую функцию f из множества F и применим f к двум членам V , один из которых

“больше” другого, то результат применения f к большему из них не меньше результата применения f к меньшему члену.

Для того чтобы сделать понятие “больше” точным, определим отношение \leq на множестве V как $\mu \leq v$ тогда и только тогда, когда $\mu \wedge v = \mu$.

Пример 10.45

В схеме достигающих определений, где оператор слияния представляет собой объединение, а члены V — множества определений, $X \leq Y$ означает $X \cup Y = X$, т.е. X представляет собой надмножество Y . Таким образом, отношение \leq выглядит “обратным” — меньший элемент V представляет собой надмножество большего.

Для доступных выражений, где слияние представляет собой пересечение, можно говорить о “правильности” отношения, так как $X \leq Y$ означает, что $X \cap Y = X$, т.е. X является подмножеством Y . \square

Заметим из примера 10.45, что отношение \leq в нашем смысле может не обладать свойствами отношения \leq для целых чисел. Отношение \leq транзитивно; в качестве упражнения, используя аксиомы для операции \wedge , читатель может доказать, что из $\mu \leq v$ и $v \leq \rho$ следует $\mu \leq \rho$. Однако отношение \leq в нашем смысле не является линейным порядком. Например, в схеме доступных выражений у нас может быть два множества X и Y , таких, что ни одно из них не является подмножеством другого; при этом ни одно из отношений $X \leq Y$ или $Y \leq X$ не является истинным.

Зачастую пониманию ситуации помогает изображение множества V в виде *решеточной диаграммы* — графа, узлы которого являются элементами V , а дуги направлены вниз от X к Y , если $Y \leq X$. Например, на рис. 10.59 показано множество V для задачи достижимых определений, в котором имеется три определения d_1 , d_2 и d_3 . Поскольку отношение \leq означает “надмножество”, дуги направлены вниз от любого подмножества этих трех определений к каждому из их надмножеств. Поскольку отношение \leq транзитивно, мы условно опускаем дугу от X к Y , если имеется другой путь от X к Y , остающийся в диаграмме. Таким образом, хотя $\{d_1, d_2, d_3\} \leq \{d_1\}$, мы не изображаем соответствующую дугу, так как на рисунке имеется путь, проходящий, например, через узел $\{d_1, d_2\}$.

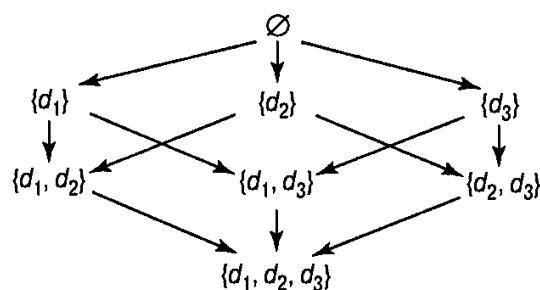


Рис. 10.59. Решетка подмножеств определений

Полезно заметить, что из таких диаграмм мы можем истолковать слияние, поскольку $X \wedge Y$ представляет собой наибольшее Z , к которому ведут нисходящие пути от X и Y . Например, если $X = \{d_1\}$, $Y = \{d_2\}$, то Z на рис. 10.59 — $\{d_1, d_2\}$, что естественно, поскольку оператором слияния является объединение. Кроме того, наибольший элемент будет находиться на вершине решеточной диаграммы, т.е. имеется нисходящий путь от \top к каждому элементу диаграммы.

Теперь мы можем определить схему (F, V, \wedge) как монотонную, если

из $\mu \leq v$ следует $f(\mu) \leq f(v)$ (10.15)
для всех μ и v из V и f из F .

Монотонность можно определить эквивалентным способом:

$$f(\mu \wedge v) \leq f(\mu) \wedge f(v) \quad (10.16)$$

для всех μ и v из V и f из F . Полезно сравнить эти два определения, так что мы приведем набросок доказательства их эквивалентности с использованием определения отношения \leq и ассоциативности, коммутативности и идемпотентности оператора \wedge , оставив читателю проверку некоторых простых наблюдений.

Покажем, что из справедливости (10.15) вытекает (10.16). Заметим, что для любых μ и v выполняется как $\mu \wedge v \leq \mu$, так и $\mu \wedge v \leq v$. Читателю предлагается обосновать это, доказав, что для любых x и y справедливо $(x \wedge y) \wedge y = x \wedge y$. Следовательно, в соответствии с (10.15), $f(\mu \wedge v) \leq f(\mu)$ и $f(\mu \wedge v) \leq f(v)$. Читатель может самостоятельно убедиться в справедливости общего закона, согласно которому из $x \leq y$ и $x \leq z$ следует $x \leq y \wedge z$. Принимая $x = f(\mu \wedge v)$, $y = f(\mu)$ и $z = f(v)$, получим (10.16).

Докажем теперь, что из (10.16) следует (10.15). Предположим, что $\mu \leq v$, и используем (10.16) для вывода $f(\mu) \leq f(v)$, доказав, таким образом, (10.15). Уравнение (10.16) говорит нам, что $f(\mu \wedge v) \leq f(\mu) \wedge f(v)$. Но в силу предположения $\mu \leq v$ по определению $\mu \wedge v = \mu$. Следовательно, (10.16) гласит, что $f(\mu) \leq f(\mu) \wedge f(v)$. Читатель может доказать общее правило, что если $x \leq y \wedge z$, то $x \leq z$. Следовательно, из (10.16) следует $f(\mu) \leq f(v)$, и мы доказали (10.15).

Зачастую схема подчиняется более строгому по сравнению с (10.16) условию, которое называется *условием дистрибутивности*:

$$f(\mu \wedge v) = f(\mu) \wedge f(v)$$

для всех μ и v из V и f из F . Безусловно, если $x = y$, то в соответствии с идемпотентностью $x \wedge y = x$ и $x \leq y$. Следовательно, из дистрибутивности вытекает монотонность.

Пример 10.46

Рассмотрим схему достигающих определений. Пусть X и Y представляют множества определений и f — функция, определяемая соотношением $f(Z) = G \cup (Z - K)$ для некоторых множеств определений G и K . Тогда мы можем убедиться, что схема достигающих определений удовлетворяет условию дистрибутивности, убедившись, что

$$G \cup ((X \cup Y) - K) = (G \cup (X - K)) \cup (G \cup (Y - K))$$

Хотя доказательство приведенного соотношения кажется сложным, оно легко выполняется с помощью диаграмм Венна. \square

Пример 10.47

Покажем, что схема вычисления констант монотонна, но не дистрибутивна. Это облегчает применение операции \wedge и отношения \leq к элементам таблицы на рис. 10.58. Итак, определим

$nonconst \wedge c = nonconst$	для любой константы c
$c \wedge d = nonconst$	для констант $c \neq d$
$c \wedge undef = c$	для любой константы c
$nonconst \wedge undef = nonconst$	
$x \wedge x = x$	для любого значения x

Тогда рис. 10.58 может быть интерпретирован как гласящий, что $\rho(a) = \mu(a) \wedge v(a)$.

По операции \wedge можно определить, что представляет собой отношение \leq на множестве значений:

$$\begin{array}{ll} \text{nonconst} & \leq c \\ & \text{для любой константы } c \\ c & \leq \text{undef} \\ \text{nonconst} & \leq \text{undef} \end{array}$$

Это отношение показано на решеточной диаграмме (рис. 10.60), где c_i означают все возможные константы. Заметим, что этот рисунок изображает не отношение \leq для элементов V , а отношение на множестве значений $\mu(a)$ для отдельных значений a . Элементы V можно считать векторами таких значений, по одному компоненту для каждой переменной, и решеточная диаграмма для V может быть экстраполирована из рис. 10.60. Вспомним, что соотношение $\mu \leq v$ справедливо тогда и только тогда, когда для всех a справедливо $\mu(a) \leq v(a)$, т.е. все компоненты векторов, представляющих μ и v , связаны отношением \leq , и это отношение имеет одно и то же направление для всех компонентов.

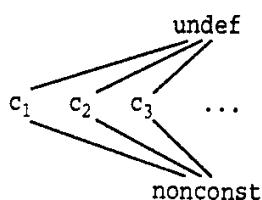


Рис. 10.60. Решеточная диаграмма для значений переменных

Следовательно, $\mu \leq v$ означает, что когда $\mu(a)$ — константа, то $v(a)$ — либо константа, либо *undef*, а когда $\mu(a)$ — *undef*, то и $v(a)$ — *undef*. Внимательное изучение различных функций f , связанных с различными видами инструкций определения, позволяет нам убедиться, что если $\mu \leq v$, то $f(\mu) \leq f(v)$; это доказывает таким образом (10.15) и показывает монотонность. Например, если f связана с присвоением $a := b+c$, то изменяются только $\mu(a)$ и $v(a)$, поэтому мы должны проверить, что если $\mu \leq v$, т.е. $\mu(x) \leq v(x)$ для всех x , то $[f(\mu)](a) \leq [f(v)](a)$ ¹⁷. Мы должны рассмотреть все возможные значения $\mu(b)$, $\mu(c)$, $v(b)$ и $v(c)$, предмет ограничений $\mu(b) \leq v(b)$ и $\mu(c) \leq v(c)$. Например, если

$$\begin{array}{ll} \mu(b) & = \text{nonconst} \\ v(b) & = 2 \\ \mu(c) & = 3 \\ v(c) & = \text{undef} \end{array}$$

то $[f(\mu)](a) = \text{nonconst}$ и $[f(v)](a) = \text{undef}$. Поскольку $\text{nonconst} \leq \text{undef}$, мы выполнили проверку одного из вариантов; проверка остальных вариантов остается читателю в качестве упражнения.

Теперь мы должны проверить наше утверждение о том, что схема вычисления констант не является дистрибутивной. Для этого примем, что функция f связана с присвоением $a := b+c$, и пусть $\mu(b) = 2$, $\mu(c) = 3$, $v(b) = 3$ и $v(c) = 2$. Пусть $\rho = \mu \wedge v$. Тогда $\mu(b) \wedge v(b) = 2 \wedge 3 = \text{nonconst}$ и, аналогично, $\mu(c) \wedge v(c) = \text{nonconst}$, что эквивалентно

¹⁷ Мы должны внимательно прочесть выражение типа $[f(\mu)](a)$. Оно гласит, что мы применяем f к μ для получения некоторого отображения $f(\mu)$, которое мы назовем μ' . Затем мы применяем μ' к a , и результат оказывается одним из значений на диаграмме на рис. 10.60.

$\rho(b) = \rho(c) = \text{nonconst}$. Отсюда следует, что $[f(\rho)](a) = \text{nonconst}$, поскольку сумма двух неконстантных значений не является константой.

В то же время $[f(\mu)](a) = 5$, поскольку из того, что $b = 2$ и $c = 3$, следует, что инструкция $a := b+c$ присваивает a значение 5. Аналогично $[f(v)](a) = 5$. Следовательно, $[f(\mu) \wedge f(v)](a) = 5$. Теперь мы видим, что $\rho(a) = [f(\mu \wedge v)](a) \neq [f(\mu) \wedge f(v)](a)$, так что условие дистрибутивности оказывается не выполненным.

Интуитивно причина нарушения условия дистрибутивности состоит в том, что схема вычисления констант недостаточно мощна для запоминания всех инвариантов (в частности, того факта, что вдоль путей, чье действие на переменные описывается либо μ , либо v , выполняется равенство $b+c=5$, хотя ни b , ни c константами не являются). Для избежания этой частной проблемы можно разработать более сложную схему, хотя выгода от такого усложнения неясна. К счастью, как мы увидим позже, монотонности вполне достаточно для работоспособности итеративных алгоритмов потока данных. \square

Решения типа „слияние всех путей“ в задачах о потоках данных

Представим, что граф потока имеет связанную с каждым своим узлом функцию передачи — одну из функций множества F . Для каждого блока B обозначим соответствующую функцию передачи как f_B .

Рассмотрим произвольный путь $P = B_0 \rightarrow B_1 \rightarrow \dots \rightarrow B_k$ из начального узла B_0 в некоторый блок B_k . Мы можем определить функцию передачи для P , как композицию функций $f_{B_0}, f_{B_1}, \dots, f_{B_{k-1}}$. Заметим, что f_{B_k} не является частью композиции, отражая точку зрения, что этот путь использован для достижения начала блока B_k , а не его конца.

Мы предположили, что значения в V представляют информацию о данных, используемых программой, и что оператор слияния \wedge говорит о том, каким образом происходит комбинирование этой информации при слиянии путей. Естественно также считать, что наибольший элемент представляет “отсутствие информации”, поскольку путь, несущий наибольший элемент, уступает любому другому пути в вопросе, какая именно информация несется после слияния путей. Таким образом, если B является блоком графа потока, информация при входе в блок B должна быть вычислима путем изучения каждого из всех возможных путей от начального узла к B и рассмотрения того, что происходит на пути, начиная с отсутствия информации, т.е. для каждого пути от B_0 к B мы вычисляем $f_P(T)$ и выполняем слияние всех полученных значений.

В принципе, такое слияние может выполняться над неограниченным количеством различных значений, поскольку может существовать бесконечно много различных путей. На практике часто достаточно рассмотреть только ациклические пути, и даже когда этого не достаточно (например, как для обсуждаемой ранее схемы вычисления констант), обычно для любого конкретного графа потока можно найти другие причины, чтобы сделать это бесконечное слияние конечным.

Формально мы определяем *решение типа “слияние всех путей”* (meet-over-paths solution), или *top-решение*, для графа потока как

$$\text{top}(B) = \bigwedge_{\substack{\text{Путь } P \\ \text{от } B_0 \text{ к } B}} f_P(T)$$

top-решение для графа потока станет естественным, когда мы поймем, что если рассматривается информация, достигающая блока B , то граф потока может иметь вид, представленный на рис. 10.61. При этом функция передачи, связанная с каждым из (возмож-

но, бесконечного количества) путей P_1, P_2, \dots , в исходном графе потока получила в качестве аргумента свой собственный путь в B . На рис. 10.61 информация, достигающая B , получается слиянием по всем путям.

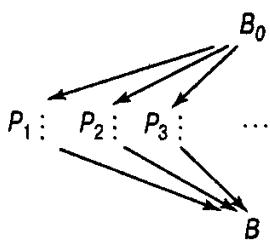


Рис. 10.61. Граф, показывающий множество всех возможных путей в B

Консервативные решения задач о потоках

Когда мы пытаемся решить уравнения потока данных, исходящие от произвольной схемы, получение *top*-решения может оказаться как легко достижимым, так и достаточно трудным делом. К счастью, как и в случае конкретных примеров схем потока данных в разделах 10.5 и 10.6, для ошибок существует “безопасное направление”, и итеративный алгоритм потока данных, который обсуждался в этих разделах, всегда дает нам безопасное решение. Мы говорим, что решение $in[B]$ является *безопасным*, если $in[B] \leq top(B)$ для всех блоков B .

Наше определение небезосновательно, что бы об этом ни думал читатель. Вспомним, что в любом графе потока множество *видимых* (apparent) путей к узлу (т.е. являющихся путями в графе потока) может быть собственным надмножеством множества *реальных* (real) путей, которые получаются при некотором выполнении программы, соответствующей этому графу потока. Для того чтобы результат анализа потока данных был применим для изначально ставившихся перед ним целей, эти данные должны оставаться безопасными при удалении из графа потока некоторых путей, поскольку в общем случае мы не в состоянии отличить реальные пути от видимых путей, не являющихся реальными.

Предположим, что среди бесконечного множества путей, предложенных на рис. 10.61, x представляет собой слияние $f_P(T)$, взятое по всем реальным путям P , по которым следует некоторое выполнение программы. Пусть y также представляет собой слияние всех $f_P(T)$ по всем другим путям P . Таким образом, $top(B) = x \wedge y$. Итак, точным ответом нашей задачи потока данных в узле B является x , в то время как *top*-решение представляет собой $x \wedge y$. Вспомним теперь, что $x \wedge y \leq y$, поскольку $(x \wedge y) \wedge y = x \wedge y$, а следовательно, *top*-решение \leq точного решения.

Предпочитая получение точного решения задачи потока данных, мы, тем не менее, почти гарантированно не имеем эффективного метода выяснить, какие пути являются реальными, а какие — нет, так что мы вынуждены принимать *top*-решение в качестве наиболее подходящего. А значит, любое использование полученной нами информации о потоке данных должно допускать возможность того, что полученное нами решение \leq точного. Принимая это требование, мы должны допустить и возможность решения, которое \leq *top*-решения (и, соответственно, \leq точного решения). Получить такие решения для схем, являющихся монотонными, но не дистрибутивными, легче, чем *top*-решения. Для дистрибутивных схем, подобных рассмотренным в разделе 10.6, *top*-решение вычисляется простым итеративным алгоритмом.

Итеративный алгоритм для обобщенных схем

Имеется очевидное обобщение алгоритма 10.2, работающее для большого множества схем. Этот итеративный алгоритм требует, чтобы схема была монотонной и конечной в том смысле, что слияние бесконечного множества путей, представленное на рис. 10.61, эквивалентно слиянию конечного подмножества. Мы приведем алгоритм, а затем обсудим способы, как убедиться в конечности схемы. Однако скажем сразу об одной общей гарантии конечности, которая была у нас с самого начала: распространение вдоль ациклических путей является достаточным.

Алгоритм 10.18. Итеративное решение для обобщенных схем¹⁸ потока данных

Вход. Граф потока, множество “значений” V , множество функций F , оператор слияния \wedge и присвоение каждому узлу графа потока элемента F .

Выход. Значение $in[B] \in V$ для каждого узла графа.

Метод. Алгоритм приведен на рис. 10.62. Как и в случае знакомых нам итеративных алгоритмов потока данных, мы вычисляем in и out для каждого узла путем последовательных приближений. Здесь предполагается, что с блоком B связана функция f_B из множества F ; эта функция играет роль множеств gen и $kill$ из раздела 10.6. \square

```
(1) for Каждый узел  $B$  do
    /* Инициализация в предположении  $in[B] = T^*$  */
(2)      $out[B] := f_B[T]$ ;
(3) while Имеются изменения в любом из  $out$  do
(4)     for Каждый блок  $B$  в порядке вглубь do begin
(5)          $in[B] := \bigwedge_{\substack{P-\text{пред-} \\ \text{шественник } B}} out[P]$ ;
        /* В строке выше слияние пустого множества дает  $T^*$  */
(6)          $out[B] := f_B(in[B])$ 
    end
```

Рис. 10.62. Итеративный алгоритм для обобщенных схем

Инструмент анализа потока данных

Теперь мы можем увидеть, каким образом идеи, изложенные в данном разделе, могут быть воплощены в виде инструмента для анализа потока данных. В своей работе алгоритм 10.18 зависит от следующих подпрограмм.

1. Подпрограмма применения данной f_B из F к данному значению из V . Эта подпрограмма используется в строках (2) и (6) на рис. 10.62.
2. Подпрограмма применения оператора слияния к двум значениям из V , которая используется в строке (5).
3. Подпрограмма, проверяющая равенство двух значений. Явно эта проверка на рис. 10.62 не показана, но используется при выяснении вопроса о том, изменилось ли в ходе итерации какое-либо из значений out .

¹⁸ Под обобщенными схемами (general frameworks) здесь понимаются схемы в их общем виде (V, F, \wedge) . — Прим. ред.

Кроме того, для передачи аргументов упомянутым подпрограммам нам требуются соответствующие объявления типов данных для F и V . Значения in и out имеют тот же тип, что и V . И наконец, нам нужна подпрограмма, которая будет получать обычное представление содержимого базового блока, т.е. список инструкций, и возвращать элемент множества F — функцию передачи блока.

Пример 10.48

Для схемы достигающих определений мы можем сначала построить таблицу, нумерующую инструкции данного графа потока целыми числами от 1 до некоторого максимального m . В таком случае типом V может быть битовый вектор длиной m , а F можно представить парой битовых векторов этого размера, т.е., по сути, множествами gen и $kill$. Подпрограмма построения битовых векторов gen и $kill$ для данных инструкций блока и таблицы, связывающей инструкции определений с позициями в битовом векторе, достаточно проста, так же как и подпрограммы для вычисления слияний (побитовая операция or над векторами), проверки равенства битовых векторов и применения функций, определенных парами $gen-kill$, к битовым векторам. \square

Инструмент анализа потока данных представляет собой нечто большее, чем просто реализацию алгоритма на рис. 10.62 с вызовами описанных подпрограмм при необходимости вычислить слияние, применить функцию или выполнить сравнение. Инструмент должен поддерживать фиксированное представление графов потока, а следовательно, должен быть способным выполнять задачи наподобие поиска всех предшественников данного узла, упорядочения графа вглубь или применения к каждому блоку подпрограммы вычисления связанной с ним функции из множества F . Преимущество такого инструмента заключается в том, что нам не придется заново переписывать подпрограммы работы с графиком или проверки сходимости из алгоритма 10.18 для каждого выполняемого нами анализа потока данных.

Свойства алгоритма 10.18

Нам следует прояснить, при каких предположениях работает алгоритм 10.18 и к какому результату приводит (в случае, если алгоритм сходится).

Во-первых, если схема монотонна и сходится, то мы утверждаем, что для всех блоков B справедливо соотношение $in[B] \leq top(B)$. Интуитивное обоснование этого состоит в том, что для любого пути $P = B_0, B_1, \dots, B_k$ от начального узла к $B = B_k$ мы можем индукцией по i показать, что влияние пути от B_0 к B_i ощущается после не более чем i итераций цикла `while` на рис. 10.62, т.е. если P_i представляет собой путь B_0, B_1, \dots, B_i , то после i итераций $in[B_i] \leq f_{P_i}(\top)$. Таким образом, когда и если алгоритм сходится, для любого пути P от B_0 к B будет справедливо соотношение $in[B] \leq f_p(\top)$. Воспользовавшись тем, что из $x \leq y$ и $x \leq z$ вытекает $x \leq y \wedge z$ ¹⁹, мы можем показать, что $in[B] \leq top(B)$.

Во-вторых, в случае дистрибутивной схемы мы можем показать, что алгоритм 10.18 в действительности сходится к *top*-решению. Основная идея состоит в доказательстве того, что в любой момент времени работы алгоритма $in[B]$, и $out[B]$ равны слиянию

¹⁹ В принципе мы должны показать, что это правило применимо не только для двух значений y и z (откуда следует правило, что если $x \leq y$, для любого конечного множества y , то $x \leq \bigwedge_i y_i$), но и для бесконечного количества y . Однако на практике всякий раз при сходимости алгоритма 10.18 мы находим конечное множество путей, таких, что слияние всех путей дает тот же результат, что и слияние этого конечного множества.

$f_P(T)$ для некоторого множества путей P , ведущих соответственно к началу и концу B . Однако в следующем примере мы покажем, что в случае монотонной, но не дистрибутивной схемы это может оказаться неверным.

Пример 10.49

Воспользуемся в качестве примера недистрибутивности схемой вычисления констант, рассмотренной в примере 10.47; соответствующий граф потока показан на рис. 10.63. Отображения μ и ν , выходящие из B_2 и B_4 , те же, что и в примере 10.47. Отображение ρ , входящее в блок B_5 , представляет собой $\mu \wedge \nu$, и σ (отображение, выходящее из B_5), устанавливает переменную a равной *nonconst*, несмотря на то что каждый реальный (и каждый видимый) путь вычисляет $a = 5$ после B_5 .

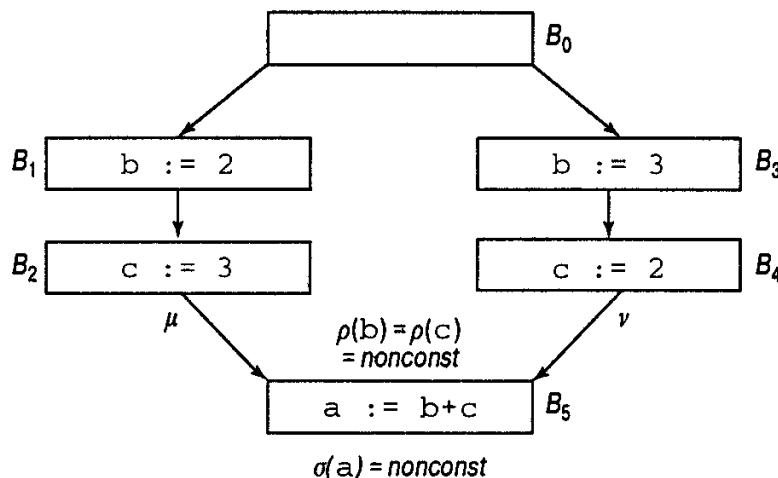


Рис. 10.63. Пример решения, меньшего тор-решения

Интуитивно проблема заключается в том, что алгоритм 10.18, работая с недистрибутивной схемой, ведет себя так, как если бы некоторые последовательности узлов, не являющиеся даже видимыми путями (путями в графе потока), были при этом реальными путями. Таким образом, на рис. 10.63 алгоритм ведет себя так, как если бы пути типа $B_0 \rightarrow B_1 \rightarrow B_4 \rightarrow B_5$ или $B_0 \rightarrow B_3 \rightarrow B_2 \rightarrow B_5$ были реальными путями, присваивающими переменным b и c комбинации значений, сумма которых отлична от 5. \square

Сходимость алгоритма 10.18

Имеются различные способы доказательства сходимости алгоритма 10.18 для конкретных схем. Вероятно, наиболее общий случай — когда требуются только ациклические пути, т.е. мы можем показать, что слияние по всем ациклическим путям эквивалентно *тор-решению* по всем путям. В этом случае алгоритм не только сходится, но и делает это весьма быстро, с максимальным количеством проходов, превышающим значение глубины графа на 2, как говорилось в разделе 10.10.

Однако схемы наподобие нашего примера вычисления констант требуют рассмотрения не только ациклических путей. Например, на рис. 10.64 показан простой график, в котором мы должны рассмотреть путь $B_1 \rightarrow B_2 \rightarrow B_2 \rightarrow B_3$ для определения того, что при входе в блок B_3 значение переменной x не является константой.

Однако для вычисления констант сходимость алгоритма 10.18 мы можем обосновать следующим образом. Прежде всего, для произвольной монотонной схемы легко показать, что значения $in[B]$ и $out[B]$ для любого блока B образуют невозрастающую после-

довательность в том смысле, что новое значение этих переменных всегда \leq старого значения. Если мы вспомним решеточную диаграмму значений переменных на рис. 10.60, то увидим, что для любой переменной значение $in[B]$ или $out[B]$ может спуститься по диаграмме дважды — от *undef* до константы и от константы до *nonconst*.

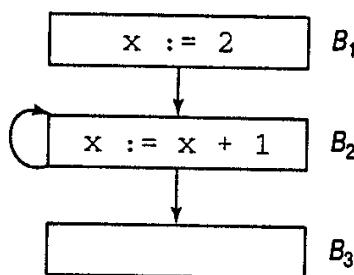


Рис. 10.64. Граф потока, требующий включения циклического пути в топ-решение

Предположим, что у нас имеется n узлов и v переменных. Тогда при каждой итерации цикла `while` на рис. 10.62 в некотором $out[B]$ должно спускаться значение как минимум одной переменной, иначе алгоритм сходится и даже бесконечное число итераций цикла не изменит значения in и out . Следовательно, количество итераций ограничено величиной $2nv$; если произошло такое количество изменений, значит, каждая переменная в каждом блоке графа потока достигла значения *nonconst*.

Исправление инициализации

В некоторых задачах потока данных имеется расхождение между получаемым по алгоритму 10.18 решением и тем, что мы интуитивно ожидаем. Вспомним, что для доступных выражений \wedge представляет собой пересечение, так что T должно быть множеством всех выражений. Поскольку в алгоритме 10.18 изначально предполагается, что $in[B] = T$ для каждого блока B , включая начальный узел, то *топ-решение*, полученное с помощью алгоритма 10.18, в действительности является множеством выражений, которые, если считать их доступными в начальном узле (где они, естественно, недоступны), будут доступны и при входе в блок B .

Отличие заключается в том, что из начального узла в B могут вести пути, на которых выражение $x+y$ не генерируется и не уничтожается. Алгоритм 10.18 будет считать выражение $x+y$ доступным, хотя на самом деле это не так, поскольку вдоль таких путей нет переменных, хранящих значение данного выражения. Исправление этой ситуации достаточно простое. Мы можем либо изменить алгоритм 10.18 так, чтобы в случае схемы доступных выражений $in[B_0]$ устанавливалось равным пустому множеству, либо изменить сам график потока введением искусственного начального узла, предшествующего реальному, который уничтожает все выражения.

10.12. Оценка типов

Теперь мы подошли к задаче потока данных, более сложной по сравнению со схемами из предыдущего раздела. Многие языки, от APL и SETL до различных диалектов Lisp, не требуют объявления типов переменных и даже позволяют одной и той же переменной хранить в разное время значения разных типов. При серьезных попытках компиляции таких языков в эффективный код использовался анализ потока данных для выяснения типов переменных, поскольку, например, код для сложения двух целых переменных существенно

более эффективен, чем вызов обобщенной процедуры, способной суммировать объекты разных типов (например, целые, действительные числа, вектора и т.д.).

Наша первая идея может заключаться в том, что типы переменных вычисляются так же, как и достигающие определения. Мы можем связать с каждой переменной в каждой точке множество возможных типов, поскольку если переменная x имеет множество S_1 возможных типов на одном пути и множество S_2 на другом пути, то после слияния этих путей x может иметь любой тип из множества $S_1 \cup S_2$. При прохождении управления через инструкцию мы можем оказаться способны сделать определенные выводы о типах переменных, основываясь на используемых в инструкции операторах, возможных типах их operandов и возможных типах получаемого результата. Пример 6.6, где мы встретились с оператором, который может перемножать как целые, так и комплексные числа, был примером выводения информации такого рода.

К сожалению, при таком подходе возникают как минимум две проблемы.

1. Множество различных типов переменной может оказаться бесконечным.
2. Определение типа обычно требует как прямого, так и обратного распространения информации для получения точных оценок возможных типов. Следовательно, даже схемы из раздела 10.11 недостаточно общие для разрешения этой задачи.

Перед тем как перейти к п. (1), рассмотрим некоторые варианты выводов о типах, которые могут быть сделаны в знакомых нам языках.

Пример 10.50

Рассмотрим инструкции

```
i := a[j]
k := a[i]
```

Предположим сначала, что мы ничего не знаем о типах переменных a , i , j и k , но знаем, что оператор обращения к массиву $[]$ требует целого аргумента. Рассмотрение первой инструкции позволяет сделать вывод о том, что в данной точке j — целое число, а a — массив элементов некоторого типа. Затем вторая инструкция говорит нам, что i — также целое число.

Теперь мы можем распространить вывод в обратном направлении. Если i , вычисляемое в первой инструкции, имеет целый тип, то тип выражения $a[i]$ также целый, а значит, тип a — массив целых чисел. Теперь мы можем продолжить рассмотрение в прямом направлении и прийти к выводу, что тип k — также целый. Заметим, что, работая только в одном направлении, мы не смогли бы определить, что элементы массива a — целые числа. \square

Работа с бесконечными множествами типов

Имеются многочисленные примеры патологических случаев, когда множество возможных типов переменной бесконечно. Например, в SETL допустимы выполняемые в цикле инструкции вида

```
x := { x }
```

Если начать с информации, что x может быть целым, то после первой итерации цикла выясняется, что x может быть как целым, так и множеством целых чисел. После второй итерации к возможным типам x добавляется множество множеств целых чисел и т.д.

Схожая проблема может возникнуть в нетипизированной версии обычного языка типа C, где инструкция

`x = &x`

с начальным предположением о возможности целого типа `x` приводит нас к выводу о том, что `x` может иметь любой тип вида “указатель на указатель на . . . указатель на целое число”.

Традиционное решение этой проблемы состоит в приведении множества возможных типов к конечному. Основная идея заключается в группировании бесконечного числа возможных типов в конечное количество классов; простые типы при этом обычно остаются нетронутыми, а наиболее сложные (и, надо полагать, наиболее редко встречающиеся) типы группируются в большие классы. При этом мы должны критически оценивать выводы о взаимодействии типов и операторов. Следующий пример показывает, что и как может быть сделано в этом направлении.

Пример 10.51

Продолжим пример из главы 6, “Проверка типов”, где оператор \rightarrow использовался нами в качестве конструктора типа для функций. В данном примере наше множество типов включает базовый тип `int` и все типы вида $\tau \rightarrow \sigma$, представляющие тип функции с областью определения τ и областью значений σ , где τ и σ являются типами из нашего множества. Таким образом, наше множество типов бесконечно и включает такие типы, как

$(int \rightarrow int) \rightarrow ((int \rightarrow int) \rightarrow int)$

Для приведения этого множества к конечному количеству классов мы ограничим выражения типов только одним конструктором типа функции \rightarrow , заменяя подвыражения с оператором \rightarrow в выражениях типа именем `func`. Таким образом, у нас имеется пять различных типов:

`int`
`int → int`
`int → func`
`func → int`
`func → func`

Представим множества типов в виде битовых векторов длиной 5 бит, позиции которых соответствуют пяти типам в приведенном выше порядке. Так, `01111` представляет любую функцию, т.е. все, кроме простого `int`. Заметим, что, по сути, смысл этого типа — `func`, поскольку `func` не может быть целым.

Базовая инструкция присвоения в нашей модели имеет вид

`x := f(y)`

Зная возможные типы `f` и `y`, мы можем определить возможные типы `x` путем поиска типа в таблице на рис. 10.65. Если `f` может быть любого типа из множества S_1 , а `y` — любого типа из S_2 мы берем все пары $\tau \in S_1$ и $\sigma \in S_2$, и смотрим на запись на пересечении строки τ и столбца σ , которую назовем $\tau(\sigma)$. Затем для получения множества возможных типов `x` мы берем объединение всех найденных записей.

Например, если $\tau = int \rightarrow func$ и $\sigma = int$, то $\tau(\sigma) = 01111$, т.е. результат применения отображения типа `int → func` к `int` дает `func`, что означает отображение на любой из четы-

рех типов, кроме *int*. Мы не можем сказать, на какой именно, из-за распределения бесконечного числа типов по пяти классам.

τ	σ				
	<i>int</i>	<i>int</i> \rightarrow <i>int</i>	<i>int</i> \rightarrow <i>func</i>	<i>func</i> \rightarrow <i>int</i>	<i>func</i> \rightarrow <i>func</i>
<i>int</i>	00000	00000	00000	00000	00000
<i>int</i> \rightarrow <i>int</i>	10000	00000	00000	00000	00000
<i>int</i> \rightarrow <i>func</i>	01111	00000	00000	00000	00000
<i>func</i> \rightarrow <i>int</i>	00000	10000	10000	10000	10000
<i>func</i> \rightarrow <i>func</i>	00000	01111	01111	01111	01111

Рис. 10.65. Значения $\tau(\sigma)$

В качестве второго примера рассмотрим то же значение τ , что и раньше, и $\sigma = \text{int} \rightarrow \text{int}$. В этом случае $\tau(\sigma) = 00000$, поскольку тип области определения τ гарантированно не равен типу σ , и такое отображение неприменимо. \square

Простая система типов

Для иллюстрации идей, лежащих в основе алгоритмов вывода типов, мы введем простую систему типов и язык, основанные на примере 10.51. В этой системе пять только что рассмотренных типов, а инструкции нашего языка подразделяются на три вида.

1. `read x`. Значение x считывается из входного потока, и о типе x при этом ничего не известно.
2. `x := f(y)`. Значение x представляет собой множество, получаемое путем применения функции f к значению y . Информация о типе x после присвоения приведена в таблице на рис. 10.65.
3. `use x as t`. При выполнении этой инструкции мы полагаем, что если программа корректна, то типом x может быть только t — как до, так и после выполнения инструкции. Значение и тип переменной x данной инструкцией не изменяются.

Выводы о типах мы делаем, выполняя анализ потока данных на основе графа потока программы, состоящей из инструкций трех указанных видов. Для упрощения мы считаем, что каждый блок состоит лишь из одной инструкции. Значения *in* и *out* для блоков представляют собой отображения переменных в множества из пяти типов из примера 10.51.

Изначально каждое *in* и *out* отображают каждую переменную в множество, состоящее из всех пяти типов. По мере распространения информации мы уменьшаем множества типов, связанные с различными переменными в определенных точках, до тех пор, пока не достигнем состояния, когда дальнейшее уменьшение множеств становится невозможным. Мы считаем, что полученные множества определяют возможные типы каждой переменной в каждой точке. Такое предположение консервативно, поскольку тип удаляется из множества возможных только в том случае, когда (для корректно работающей программы) доказывается его невозможность. Поскольку обычно нас интересует тот факт, что определенные типы данной переменной невозможны, ошибки в сторону увеличения множества возможных типов являются безопасными.

Для изменения множеств *in* и *out* мы применяем две схемы — прямую и обратную. Прямая схема использует инструкцию в блоке *B* и значение *in[B]* для ограничения мно-

жества $out[B]$ ²⁰, а обратная работает, как и следует из ее названия, в обратном направлении. В каждой из схем оператор слияния представляет собой “покомпонентное объединение” в том смысле, что слияние двух отображений α и β — это отображение γ , такое, что для всех переменных x (компонентов) $\gamma[x] = \alpha[x] \cup \beta[x]$.

Прямая схема

Предположим, что у нас имеется блок B с $in[B]$ (отображение μ) и $out[B]$ (отображение ν). В случае прямой схемы мы ограничиваем ν , и правила этого ограничения, естественно, зависят от того, какая именно инструкция находится в блоке B .

- Если эта инструкция — `read x`, то может быть прочитан любой тип. Если нам уже что-то известно о типе x после чтения, мы должны не потерять эту информацию при прямом проходе — так что мы просто не изменяем $\nu(x)$ при прямом проходе. Для всех остальных переменных y мы устанавливаем $\nu(y) := \nu(y) \cap \mu(y)$.
- Теперь предположим, что нам встретилась инструкция `use x as t`. После этой инструкции единственным возможным типом для x является t . Если к этому моменту мы уже знаем, что переменная x не может иметь тип t , следовательно, после данной инструкции для переменной x возможных типов нет. Это наблюдение можно подытожить как

$$\begin{aligned}\nu(x) &:= \nu(x) \cap \{t\} \\ \nu(y) &:= \nu(y) \cap \mu(y) \text{ для } y \neq x\end{aligned}$$

- Теперь рассмотрим случай инструкции `x := f(y)`. После такого присвоения единственными возможными типами для x являются
 - типы, возможные в соответствии с текущим значением ν ,
 - типы, являющиеся результатом применения отображения некоторого типа t на тип σ ; типы t и σ являются соответственно типами, возможными для f и y перед выполнением инструкции.

Формально, $\nu(x) := \nu(x) \cap \{\rho \mid \rho = t(\sigma), t \in \mu(f), \sigma \in \mu(y)\}$.

Мы можем также сделать определенные выводы о типах f и y , поскольку из предположения о корректности программы следует, что f не может иметь тип, не применимый хотя бы к одному из возможных типов своего аргумента y , а переменная y не может иметь тип, который не может служить типом аргумента функции, тип которой — один из возможных типов f . Другими словами, если $f \neq x$, то

$$\nu(f) := \nu(f) \cap \{t \in \mu(f) \mid \text{для некоторого } \sigma \in \mu(y), t(\sigma) \neq \emptyset\}$$

Если $y \neq x$, то

²⁰ Стоит отметить, что в традиционных прямых схемах потока данных мы не ограничиваем out , вычисляя вместо этого in каждый раз заново. Мы можем так поступать, поскольку in и out всегда изменяются в одном направлении — либо увеличиваясь, либо уменьшаясь. Однако в задачах наподобие вывода типов, где поочередно выполняются проходы в прямом и обратном направлениях, мы можем встретиться с ситуацией, когда обратный проход сделает out гораздо меньше, чем мы можем ограничить его путем применения прямых правил к in . Таким образом, мы не должны случайно увеличить out при прямом проходе, и можем только уменьшать его при обратном (хотя, вероятно, и не настолько сильно). Подобное примечание справедливо и к обратным проходам: мы должны ограничивать in , а не вычислять его заново.

$$v(y) := v(y) \cap \{\sigma \in \mu(y) \mid \text{для некоторого } \tau \in \mu(f), \tau(\sigma) \neq \emptyset\}$$

Для всех остальных z

$$v(z) := v(z) \cap \mu(z)$$

Обратная схема

Рассмотрим теперь, каким образом при обратном проходе мы можем ограничить μ на основании информации о v и рассматриваемой инструкции.

- Если рассматриваемая инструкция — `read x`, то, как легко увидеть, никаких новых выводов о невозможности типов перед данной инструкцией сделать нельзя, так что $\mu(x)$ остается неизменным. Однако для всех $y \neq x$ мы можем распространить информацию в обратном направлении установкой $\mu(y) := \mu(y) \cap v(y)$.
- В случае инструкции `use x as t` мы можем сделать те же выводы, что и в случае прямого распространения: x перед инструкцией может иметь только тип t , типы всех остальных переменных могут быть только теми, которые переменные могут принимать и до, и после данное инструкции, т.е.

$$\mu(x) := \mu(x) \cap \{t\}$$

$$\mu(y) := \mu(y) \cap v(y) \text{ для } y \neq x$$

- Как и ранее, наиболее сложным случаем является случай инструкции вида $x := f(y)$. Для начала заметим, что если x — не f и не y , то мы не можем сделать никакого нового вывода о типе x до присвоения. Таким образом, $\mu(x)$ не изменяется, кроме как по правилам, касающимся f и y . Далее, как и в прямой схеме, мы можем сделать выводы на основе того факта, что в корректной программе типы f и y должны быть совместимы перед выполнением инструкции. Однако если $f \neq x$, мы можем также ограничить $\mu(f)$ типами из $v(f)$ (аналогичное утверждение справедливо и для y). Однако если $f = x$, то после инструкции типы f никак не связаны с типами f до инструкции, так что такие ограничения не допустимы (вновь аналогичное утверждение справедливо и при $y = x$). Для отражения этого решения полезно определить специальное отображение, действующее только для f и y . Итак, определим:

если $f = x$, то $\mu_1(f) := \mu(f)$; иначе $\mu_1(f) := \mu(f) \cap v(f)$

если $y = x$, то $\mu_1(y) := \mu(y)$; иначе $\mu_1(y) := \mu(y) \cap v(y)$

Теперь мы можем ограничить f и y типами, совместимыми друг с другом. В то же время мы можем ограничить типы f и y , основываясь на том, что они должны быть не только совместимы, но и должны приводить к типам, которые может принимать x в соответствии со значением v . Итак, мы определяем следующее.

$$\mu(f) := \{\tau \in \mu_1(f) \mid \tau(\sigma) \cap v(x) \neq \emptyset \text{ для некоторого } \sigma \in \mu_1(y)\}$$

$$\mu(y) := \{\sigma \in \mu_1(y) \mid \tau(\sigma) \cap v(x) \neq \emptyset \text{ для некоторого } \tau \in \mu_1(f)\}$$

$$\mu(z) := \mu(z) \cap v(z) \text{ для } z, \text{ отличающегося от } x, y \text{ и } f$$

Перед тем как приступить к алгоритму определения типов, вспомним из обсуждения вопроса о достигающих определениях в разделе 10.5, что если мы начнем с ложного предположения о том, что некоторое определение d доступно в некоторой точке цикла, то можем ошибочно распространить этот факт на весь цикл, получив в конечном счете большее множество достигающих определений, чем необходимо. Аналогичная проблема

может возникнуть и в случае определения типов, где предположение о том, что переменная может иметь некий тип, “доказывает” само себя при проходе по циклу. Поэтому мы должны ввести 33-е значение в дополнение к 32 множествам типов из примера 10.51, которое отображение μ может назначить переменной, а именно значение *undef*. Использование *undef* аналогично его использованию в случае схемы распространения констант из предыдущего раздела.

В процессе слияния значение *undef* “уступает” любому другому значению, т.е. действует подобно типу 00000. Однако при пересечении множеств типов, т.е. вычислении $\mu(x) \cap v(x)$ значение *undef* также “уступает” любому другому множеству типов, действуя аналогично типу 11111. Таким образом, например, когда мы считываем значение переменной x , то при этом тот факт, что “тип” x считался равным *undef*, перестает действовать и тип x становится равным 11111.

Алгоритм 10.19. Вычисление множества типов

Вход. Граф потока, блоки которого представляют собой отдельные инструкции одного из трех типов (чтение, присвоение и *use as*), упомянутых ранее.

Выход. Множество типов для каждой переменной в каждой точке графа. Эти множества консервативны в том смысле, что любые реальные вычисления должны привести к типу, имеющемуся в множестве.

Метод. Для каждого блока B мы вычисляем отображения $in[B]$ и $out[B]$. Каждое из них отображает переменные программы в множества типов из системы типов, описанной в примере 10.51. Изначально каждая переменная отображается в значение *undef*.

Затем мы поочередно выполняем прямые и обратные проходы по графу потока, пока очередная пара проходов приводит к изменениям отображений. Прямой проход выполняется следующим образом.

```
for Каждый блок  $B$  в порядке вглубь do begin
     $in[B] := \bigcup_{P-\text{пред-}шественник  $B$ } out[P];$ 
     $out[B] :=$  функция от  $in[B]$  и  $out[B]$ , определенная ранее
end
```

Обратный проход выглядит следующим образом.

```
for Каждый блок  $B$  в обратном порядке вглубь do begin
     $out[B] := \bigcup_{S-\text{преемник } B} in[S];$ 
     $in[B] :=$  функция от  $in[B]$  и  $out[B]$ , определенная ранее
end
```

□

Пример 10.52

Рассмотрим простую “прямолинейную” программу, показанную на рис. 10.66. Нас интересуют четыре отображения, обозначенные на рисунке как μ_1 – μ_4 . Каждое μ , представляет собой как $out[B_i]$, так и $in[B_{i+1}]$. Технически блок B_1 не должен состоять из двух инструкций, но поскольку нас не интересует, что происходит перед концом блока B_1 , где все переменные могут быть любого типа, такой блок в нашем случае вполне допустим.

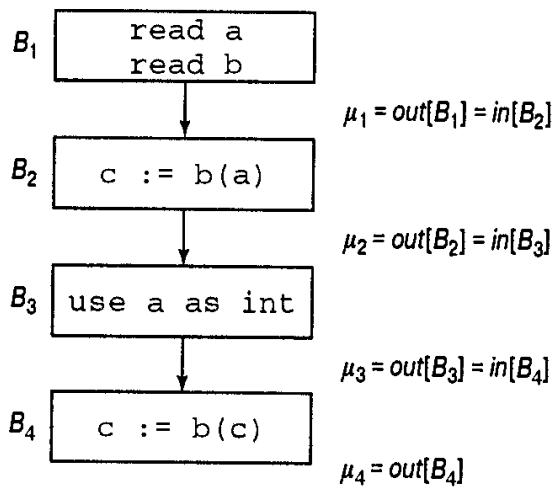


Рис. 10.66. Пример программы

Мы увидим, что нам необходимы пять проходов до того, как результат сойдется, и еще два — для выяснения этого факта. Эти проходы подытожены в таблицах на рис. 10.67. Первый проход — прямой. При рассмотрении блока B_2 мы находим, что b не может быть целым, поскольку используется в качестве отображения. Мы также выясняем, что a используется в B_3 как целое и, таким образом, в μ_3 и μ_4 может быть отображено только в int . Все эти наблюдения подытожены на рис. 10.67а.

	a	b	c
μ_1	11111	11111	<i>undef</i>
μ_2	11111	01111	11111
μ_3	10000	01111	11111
μ_4	10000	01111	11111

а) Прямой проход

	a	b	c
μ_1	10000	01100	<i>undef</i>
μ_2	10000	01100	11111
μ_3	10000	01100	11111
μ_4	10000	01100	11111

в) Прямой проход

	a	b	c
μ_1	10000	01100	<i>undef</i>
μ_2	10000	01111	11111
μ_3	10000	01111	11111
μ_4	10000	01111	11111

б) Обратный проход

	a	b	c
μ_1	10000	01000	<i>undef</i>
μ_2	10000	01100	10000
μ_3	10000	01100	10000
μ_4	10000	01100	11111

г) Обратный проход

	a	b	c
μ_1	10000	01000	<i>undef</i>
μ_2	10000	01000	10000
μ_3	10000	01000	10000
μ_4	10000	01000	10000

д) Прямой проход

Рис. 10.67. Моделирование алгоритма 10.19 для графа потока на рис. 10.66

Второй проход, показанный на рис. 10.67б, — обратный. При этом проходе при рассмотрении B_2 мы знаем, что a должно быть целым при применении к нему b , а значит, b может иметь только тип $int \rightarrow int$ или $int \rightarrow func$. При третьем (прямом) проходе это ограничение на b распространяется вниз по графу, как показано на рис. 10.67в.

Четвертый (обратный) проход показан на рис. 10.67г. Здесь тот факт, что в блоке B_3 с является аргументом b , говорит нам о том, что c может иметь только один тип — int . Рассматривая B_2 , мы обнаруживаем, что результат $b(a)$ имеет тот же тип, что и c , и этот факт исключает для b возможность иметь тип $int \rightarrow func$. И наконец, на рис. 10.67д мы видим распространение этих фактов о типах b и c . Дальнейшие проходы ничего нового в наш результат не вносят. В рассмотренном нами примере мы сократили множества возможных типов для каждой переменной в каждой точке до одного единственно возможного типа: a и c — целые числа, а b — отображение целых чисел на целые числа. В общем же случае для переменной в некоторой точке могут оказаться возможными несколько типов. \square

10.13. Символьная отладка оптимизированного кода

Символьный отладчик (symbolic debugger) представляет собой систему, которая позволяет нам просматривать данные программы в процессе ее выполнения. Обычно отладчик вызывается при ошибке в программе (например, при возникновении переполнения) или при достижении некоторой инструкции исходного кода, указанной программистом. После вызова символьный отладчик позволяет программисту просмотреть, а возможно, и изменить любую переменную, доступную в данный момент работающей программе.

Для того чтобы команда пользователя типа “подать сюда значение переменной a !” была понятна отладчику, ему должна быть доступна определенная информация.

1. Нужно иметь способ связи идентификатора типа a с ячейкой памяти, которую он представляет. Таким образом, та часть таблицы символов, которая назначает каждой переменной местоположение в памяти, т.е. место в глобальной области данных или в записи активации некоторой процедуры, должна быть записана компилятором и сохранена для использования отладчиком. Например, такая информация может быть закодирована в загрузочном модуле программы.
2. Нужна также информация об области видимости, чтобы мы могли устраниТЬ неоднозначности при обращении к идентификатору, объявленному более одного раза, и чтобы для данной процедуры p могли сказать, какие данные этой процедуры доступны и как найти их в стеке или другой структуре времени исполнения. Опять-таки, эта информация должна быть взята из таблицы символов компилятора и сохранена для последующего использования отладчиком.
3. Мы должны знать текущее положение в программе при вызове отладчика. Эта информация передается компилятором при обработке вызова отладчика пользователем. В случае ошибки времени исполнения, вызывающей отладчик, эта информация обеспечивается обработчиком ошибок.
4. Для того чтобы информация о положении в программе, указанная в (3), имела смысл для пользователя, нужна таблица связи машинных инструкций с инструкциями ис-

ходной программы. Такая таблица может быть подготовлена компилятором в процессе генерации кода.

Хотя задача разработки отладчика интересна сама по себе, здесь мы рассмотрим лишь сложности, возникающие при попытках написания символьного отладчика для оптимизирующего компилятора. На первый взгляд кажется, что нет необходимости отлаживать оптимизированную программу. При нормальном цикле разработки программного обеспечения в процессе отладки программы пользователем, пока последний не будет удовлетворен полученным результатом, используется быстрый неоптимизирующий компилятор. И только после этого используется оптимизирующий компилятор.

К сожалению, программа может корректно работать при компиляции неоптимизирующим компилятором, а затем, при тех же входных данных, будучи скомпилированной оптимизирующим компилятором, может давать сбои. Например, в оптимизирующем компиляторе может быть ошибка, либо в результате переупорядочения операций может происходить переполнение или потеря значимости. Кроме того, даже “неоптимизирующий” компилятор может выполнять некоторые простые преобразования, типа устранения локальных общих подвыражений или переупорядочения кода внутри базового блока, что приводит к усложнению задачи создания символьического отладчика. Таким образом, мы должны рассмотреть алгоритмы и структуры данных, используемые в символьных отладчиках для оптимизирующих компиляторов, которые произвольным образом изменяют базовые блоки.

Отслеживание значений переменных в базовых блоках

Для простоты предположим, что и исходный, и объектный коды представляют собой последовательности инструкций промежуточного представления. Рассмотрение исходного текста как промежуточного кода не представляет проблем, поскольку промежуточный код является более обобщенным. Например, пользователь может разместить точки останова (вызовы отладчика) только между инструкциями в исходном тексте, в то время как в случае промежуточного кода мы можем поместить точки останова после любой промежуточной инструкции. Рассмотрение объектного кода как промежуточного сомнительно только в том случае, когда оптимизатор разбивает единые промежуточные инструкции на несколько отдельных машинных инструкций. Например, по некоторым причинам мы можем скомпилировать две промежуточные инструкции

```
u := v + w  
x := y + z
```

в код, в котором два сложения выполняются в различных регистрах и соответствующие им машинные инструкции чередуются. В таком случае мы можем рассматривать загрузки и сохранения регистров, как если бы регистры представляли собой временные переменные в промежуточном коде, например:

```
r1 := v  
r2 := y  
r1 := r1 + w  
r2 := r2 + z  
u := r1  
x := r2
```

При работе пользователя с данным блоком из-за того, что пользователь полагает, что выполняется исходный блок, в то время как выполняется оптимизированная его версия, возникает ряд проблем.

- Предположим, что мы выполняем программу, которая является результатом “оптимизации” некоторого базового блока исходной программы, и в процессе выполнения инструкции $a := b+c$ происходит переполнение. Мы должны сообщить пользователю о том, что в одной из исходных инструкций произошла ошибка. Поскольку $b+c$ является общим подвыражением, появляющимся в двух или большем количестве исходных инструкций, — к какой из них отнести данную ошибку?
- Ещё более сложная проблема возникает, когда пользователь хочет просмотреть “текущее” значение некоторой переменной d . В оптимизированной программе переменной d последний раз значение было присвоено в инструкции s , однако в исходной программе s может оказаться после инструкции, в которой вызван отладчик; так что значение d , доступное отладчику, оказывается совсем не тем, которое является “текущим” в соответствии с исходным текстом программы. Аналогично инструкция s может и предшествовать вызову отладчика, но при этом в исходном коде между этими моментами присутствует еще одно присвоение переменной d , и потому пользователю указывается устаревшее значение переменной. Можно ли каким-то образом сделать доступным пользователю корректное значение переменной? Например, не может ли оно оказаться в оптимизированной версии значением некоторой другой переменной или нельзя ли получить его путем вычисления на основе значений других переменных?
- И наконец, если пользователь размещает точку останова после некоторой инструкции исходного кода, то в какой момент следует передать управление отладчику при выполнении оптимизированного кода?

Одно из решений состоит в запуске неоптимизированной версии блока вместе с оптимизированной версией, для того чтобы в каждый момент времени иметь возможность доступа к корректным значениям каждой переменной. Мы отказываемся от такого “решения”, поскольку при этом очень трудно находить ошибки, в особенности внесенные компилятором.

Приемлемое решение состоит в обеспечении отладчика достаточным количеством информации о каждом блоке, чтобы он мог ответить, по крайней мере, на вопрос о том, возможно ли предоставить пользователю корректное значение переменной a , и если да, то как. Используемая для этой информации структура данных представляет собой даг базового блока, аннотированный информацией о том, какие переменные и какое время хранят значения, соответствующие узлам дага — как для исходной, так и для оптимизированной программы. Связанная с узлом запись

$a: i - j$

означает, что значение, представленное этим узлом, хранится в переменной a от начала инструкции i вплоть до момента в инструкции j непосредственно перед присвоением переменной a . Если $j = \infty$, то a хранит это значение до конца блока.

Пример 10.53

На рис. 10.68 a мы видим базовый блок исходного кода, а на рис. 10.68 b — одну из возможных версий оптимизации этого кода. На рис. 10.69 представлен даг для этих блоков с указанием диапазонов хранения значений переменными как в исходном, так и оптимизированном коде. Штрихи на рисунке указывают диапазоны инструкций в оптимизированном коде. Например, узел, помеченный +, представляет значение c в ис-

ходном коде от начала инструкции (2) до присвоения в инструкции (3). Он также представляет значение d в исходном коде от начала инструкции (3) до конца блока. Кроме того, тот же узел представляет значение d в оптимизированном коде от инструкции (2') до конца блока. \square

- (1) $c := a + b$
- (2) $d := c$
- (3) $c := c - e$
- (4) $a := d - e$
- (5) $b := b * e$
- (6) $b := d / b$

(a)

- (1') $d := a + b$
- (2') $t := b * e$
- (3') $a := d - e$
- (4') $b := d / t$
- (5') $c := a$

(б)

Рис. 10.68. Исходный и оптимизированный коды

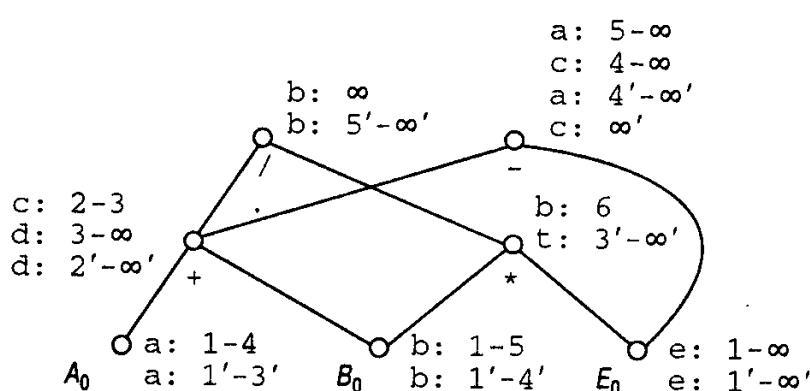


Рис. 10.69. Аннотированный даг

Теперь мы можем ответить на первый поставленный выше вопрос. Предположим, что ошибка, например переполнение, произошла при выполнении инструкции j' оптимизированного кода. Поскольку одно и то же значение будет вычислено любой исходной инструкцией, вычисляющей узел j' дага, имеет смысл сообщить пользователю об ошибке в первой исходной инструкции, вычисляющей данный узел. Таким образом, в примере 10.53, если ошибка происходит в инструкциях 1', 2', 3' или 4', мы сообщаем об ошибках в инструкциях соответственно 1, 5, 3 и 6. В инструкции 5' никаких ошибок возникнуть не может, так как в ней не вычисляется никакое выражение. Рассмотрение деталей вычисления соответствующих инструкций мы отложим до примера 10.54.

Мы можем ответить и на второй вопрос. Предположим, что мы находимся на инструкции j' оптимизированного кода, а пользователю сообщается, что управление находится на i -й инструкции исходного кода, где произошла ошибка. Если пользователь хочет увидеть значение переменной x , мы должны найти переменную y (зачастую, хотя и не всегда, y есть x), такую, что значение x в инструкции i исходного кода является тем же узлом дага, что и y в инструкции j' оптимизированного кода. Мы просматриваем даг в поисках узла, представляющего значение x в инструкции i , и можем узнать из этого узла все переменные объектной программы, имеющие данное значение, чтобы определить, какая из них хранит интересующее нас значение в инструкции j' .

Если мы находим такую переменную, задача решена; если нет — мы можем попытаться вычислить значение переменной x в исходной инструкции i на основании других переменных в j' . Пусть n — узел, соответствующий x в момент i . Тогда мы можем рассмотреть дочерние по отношению к n узлы, скажем m и p , чтобы узнать, не представляют ли они значение некоторой переменной в момент j' . Если, скажем, такая переменная имеется для m , но не для p , то мы можем рекурсивно рассмотреть потомков p . В конечном счете мы либо найдем путь вычисления значения x в момент i , либо сделаем вывод, что такого пути нет. Если мы находим путь вычисления значений m и p , то мы вычисляем их и применяем оператор в узле n для вычисления значения x в момент i ²¹.

Пример 10.54

Предположим, что при выполнении кода на рис. 10.68б в инструкции 2' происходит ошибка. Инструкция вычисляет узел, помеченный * на рис. 10.69, и первая инструкция исходного кода, которая вычисляет это значение, — это инструкция 5. Следовательно, мы должны сообщить о наличии ошибки в инструкции 5.

На рис. 10.70 показаны узлы дага, соответствующие каждой переменной в исходном и оптимизированном коде в начале инструкций 5 и 2'. Узлы указаны своими метками — либо символами операторов, либо символами начальных значений наподобие A_0 . Здесь также показано, как можно вычислить значение в момент 5 по значениям переменных в момент 2'. Например, если пользователь запрашивает значение a , он получает значение узла, помеченного оператором $-$. В момент 2' ни одна из переменных не содержит это значение, но, к счастью, имеются переменные d и e , хранящие значения каждого из потомков узла $-$ в момент 2', так что мы можем вывести значение a , вычисляя значение $d-e$. \square

ПЕРЕМЕННАЯ	ЗНАЧЕНИЕ В		ПОЛЧАЕТСЯ
	МОМЕНТ 2'	МОМЕНТ 5	
a	A_0	$-$	$d-e$
b	B_0	B_0	b
c	Не определено	$-$	$d-e$
d	$+$	$+$	d
e	E_0	E_0	e
t	Не определено		

Рис. 10.70. Значения переменных в моменты 2' и 5

Ответим теперь на третий вопрос — каким образом следует обрабатывать вызов отладчика, указанный пользователем. В некотором смысле ответ тривиален: если пользователь запрашивает останов после инструкции i исходной программы, мы можем прекратить выполнение программы в начале блока. Если пользователя интересует значение некоторой переменной после выполнения инструкции i , мы можем обратиться к анноти-

²¹ Имеется одна тонкость в том, что вычисление значения в узле i может вызвать другую ошибку. В таком случае мы должны сообщить пользователю, что в действительности ошибка произошла ранее, в первом операторе исходного кода, вычисляющем значение i .

рованному дагу и выяснить, какой узел представляет интересующее нас значение x , после чего вычислить это значение из начальных значений переменных для этого блока.

В то же время, отложив, насколько это возможно, вызов отладчика, мы сможем оставить для него меньшее количество работы, а также избежать некоторых ситуаций, когда попытки вычисления значений приводят к ошибкам, о которых следует сообщить пользователю. Достаточно просто вычислить последнюю перед вызовом отладчика инструкцию j' оптимизированной программы, после чего вызываем отладчик и сообщаем пользователю, что вызов был сделан после i -й инструкции исходной программы. Для того чтобы определить j' , обозначим через S множество узлов дага, соответствующих значению некоторой переменной исходной программы непосредственно после инструкции i . Пользователь может запросить значение любой переменной из S . Таким образом, мы можем сделать останов после инструкции j' оптимизированного кода, только если для каждого узла $n \in S$ существует некоторое $k' > j'$, такое, что с узлом n в момент k' в оптимизированном коде связана некоторая переменная. В таком случае нам известно, что значение n либо доступно непосредственно после инструкции j' , либо может быть вычислено после нее. В первом случае, при останове после j' вычисление узла n тривиально; во втором случае нам известно, что доступных после j' значений достаточно для вычисления n каким-либо образом.

Пример 10.55

Вновь рассмотрим исходный и оптимизированный код на рис. 10.68, и предположим, что пользователь вставил точку останова после инструкции (3) исходного кода. Для того чтобы найти множество S , рассмотрим даг на рис. 10.69 и найдем, какие узлы содержат переменные исходной программы, назначенные им в момент (4). Это узлы, помеченные на рисунке как A_0 , B_0 , E_0 , $+$ и $-$.

Теперь взглянем на даг еще раз и найдем наибольшее j' , такое, что каждый из узлов из S содержит связанную с ним некоторую переменную оптимизированного кода в момент, строго больший j' . Узлы, помеченные как $+$, $-$ и E_0 , не представляют проблем, поскольку их значения хранятся переменными соответственно d , a и e в момент ∞' . Узлы A_0 и B_0 ограничивают значение j' , и наименьшее из них соответствует потере значения A_0 в инструкции 3'. Таким образом, $j' = 2'$ — наибольшее возможное значение j' , т.е. если пользователь запрашивает останов после инструкции 3 исходного кода, мы должны реализовать этот останов после инструкции 2'. \square

Читатель должен заметить одну тонкость в примере 10.55, для которой нет действительно хорошего решения. Если мы выполним инструкцию 2' оптимизированного кода перед вызовом отладчика, ошибка в вычислении b^*e в инструкции 2' (например, потеря значимости результата) может привести к вызову отладчика до достижения определенной пользователем точки. Однако поскольку вычисления, соответствующие инструкции 2', в исходной программе не выполняются до инструкции 5, мы должны сообщить пользователю об ошибке в инструкции 5. Для пользователя может оказаться несколько странным, что при точке останова на инструкции 3 отладчик вдруг выводит сообщение об ошибке в инструкции 5. Вероятно, наилучшее решение этой проблемы состоит в том, чтобы не позволять j' быть настолько большим, что существует инструкция $k' \leq j'$ оптимизированного кода, такая, что исходный код вычисляет значение, вычисляемое инструкцией k' , после инструкции i , на которой установлена точка останова.

Влияние глобальной оптимизации

Когда наш компилятор выполняет глобальную оптимизацию, перед символьным отладчиком встают более сложные проблемы, и зачастую не существует способов найти корректное значение переменной в некоторой точке. Двумя важными преобразованиями, которые не вызывают особых проблем, являются устранения переменных индукции и глобальных общих подвыражений; в каждом из этих случаев проблема может быть локализована в нескольких блоках и решена способом, аналогичным рассмотренным ранее.

Устранение переменных индукции

Если мы устраним переменную индукции исходной программы i в пользу некоторого члена семейства i , скажем t , то в этом случае i и t связаны между собой некоторой линейной функцией. Кроме того, если мы следуем методам из раздела 10.7, то оптимизированный код будет изменять t в тех же блоках, где должно было бы изменяться i , так что это линейное соотношение справедливо всегда. Таким образом, после учета переупорядочения инструкций в блоке, присваивающем t (а в исходном коде — и i), мы можем предоставить пользователю “текущее” значение i , вычисленное как линейное преобразование t .

Мы должны быть осторожны, если i не определено до цикла, поскольку t обязательно определяется перед входом в цикл, и мы можем предоставить пользователю значение i в той точке программы, где значение i , по сути, не определено. К счастью, обычно переменные индукции в исходных программах определяются до цикла, и не определенными при входе в цикл остаются только генерируемые компилятором переменные (значения которых пользователь запросить не может). Если для некоторой переменной i это не так, то мы получаем проблему, похожую на ту, которая возникает при перемещении кода, о которой мы поговорим немного позже.

Устранение глобальных общих подвыражений

При устранении глобальных общих подвыражений для выражения $a+b$, мы воздействуем на ограниченное количество блоков достаточно простым способом. Если t — переменная, используемая для хранения значения $a+b$, то в некоторых блоках, вычисляющих $a+b$, мы можем заменить

$c := a + b$

кодом

$t := a + b$

$c := t$

Этот тип изменений кода может быть обработан уже рассмотренными методами работы с базовыми блоками.

В других блоках использование выражения типа $d := a+b$ заменяется использованием $d := t$. Для обработки этой ситуации предыдущими методами мы должны заметить, что в даге для этого блока значение t на все время остается значением узла для $a+b$ (который появляется в даге для исходного кода, но не должен возникать в даге для оптимизированного кода).

Перемещение кода

С прочими преобразованиями работать не так просто. Предположим, что мы вынесли инструкцию

$s: a := b+c$

из цикла в связи с тем, что она представляла собой инвариант относительно цикла. При вызове отладчика в цикле мы не знаем, выполнена ли уже инструкция s в исходной программе, и не можем сказать, является ли значение a тем, которое пользователь увидел бы в исходной программе.

Одна из возможностей состоит во вставке в оптимизированный код новой переменной, которая указывает в цикле, где выполнялось присвоение переменной a (что может быть только на старом месте инструкции s). Однако такая стратегия не всегда применима, поскольку для абсолютной надежности в отладчике должен использоваться только реальный код, а не созданный специально для отладки.

Однако имеется один частный случай, в котором наша задача упрощается. Предположим, что блок B , содержащий в исходной программе инструкцию s , разделяет цикл на два множества узлов: которые доминируют над ним и над которыми доминирует этот блок. Кроме того, предположим, что B доминирует над всеми предшественниками заголовка, как показано на рис. 10.71. Тогда при первом проходе по блокам, доминирующими над B , мы можем считать, что a еще не было присвоено значение в цикле, а при первом проходе по блокам, над которыми доминирует B , значение a присвоено в s . Конечно же, при втором и последующем проходах a содержит присвоенное ему значение.

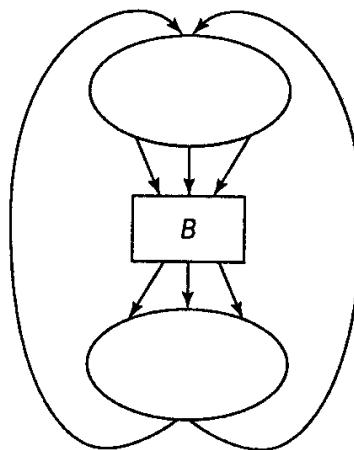


Рис. 10.71. Блок, разделяющий цикл на две части

Если вызов отладчика инициирован ошибкой времени исполнения, то шансы, что ошибка обнаружилась при первом проходе по циклу, достаточно велики. Если это так, то мы должны знать, находимся ли в цикле выше или ниже блока B на рис. 10.71; исходя из этого мы выясняем, определено ли к этому моменту значение a в инструкции s (и в этом случае мы просто выводим значение a , которое дает оптимизированный код), или значение a должно быть тем, что и в исходной программе при входе в цикл. В этом случае мы мало что в состоянии сделать, за исключением ситуаций, когда выполняются следующие условия.

1. Отладчик имеет доступ к информации о достигающих определениях как для исходной, так и оптимизированной программы.
2. Имеется единственное определение a , достигающее заголовка в исходной программе.

3. Это определение является также единственным для некоторой переменной x , которое достигает точки, где был вызван отладчик.

Если выполняются все три условия, то мы можем вывести значение x , сообщив, что это и есть значение a .

Читатель должен отдавать себе отчет, что все это, по сути, невыполнимо при вызове отладчика установленной пользователем точкой останова, поскольку тогда мы не в состоянии определить, первый ли это проход по циклу или нет. Однако при работе с точками останова можно вставить в оптимизированную программу код, помогающий определить, первый ли это проход по циклу или нет — но это означает подход, раскритикованый нами ранее, поскольку при этом мы работаем со специально подготовленным для отладки кодом.

Упражнения

- 10.1. Рассмотрим программу умножения матриц, приведенную на рис. 10.72.

```
begin
    for i := 1 to n do
        for j := 1 to n do
            c[i,j] := 0;
    for i := 1 to n do
        for j := 1 to n do
            for k := 1 to n do
                c[i,j] := c[i,j] + a[i,k] * b[k,j]
end
```

Рис. 10.72. Программа умножения матриц

- Полагая, что для a , b и c выделена статическая память и что на одно слово используется четыре байта адресуемой побайтно памяти, приведите трехадресные инструкции для программы на рис. 10.72.
- Сгенерируйте на их основе целевой код.
- Постройте граф потока для трехадресных инструкций.
- Устраните общие подвыражения из каждого базового блока.
- Найдите циклы в графе потока.
- Переместите инвариантные относительно цикла вычисления из циклов.
- Найдите переменные индукции каждого цикла и по возможности устраните их.
- Сгенерируйте целевой код на основе графа потока из (g) и сравните его с кодом, полученным при выполнении (b).

- 10.2. Вычислите достигающие определения и ои-цепочки для исходного графа потока из упр. 10.1c и окончательного графа потока из 10.1g.

- 10.3. Программа на рис. 10.73 подсчитывает простые числа от 2 до n методом решета Эратосфена на достаточно большом массиве.

```
begin
    read n;
    for i := 2 to n do
        a[i] := true; /* Инициализация */
    count := 0;
    for i := 2 to n ** .5 do
```

```

if a[i] then /* i - простое число */
begin
    count := count + 1;
    for j := 2 * i to n by i do
        a[j] := false /* j делится на i */
end
print count
end

```

Рис. 10.73. Программа для вычисления простых чисел

- Транслируйте программу на рис. 10.73 в трехадресный код в предположении статического выделения памяти для массива a.
 - Сгенерируйте на основе полученного трехадресного кода целевой машинный код.
 - Постройте граф потока на основе полученного трехадресного кода.
 - Покажите дерево доминаторов для графа потока из (a).
 - Для графа потока из (c) укажите обратные дуги и их естественные циклы.
 - Вынесите инвариантные вычисления из циклов с использованием алгоритма 10.7.
 - Устраните по возможности переменные индукции.
 - Распространите по возможности инструкции копирования.
 - Возможно ли сжатие цикла? Если да — сделайте это.
 - В предположении, что n всегда четно, разверните по одному разу каждый внутренний цикл. Какие новые варианты оптимизации возможны теперь?
- 10.4.** Повторите упр. 10.3 в предположении, что память для массива a выделяется динамически; при этом первое слово a указывает указатель ptr.
- 10.5.** Для графа потока на рис. 10.74 вычислите следующее:
- ои- и ио-цепочки,
 - активные переменные в конце каждого блока,
 - доступные выражения.

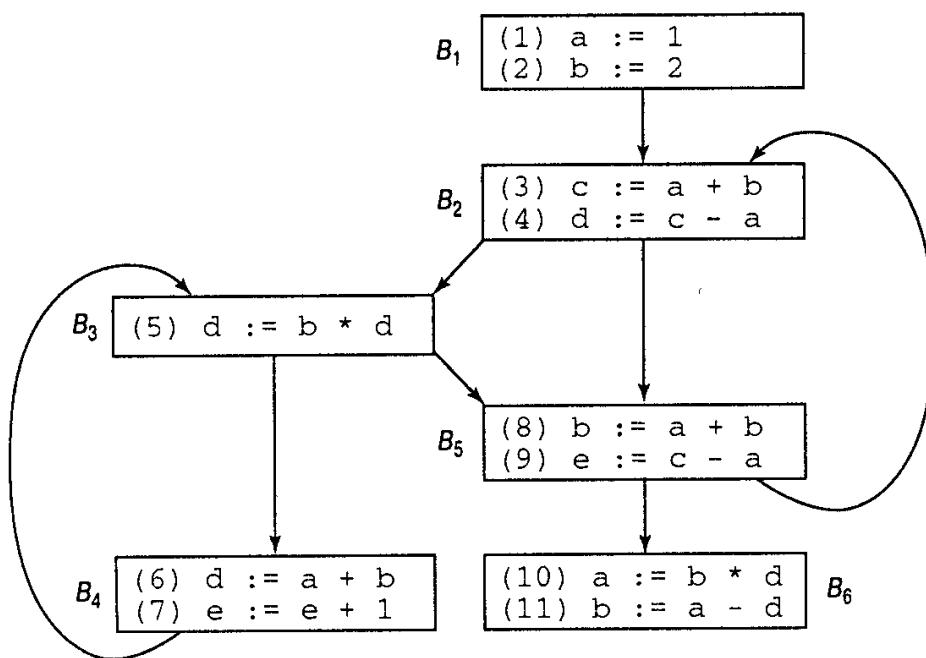


Рис. 10.74. Граф потока

- 10.6. Возможно ли дублирование констант на рис. 10.74? Если да, выполните его.
- 10.7. Имеются ли на рис. 10.74 общие подвыражения? Если да — устраните их.
- 10.8. Выражение e называется *очень занятым* (*very busy*) в точке p , если независимо от пути из p выражение e будет вычисляться до определения любого из его операндов. Приведите алгоритм потока данных в духе раздела 10.6 для поиска всех очень занятых выражений. Какой оператор слияния вы при этом используете, и будет ли распространение прямым или обратным? Примените ваш алгоритм к графу потока на рис. 10.74.
- *10.9. Если выражение e очень занято в точке p , мы можем *поднять* (*hoist*) e путем его вычисления в точке p и сохранения его значения для последующего использования. (Примечание. Такая оптимизация обычно не экономит время работы, но может сэкономить память.) Приведите алгоритм для поднятия очень занятых выражений.
- 10.10. Имеются ли выражения, которые могут быть подняты, на рис. 10.74? Если да — поднимите их.
- 10.11. Там, где это возможно, распространите шаги копирования, введенные в изменениях, предложенных в упр. 10.6, 10.7 и 10.10.
- 10.12. *Расширенный базовый блок* представляет собой последовательность блоков B_1, \dots, B_k , таких, что B_i для $1 \leq i < k$ является единственным предшественником B_{i+1} , а B_1 не имеет единственного предшественника. Найдите расширенные базовые блоки, заканчивающиеся в каждом узле
 - на рис. 10.39,
 - графа потока, построенного в упр. 10.1c,
 - на рис. 10.74.
- *10.13. Приведите алгоритм, который для n -узельного графа потока за время $O(n)$ находит расширенный базовый блок, заканчивающийся в каждом узле.
- 10.14. Мы можем выполнить некоторую межблочную оптимизацию кода без анализа потока данных, рассматривая каждый расширенный базовый блок, как если бы он был базовым. Приведите алгоритмы для каждой из указанных далее оптимизаций в расширенном базовом блоке. В каждом случае укажите, какое влияние могут оказывать изменения в одном расширенном базовом блоке на другие расширенные базовые блоки?
 - Устранение общих подвыражений.
 - Дублирование констант.
 - Распространение копий.
- 10.15. Для графа потока из упр. 10.1c выполните следующее.
 - Найдите последовательность преобразований T_1 и T_2 для графа.
 - Найдите последовательность графов интервалов.
 - Что из себя представляет предельный граф потока? Является ли график приводимым?
- 10.16. Повторите упр. 10.15 для графа потока на рис. 10.74.

****10.17.** Покажите, что следующие условия эквивалентны (и являются альтернативными определениями “приводимого графа потока”).

- Предельное приведение T_1-T_2 представляет собой единственный узел.
- Предел анализа интервалов представляет собой единственный узел.
- Дуги графа потока могут быть разделены на два класса; один из них образует ациклический граф, а второй состоит из обратных дуг, головы которых доминируют над их хвостами.
- Граф потока не имеет подграфов приведенного на рис. 10.75 вида. Здесь n_0 представляет собой начальный узел, и все узлы n_0, a, b, c различны, с возможным исключением $n_0 = a$. Стрелки представляют пути, разделенные узлами (естественно, за исключением конечных точек).

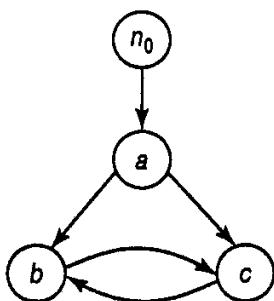


Рис. 10.75. Подграф, запрещенный для приводимого графа потока

10.18. Разработайте алгоритм для вычисления (a) доступных выражений и (b) активных переменных для языка с указателями, обсуждавшегося в разделе 10.8. Убедитесь в консервативности предположений относительно *gen*, *kill*, *use* и *def* в случае (b).

10.19. Разработайте алгоритм для межпроцедурного вычисления достигающих определений с использованием модели из раздела 10.8. Убедитесь в консервативности приближения к истинному решению.

10.20. Предположим, что параметры передаются не по ссылке, а по значению. Могут ли в этом случае два имени оказаться псевдонимами друг друга? А в случае использования технологии копирования-восстановления?

10.21. Чему равна глубина графа потока в упр. 10.1c?

****10.22.** Докажите, что глубина приводимого графа потока никогда не меньше количества выполнений анализа интервалов, которое должно быть выполнено для получения единственного узла.

***10.23.** Обобщите алгоритм анализа потока данных из раздела 10.8, основанный на структуре, для обобщенной схемы потока данных в смысле раздела 10.11. Какие предположения о F и \wedge требуется сделать для того, чтобы гарантировать работоспособность алгоритма?

***10.24.** Интересная и мощная схема анализа потока данных может быть получена, если представить, что распространяемые “значения” являются всеми возможными разбиениями выражений, так что два выражения принадлежат одному классу только в том случае, если они имеют одно и то же значение. Чтобы избежать перечисления бесконечного множества всех возможных выражений, мы можем

представить такие значения, перечислив только минимальные выражения, эквивалентные некоторым другим. Например, при выполнении инструкций

$A := B$
 $C := A + D$

у нас есть следующие минимальные тождества: $A \equiv B$ и $C \equiv A+D$. Отсюда вытекают другие тождества, такие как $C \equiv B+D$ или $A+E \equiv B+E$, но их не нужно перечислять явно.

- a) Какой оператор слияния должен использоваться в этой схеме?
- b) Приведите структуру данных для представления значений и алгоритм реализации оператора слияния.
- c) Что представляют собой функции, связываемые с инструкциями? Поясните то действие, которое оказывает на разбиение функция, связанная с присвоением типа $A := B+C$.
- d) Является ли эта схема дистрибутивной? Монотонной?

10.25. Как вы воспользуетесь данными, собранными схемой из упр. 10.24 для

- a) устранения общих подвыражений?
- b) распространения копий?
- c) дублирования констант?

* 10.26. Приведите доказательства следующих утверждений о соотношении \leq над решетками.

- a) Из $a \leq b$ и $a \leq c$ следует $a \leq (b \wedge c)$.
- b) Из $a \leq (b \wedge c)$ следует $a \leq b$.
- c) Из $a \leq b$ и $b \leq c$ следует $a \leq c$.
- d) Из $a \leq b$ и $b \leq a$ следует $a = b$.

* 10.27. Покажите, что следующее условие является необходимым и достаточным для сходимости итеративного алгоритма потока данных с упорядочением вглубь за количество проходов, на 2 большее глубины: для всех функций f и g и значения a $f(g(a)) \geq f(a) \wedge g(a) \wedge a$.

10.28. Покажите, что схемы достигающих определений и доступных выражений удовлетворяют условию из упр. 10.27. (*Примечание.* На самом деле эти схемы сходятся за количество проходов, на единицу превышающее глубину.)

* 10.29. Вытекает ли условие упр. 10.27 из монотонности? Из дистрибутивности? А наоборот?

10.30. На рис. 10.76 мы видим два базовых блока — “первоначальный” код и его оптимизированную версию.

- a) Постройте даги для блоков на рис. 10.76a и 10.76б. Проверьте, что при предположении, что только переменная J активна на выходе, эти два блока эквивалентны.
- b) Аннотируйте узлы дага временем, в течение которого в этих узлах известны значения каждой переменной.
- c) Укажите, каким инструкциям на рис. 10.76a соответствуют ошибки, возникающие в каждой из инструкций от (1') до (4').

d) Для каждой ошибки из части (c) укажите, для каких переменных на рис. 10.76а можно вычислить их значения и каким образом это можно сделать.	
e) Предположим, что мы можем использовать алгебраически корректные законы типа “если $a+b = c$, то $a = c-b$ ”. Изменится ли при этом ответ (d)?	
(1) $E := A + B$	(1') $E := A + B$
(2) $F := E - C$	(2') $E := E - C$
(3) $G := F * D$	(3') $F := E * D$
(4) $H := A + B$	(4') $J := E + F$
(5) $I := I - C$	
(6) $J := I + G$	
(a) Первоначальный код	(б) Оптимизированный код

Рис. 10.76. Первоначальный и оптимизированный код

- 10.31. Обобщите пример 10.14 для учета произвольного множества инструкций `break`. Обобщите его также для использования инструкций `continue`, которые, не прекращая работу внутреннего цикла, переходят непосредственно к очередной итерации. (Указание. Воспользуйтесь методами для приводимых графов потока из раздела 10.10.)
- 10.32. Покажите, что в алгоритме 10.3 множества определений *in* и *out* никогда не уменьшаются. Аналогично покажите, что в алгоритме 10.4 те же множества выражений никогда не увеличиваются.
- 10.33. Обобщите алгоритм 10.9 для устранения переменных индукции на случай отрицательного множителя.
- 10.34. Обобщите алгоритм для определения возможных целей указателя из раздела 10.8 на случай, когда указатели могут указывать на другие указатели.
- *10.35. При оценке каждого из следующих множеств определите, какая из оценок — завышенная или заниженная — является консервативной. Поясните ваш ответ на основании предполагаемого использования полученной информации.
- a) Доступные выражения.
 - b) Переменные, изменяемые процедурой.
 - c) Переменные, не изменяемые процедурой.
 - d) Переменные индукции, принадлежащие данному семейству.
 - e) Инструкции копирования, достигающие данной точки.
- *10.36. Уточните алгоритм 10.12 для вычисления псевдонимов данной переменной в данной точке.
- *10.37. Модифицируйте алгоритм 10.12 для передачи параметров
- a) по значению;
 - b) с использованием копирования и восстановления.
- *10.38. Докажите, что алгоритм 10.13 сходится к надмножеству (не обязательно собственному) истинного множества измененных переменных.

- *10.39. Обобщите алгоритм 10.13 для определения измененных переменных в случае, когда разрешены процедурные переменные.
- *10.40. Докажите, что в любом графе интервалов каждый узел представляет область исходного графа потока.
- 10.41. Докажите, что алгоритм 10.16 правильно вычисляет множество доминаторов каждого узла.
- *10.42. Модифицируйте алгоритм 10.17 (достигающие определения, основанные на структуре) для вычисления достигающих определений только для определенных небольших областей, без требования размещения в памяти всего графа целиком. Убедитесь, что ваш результат консервативен. Адаптируйте ваш алгоритм для случая доступных выражений. Какой из этих алгоритмов предоставляет более полезную информацию?
- *10.43. В разделе 10.10 мы предложили ускорение алгоритма 10.17, основанное на комбинировании преобразований T_1 и T_2 . Докажите корректность предложенной модификации.
- 10.44. Обобщите итеративный метод из раздела 10.11 для задач обратного потока.
- **10.45. Докажите, что при сходимости алгоритма 10.18 конечное решение $\leq \text{top}$ -решения, показав, что для каждого пути P длиной i после выполнения i итераций $in[B_i] \leq f_p[\text{top}]$.
- 10.46. На рис. 10.77 приведен график потока программы на гипотетическом языке, рассмотренном в разделе 10.12. Найдите наилучшую оценку типов каждой переменной с использованием алгоритма 10.19.

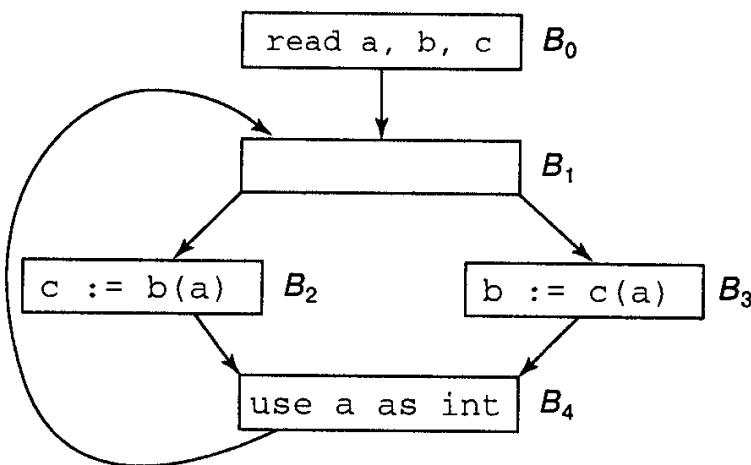


Рис. 10.77. Пример программы для вычисления типов

Библиографические примечания

Дополнительная информация об оптимизации кода содержится в работах [1, 90, 177, 326, 391]. В [26] можно найти библиографию по данному вопросу.

В литературе описаны многие оптимизирующие компиляторы. Так, в [124] рассматривается ранний компилятор, использующий сложные технологии оптимизации. В [294, 390] можно найти описание деталей создания оптимизирующего компилятора для For-

tran, а с дополнительными методами оптимизации Fortran можно познакомиться в [70, 314]. В [460] обсуждается создание оптимизирующего компилятора для Bliss.

В [27] описана система, созданная для проведения экспериментов по оптимизации программ. Об эффективности различных методов оптимизации для PL/I-подобного языка рассказано в [88]. В [35] вы найдете описание реализации оптимизирующих преобразований, использованных в компиляторах PL/I и С. О компиляторе для варианта PL/I, использующем простой алгоритм для получения низкоуровневого промежуточного кода, который затем усовершенствуется глобальными оптимизирующими преобразованиями, сообщается в [37]. О построении оптимизатора для SETL рассказывается в [149], об экспериментах с переносимым машинно-независимым оптимизатором для Pascal — в работе [80], а о переносимом машинно-независимом оптимизирующем компиляторе для Modula-2 — в [351].

Систематическое изучение анализа потока данных начато Алленом [24] и Коком [85] и продолжено в их совместной работе [29], хотя различные методы анализа потока данных применялись и до того.

Синтаксически управляемый анализ потока данных, описанный в разделе 10.5, использован в языках программирования Bliss [160, 460], SIMPL [463] и Modula-2 [351]. Дополнительное обсуждение этого семейства алгоритмов вы найдете в работах [177, 178, 379].

Итеративный подход к решению задач анализа потока данных, обсуждавшийся в разделе 10.6, прослежен в работе [438], авторы которой использовали этот метод в компиляторе Fortran. Использование упорядочения вглубь для повышения эффективности было предложено в [181].

Подход к задачам анализа потока данных с использованием интервалов был впервые предложен в [85], а в [238] этот метод был распространен на задачи обратного потока типа активных переменных. Причина доверия к этому методу, по мнению автора [240], кроется в том, что основанный на интервалах метод более эффективен, чем итеративный, в случае если оптимизируемый язык приводит к появлению малого количества (если они имеются вообще) неприводимых графов потока. Вариант с преобразованиями T_1 и T_2 , использованный в данной книге, предложен в работе [435]; несколько более быстрая версия, использующая тот факт, что большинство областей имеют единственный выход, приведена в [171].

Изначальное определение приводимого графа потока, который превращается в отдельный узел при итеративном анализе интервалов, дано в [24]. Эквивалентные характеристики можно найти в [179, 180, 236, 425], а метод разделения узлов для неприводимых графов потока — в [89].

Идея о моделировании структурированного потока управления приводимыми графиками потока содержится в [77, 233, 266], а в [42] описано их использование в алгоритме структуризации программы.

Решеточный подход к итеративному анализу потока данных введен в [249]; в [429, 447] даны аналогичные формулировки. Эффективная версия алгоритма Килдалла, в которой используется упорядочение вглубь, предложена в [229].

Хотя Килдаллом и использовано предположение о дистрибутивности (которому в действительности не удовлетворяют схемы типа вычисления констант из примера 10.42), достаточность монотонности была осознана во многих работах по алгоритмам потоков данных, таких, например, как [171, 226, 230, 380, 395, 396, 427, 429].

Другим направлением, вытекающим из статьи Килдалла, оказалось усовершенствование алгоритмов для работы с частными задачами потока данных (таких как пример 10.42). Одна из ключевых идей заключается в том, что элементы решетки рассматриваются не атомарно и можно воспользоваться тем, что в действительности они представляют собой отображения переменных в значения ([365, 449]; в [269] эта идея используется для более обычных задач).

В [241] содержится обзор технологий анализа потока данных, а в [99] — обзор идей, связанных с решетками.

Основные идеи оптимизации циклов путем перемещения кода и ограниченных видов устранения переменных индукции появились в [159]; фундаментальной статьей по оптимизации циклов является [23]. В [28, 442] можно найти обзоры соответствующих технологий, а в [321] описан алгоритм, одновременно устраняющий из циклов избыточные и инвариантные вычисления.

Обсуждение устранения переменных индукции в разделе 10.7 основано на работе [294]; более мощные алгоритмы можно найти в работе [30].

Алгоритмы для некоторых задач оптимизации циклов подробно в книге не рассмотрены; например, поиск наличия пути от a до b , не проходящего через c , может быть осуществлен с помощью эффективного алгоритма из [448].

Использование доминаторов, как для обнаружения циклов, так и выполнения перемещения кода, было впервые предложено в работе [294], хотя общая идея приписывается Проссеру [355]. Алгоритм 10.16 для поиска доминаторов был предложен независимо в работах [19] и [357]. Использование упорядочения вглубь для ускорения алгоритма предложено в [181], в то время как асимптотически более эффективный способ выполнения этой работы описан в [424], а в [281] можно найти алгоритм, более пригодный для практического использования.

Изучение псевдонимов и межпроцедурный анализ берут свое начало в работах [408] и [25]. Был разработан ряд более мощных методов, чем описанные в разделе 10.8, которые в целом имеют дело с отношением псевдонимности в каждой точке программы для того, чтобы избежать некоторых пар, которые “находит” простой алгоритм и которые на самом деле невозможны. Эти алгоритмы можно найти в работах [44, 46, 451] (см. также построение графов вызовов в [385]).

Влияние исключений на анализ потока данных, аналогичное межпроцедурному анализу, рассматривается в работе [183].

Наше рассмотрение определения типов в разделе 10.12 основано на фундаментальной статье [429]. В [231] имеется более мощный алгоритм определения типов.

Обсуждение символьной отладки в разделе 10.13 взято из [184].

Имеется ряд работ, в которых производятся попытки вычислить эффект от применения той или иной оптимизации, однако числовые значения сильно зависят от рассматриваемого языка программирования. Классической в этой области стала работа [256] Кнута; представляют также интерес статьи [80, 81, 86, 88, 150, 341, 351].

Еще один вопрос оптимизации, не рассмотренный в нашей книге, — оптимизация языков “очень высокого уровня”, таких как язык работы с множествами SETL, в котором мы в действительности изменяем лежащие в основе вычислений алгоритмы и структуры данных²². В этой области основная оптимизация представляет собой обобщенное устранение переменных индукции ([120, 137, 138, 340]).

²² Другим языком такого типа можно считать язык структурированных запросов SQL. — Прим. ред.

Еще одним ключевым шагом в оптимизации языков очень высокого уровня является выбор структур данных; этому вопросу посвящены работы [293, 392, 395, 396].

Мы также не касались вопросов инкрементальной оптимизации кода, при которой малые изменения в программе не требуют полного проведения оптимизации заново. В [386] обсуждается инкрементальный анализ потока данных, а в [350] делается попытка осуществить инкрементальную оптимизацию базовых блоков.

И наконец, мы должны упомянуть другие пути использования анализа потока данных, в частности, для восстановления после ошибок или проверки диапазона массивов в процессе компиляции (см. [39, 176, 417]).

Одно из наиболее значительных применений анализа потока данных вне оптимизации кода — статическая проверка программ на наличие ошибок, осуществляемая в процессе компиляции, где мы порекомендовали бы обратиться к работам [3, 139, 148, 336].

ГЛАВА 11

Создание компилятора

Теперь, после того как мы рассмотрели принципы, технологии и инструменты, используемые при разработке компиляторов, предположим, что мы хотим создать компилятор. Хорошо спланированная работа может быть выполнена более быстро и гладко, а потому в этой небольшой главе мы рассмотрим некоторые вопросы, возникающие при практическом создании компиляторов. В основном, наше рассмотрение будет связано с использованием операционной системы UNIX и ее инструментария.

11.1. Планирование компилятора

Новый компилятор может быть предназначен для компиляции нового исходного языка или для получения нового целевого кода (или и для того, и для другого одновременно). Используя предложенную в этой книге схему, мы в конечном счете получим конструкцию компилятора, состоящего из множества модулей, на разработку и реализацию которых влияют различные факторы.

Вопросы исходного языка

На размер и количество модулей компилятора влияет “размер” исходного языка. Хотя точного определения, что такое размер языка, не существует, представляется очевидным, что компилятор для языка типа Ada или PL/I больше и сложнее в реализации, чем компилятор для небольшого языка типа Ratfor (“Rational” Fortran [244]), или EQN, — язык типографской печати математических формул.

Еще одним важным фактором является то, насколько будет расширен исходный язык в процессе построения компилятора. Хотя спецификации исходного языка и выглядят нерушимой данностью, только немногие из языков остаются неизменны в течение всей жизни компилятора. Эволюционируют даже естественные языки, хотя и достаточно медленно. Например, Fortran достаточно заметно изменился со временем своего появления на свет в 1957 году; циклы, строки Холлерита и условные инструкции в языке Fortran 77 совсем не такие, как в первоначальном варианте. В [383] можно познакомиться с эволюцией языка С.

Однако новый, экспериментальный язык может измениться в процессе реализации очень сильно. Один из путей создания нового языка представляет собой эволюцию компилятора для работающего прототипа языка в компилятор, удовлетворяющий запросам некоторых групп пользователей. Многие из “небольших”, изначально разрабатывавшихся в UNIX языков типа AWK и EQN были созданы именно таким образом.

Следовательно, разработчик компилятора может ожидать, что в процессе времени жизни компилятора произойдут определенные изменения в исходном языке. Модульная разработка и использование различных инструментов могут помочь разработчику справиться с этими изменениями. Например, использование генераторов для реализации лек-

сического и синтаксического анализаторов компилятора позволит разработчику существенно легче отреагировать на изменения в языке, чем в случае непосредственной разработки кода анализаторов.

Вопросы целевого языка

При разработке компилятора следует внимательно учитывать природу и ограничения целевого языка и среды исполнения в связи с их сильным влиянием на конструкцию компилятора и используемую стратегию генерации кода. В случае нового целевого языка разработчик компилятора должен быть уверен в корректности языка и правильном его понимании. Новая машина или новый ассемблер могут содержать ошибки, которые компилятор, вероятно, вскроет. Ошибки в целевом языке могут существенно затруднить задачу отладки компилятора.

Хорошо разработанный исходный язык зачастую реализуется на различных целевых машинах, а в случае продолжительного времени жизни языка встанет вопрос о генерации кода для нескольких поколений целевых машин. Можно быть уверенным в дальнейшей эволюции аппаратного обеспечения, так что легко перестраиваемые компиляторы, вероятно, имеют преимущество. Следовательно, становится крайне важным вопрос тщательной разработки промежуточного языка, ограничивающего зависимость от целевой машины только небольшим числом модулей.

Критерии производительности

Имеется ряд аспектов производительности компилятора: скорость его работы, качество кода, диагностика ошибок, переносимость и поддержка. Достаточно сложно найти компромисс среди всех этих критериев, а кроме того, ряд параметров может остаться не определенным спецификацией компилятора. Например, что важнее — скорость работы компилятора или скорость работы генерируемого им кода? Насколько важны хорошие сообщения об ошибках и восстановление работы компилятора после ошибок?

Скорость работы компилятора достигается за счет максимально возможного сокращения количества модулей и проходов, возможно, вплоть до непосредственной генерации машинного кода за один проход. Однако такой подход может не привести к компилятору, который генерирует высококачественный целевой код и имеет простую поддержку.

Есть два аспекта переносимости компилятора: смена платформы, на которой работает компилятор (*rehostability*), и смена платформы для создаваемых им программ (*retargetability*). Переносимый компилятор может оказаться не настолько эффективен, как компилятор, разрабатываемый для определенной машины, поскольку последний может использовать информацию о целевой машине, которую не может использовать переносимый компилятор.

11.2. Подходы к разработке компилятора

Существует ряд общих подходов, которые разработчик может использовать при реализации компилятора. Простейший из них состоит в изменении уже существующего компилятора для работы на новой платформе или для компиляции программ для другой платформы. Если подходящего для этой цели компилятора нет, разработчик может попробовать приспособить имеющийся компилятор для схожего языка и реализовать необходимые компоненты, используя соответствующий инструментарий либо действуя вручную. Организация совершенно нового компилятора требуется относительно редко.

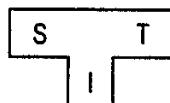
Независимо от того, какой именно подход будет принят, написание компилятора представляет собой задачу создания программного обеспечения, в которой могут быть применены все принципы технологии разработки программного обеспечения. Опыт, вынесенный из разработки программ другого типа (см., например, [65]), вполне применим для увеличения надежности и возможностей поддержки и сопровождения конечного продукта. Конструкция компилятора, обеспечивающая легкое внесение изменений, позволит ему эволюционировать вместе с языком. В этом вам может помочь многочисленный инструментарий для построения компиляторов.

Раскрутка

Компилятор — достаточно сложная программа, которую желательно писать на языке более дружественном, чем ассемблер. В среде программирования UNIX компиляторы обычно пишут на С — даже компиляторы языка С. Использование возможностей языка для компиляции его самого является сущностью *раскрутки* (bootstrapping). Здесь мы рассмотрим использование раскрутки для создания компиляторов и перенесения их с одной машины на другую путем изменения заключительной стадии компиляции. Основные идеи раскрутки известны со средины 50-х годов (см. [414]).

Раскрутка может привести к вопросу о том, как был скомпилирован первый компилятор (этот вопрос напоминает известное “Что было раньше — курица или яйцо?”, однако ответить на него легче). Для ответа мы рассмотрим, каким образом Lisp стал языком программирования. Мак-Карти [305] упоминает, что первоначально Lisp использовался как система записи функций, которые затем вручную транслировались в ассемблерный код и исполнялись. Реализация интерпретатора Lisp возникла неожиданно. Мак-Карти хотел показать, что Lisp был способом записи для описания функций “более ясным, чем машина Тьюринга или рекурсивные определения из теории рекурсивных функций”, и написал на языке Lisp функцию *eval*[*e, a*], принимающую в качестве аргумента Lisp-выражение *e*. Рассел обратил внимание, что *eval* может служить в качестве интерпретатора Lisp, закодировал его вручную и тем самым создал язык программирования с интерпретатором. Как упоминалось в разделе 1.1, вместо генерации целевого кода интерпретатор выполняет операции исходной программы.

При использовании технологии раскрутки компилятор характеризуется тремя языками: исходным языком S, который он компилирует, целевым языком T, для которого он генерирует код, и языком реализации I, на котором он написан. Эти три языка мы представляем в виде показанной далее T-диаграммы [63].



В тексте мы будем обозначать такую диаграмму как SIT. Три языка — S, I и T — могут существенно отличаться. Например, компилятор может работать на одной машине, давая целевой код для другой машины (такой компилятор называется *кросс-компилятором* (cross-compiler)).

Предположим, что мы создали кросс-компилятор для нового языка L на языке реализации S с генерацией кода для машины N, т.е. создали LSN. Если существующий компилятор для языка S работает на машине M и генерирует код для M, он характеризуется как SMM. Если LSN пропущен через SMM, мы получаем компилятор LMN, т.е. компилятор языка L для целевой машины N, работающий на машине M. Этот процесс проиллюстрирован на рис. 11.1, где Т-диаграммы этих компиляторов собраны вместе.

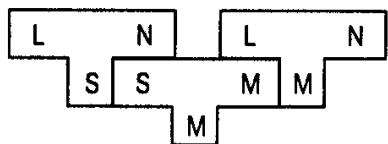


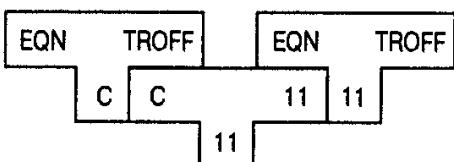
Рис. 11.1. Компиляция компилятора

После того как мы собрали Т-диаграммы, заметим, что язык реализации S компилятора $L_S N$ должен быть таким же, как и исходный язык существующего компилятора $S_M M$, и что целевой язык M существующего компилятора должен быть таким же, как и язык реализации компилятора $L_M N$. Тройка Т-диаграмм (как, например, на рис. 11.1) может быть представлена в виде уравнения

$$L_S N + S_M M = L_M N.$$

Пример 11.1

Первая версия компилятора EQN (см. раздел 12.1) в качестве языка реализации имела С и генерировала команды для языка форматирования текста TROFF. Как показано на приведенной далее диаграмме, кросс-компилятор для EQN, работающий на PDP-11, был получен путем компиляции $EQN_C TROFF$ на PDP-11 с помощью транслятора C_{1111} .

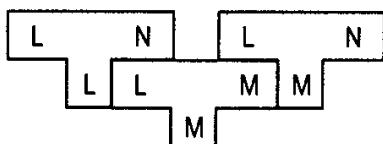


□

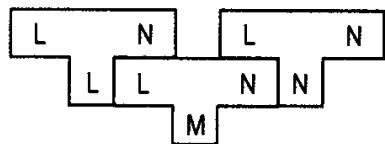
Один вид раскрутки строит компилятор для все больших и больших подмножеств языка. Предположим, что новый язык L должен быть реализован на машине M . В качестве первого шага мы можем написать небольшой компилятор, который транслирует подмножество S языка L в целевой код для M , т.е. компилятор $S_M M$. Затем это подмножество S используется для написания компилятора $L_S M$ для L . При пропускании $L_S M$ через $S_M M$ мы получим реализацию L , а именно $L_M M$. Одним из первых языков, реализованных на самом себе, был Neliac [202].

Вирт [456] заметил, что Pascal был впервые реализован путем написания компилятора на Pascal. Затем компилятор был “вручную” транслирован в доступный низкоуровневый язык без малейших попыток оптимизации. Этот компилятор был написан для подмножества “(>60%)” Pascal, после чего потребовалось несколько стадий раскрутки для получения полной версии языка. Методы, используемые при этом, подытожены в работе [278].

Для полной реализации преимуществ технологии раскрутки компилятор должен быть написан на языке, который он компилирует. Предположим, что мы создаем компилятор $L_L N$ для языка L на языке L , генерирующий код для машины N . Разработка происходит на машине M , где имеется существующий компилятор $L_M M$, генерирующий для языка L код для машины M . Первая компиляция $L_L N$ с помощью $L_M M$ дает нам кросс-компилятор $L_M N$, работающий на машине M и производящий код для N .



Компилятор $L_L N$ может быть скомпилирован второй раз, теперь уже с помощью созданного кросс-компилятора.



Результат второй компиляции — компилятор L_NN , который работает на машине N и генерирует код для нее. У такого процесса, состоящего из двух стадий, множество полезных применений, так что мы полностью приведем его на рис. 11.2.

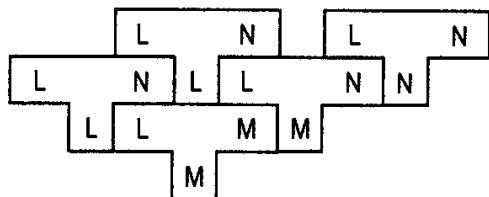


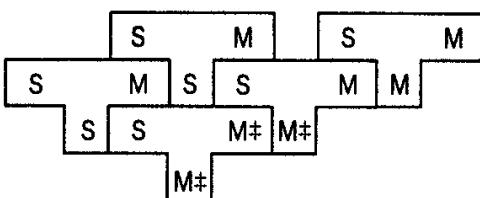
Рис. 11.2. Раскрутка компилятора

Пример 11.2

Данный пример навеян разработкой компилятора Fortran H (см. раздел 12.4). “Этот компилятор был написан на Fortran и прошел три стадии раскрутки. Первым шагом было преобразование компилятора, работающего на IBM 7094, в компилятор для System/360 — весьма непростая задача. Второй шаг состоял в его оптимизации, что сократило размер компилятора с 550Кб до 400Кб” ([294]).

При использовании технологии раскрутки оптимизирующий компилятор может оптимизировать сам себя. Предположим, что вся работа выполнялась на машине M . У нас есть $S_S M$, хороший оптимизирующий компилятор для языка S , написанный на языке S , и мы хотим получить $S_M M$ — хороший оптимизирующий компилятор языка S для машины M .

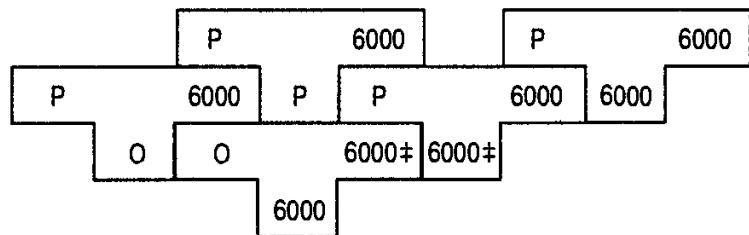
Мы можем создать $S_M \ddagger M \ddagger$, сделанный на скорую руку быстрый компилятор для S на M , который не только генерирует не очень хороший код для M , но и делает это слишком медленно. (Символ \ddagger указывает на “недоведенность”, т.е. $S_M \ddagger M \ddagger$ представляет собой не очень хорошую реализацию компилятора, генерирующего не очень хороший код.) Однако мы можем применить посредственный компилятор $S_M \ddagger M \ddagger$ для получения хорошего компилятора S за два шага.



Сначала оптимизирующий компилятор $S_S M$ транслируется компилятором $S_M \ddagger M \ddagger$ для получения $S_M \ddagger M$ — не очень хорошей реализации оптимизирующего компилятора, дающего хороший код. Хороший оптимизирующий компилятор $S_M M$ получается перекомпиляцией $S_S M$ с помощью $S_M \ddagger M$. □

Пример 11.3

В работе [32] описана реализация Pascal аналогичным путем. Изменения, вносимые в Pascal, привели к компилятору для машин CDC 6000. В приведенной диаграмме О означает “старый” Pascal, а Р — обновленную версию.



Компилятор обновленного Pascal был написан на старом Pascal и транслирован в P6000 \ddagger 6000 (как и в примере 11.2, символ \ddagger указывает источник неэффективности). Старый компилятор генерировал недостаточно эффективный код. “Таким образом, скорость компилятора [P6000 \ddagger 6000] была весьма посредственной, а его требования к памяти непомерно велики” [32]. Изменения в Pascal были достаточно невелики, так что оказалось возможным небольшими усилиями транслировать компилятор РО6000 вручную в РР6000 и пропустить его через неэффективный компилятор P6000 \ddagger 6000 для получения окончательного варианта. □

11.3. Среда разработки компилятора

По существу, компилятор является просто программой. Среда, в которой эта программа разрабатывается, может повлиять на скорость и надежность реализации компилятора. Не менее важен и язык разработки компилятора. Хотя имеются компиляторы, написанные на языках типа Fortran, все же наиболее логичным выбором для большинства разработчиков компиляторов будут системно-ориентированные языки наподобие C.

Если исходный язык представляет собой новый системно-ориентированный язык, то имеет смысл разрабатывать компилятор на нем самом. С использованием рассмотренной в предыдущем разделе технологии, компилирование компилятора облегчает его отладку.

Существенно облегчить создание эффективного компилятора может инструментарий разработки программного обеспечения, имеющийся в используемой среде программирования. При написании компилятора обычным является разделение всей программы на модули, где каждый модуль может обрабатываться своим способом. Программа, управляющая обработкой таких модулей, — незаменимый помощник создателя компилятора. UNIX содержит утилиту, именуемую *make* [130], которая облегчает управление и обслуживание модулей, составляющих компьютерную программу; *make* отслеживает связи между модулями программы и выполняет только те команды, которые необходимы после внесения изменений в модули.

Пример 11.4

Утилита *make* считывает из файла *makefile* спецификацию задач, которые должны быть выполнены. В разделе 2.9 мы создавали транслятор, компилируя семь файлов с помощью компилятора C; каждый файл зависел от глобального заголовочного файла *global.h*. Чтобы показать, как задача сборки компилятора может быть выполнена с помощью *make*, предположим, что получаемый в результате компилятор имеет имя *trans*. Тогда спецификации в *makefile* могут выглядеть следующим образом.

```
OBJS = lexer.o parser.o emitter.o symbol.o \
        init.o error.o main.o
```

```
trans:  $(OBJS)
        cc $(OBJS) -o trans
```

```
lexer.o parser.o emitter.o symbol.o \
init.o error.o main.o : global.h
```

Знак равенства в первой строке делает OBJS из левой части обозначением семи объектных файлов из правой (длинные строки разбиваются на меньшие обратной косой чертой в конце продолжающейся части, так что первые две строки должны рассматриваться как одна большая). Двоеточие во второй строке говорит о том, что файл `trans` слева от него зависит от всех файлов, перечисленных в OBJS. За такой строкой зависимости могут следовать команды для “создания” файла, указанного слева от двоеточия, так что третья строка гласит, что целевая программа `trans` создается путем связывания объектных файлов `lexer.o`, `parser.o`, ..., `main.o`. Однако `make` знает, что сначала должны быть созданы объектные файлы, и программа создает их автоматически из соответствующих .c-файлов `lexer.c`, `parser.c`, ..., `main.c`, компилируя каждый из них с помощью компилятора C. В последней строке `makefile` указано, что все семь объектных файлов зависят от глобального заголовочного файла `global.h`.

Транслятор создается посредством ввода команды `make`, что приводит к выполнению следующих команд.

```
cc -c lexer.c
cc -c parser.c
cc -c emitter.c
cc -c symbol.c
cc -c init.c
cc -c error.c
cc -c main.c
cc lexer.o parser.o emitter.o symbol.o \
init.o error.o main.o -o trans
```

Впоследствии компиляция будет выполняться только для файла, в который были внесены изменения со времени последней компиляции, и зависящих от него файлов. Примеры использования `make` для упрощения создания компилятора можно найти в [247]. □

Еще одним полезным инструментом при разработке компиляторов является профайлер. После создания компилятора профайлер может использоваться для определения того, какое время затрачивается при компиляции на решение различных задач. Определение “горячих точек” компилятора и их изменение могут ускорить работу компилятора в два-три раза.

В дополнение к общим инструментам для разработки программного обеспечения, при разработке компиляторов используются специализированные инструменты. В разделе 3.5 мы описали генератор Lex, который может быть использован для автоматизации создания лексических анализаторов по их спецификациям в виде регулярных выражений; в разделе 4.9 рассмотрен генератор Yacc, который позволяет автоматически создавать LR-анализаторы по грамматическому описанию синтаксиса языка. Рассмотренная выше команда `make` при необходимости автоматически вызывает Lex и Yacc. В дополнение к генераторам лексических и синтаксических анализаторов, созданы генераторы атрибутных грамматик и генераторов кода, облегчающие создание соответствующих компонентов компилятора. Многие из этих инструментов обеспечивают поиск ошибок в спецификациях компиляторов.

По поводу эффективности и удобства генераторов программ при построении компиляторов велись определенные споры [443]. Однако нельзя отрицать тот факт, что хорошо реализованный генератор может оказаться неоценимую помощь при разработке надежных

компонентов компилятора. Корректный синтаксический анализатор гораздо проще получить при использовании грамматического описания языка и генератора анализаторов, чем при кодировании его вручную. Важным вопросом, однако, является то, насколько корректно эти генераторы работают в связке друг с другом и с другими программами. Обычная ошибка при разработке генераторов состоит в предположении о том, что это и есть центральная задача всего построения компиляторов. Однако намного лучшие результаты приносит конструкция, в которой генератор производит подпрограммы с четким интерфейсом, которые могут вызываться другими программами¹ ([219]).

11.4. Тестирование и сопровождение

Компилятор должен генерировать корректный код. В идеале хотелось бы, чтобы компьютер механически проверял, что реализация компилятора в точности соответствует его спецификациям. В ряде статей обсуждается корректность различных алгоритмов компиляции, но, к сожалению, спецификации компиляторов редко определяются таким образом, что произвольную реализацию можно механически проверить на соответствие формальной спецификации. Поскольку компилятор — весьма сложная функция, встает вопрос о проверке корректности самих спецификаций компилятора.

На практике, чтобы убедиться, что компилятор работает удовлетворительно, мы должны обратиться к некоторым систематическим методам проверки. Один из подходов, успешно используемый многими разработчиками компиляторов, представляет собой “регрессивный” тест. Здесь используется комплект проверочных программ, и при каждом изменении компилятора эти программы компилируются с помощью как новой, так и старой его версии. Для автоматизации тестирования может также использоваться и команда *make* системы Unix.

Выбор программ для включения в текстовый набор — непростая задача. Нам желательно, чтобы тестовые программы проверили каждую инструкцию компилятора, по меньшей мере, один раз. Обычно для разработки такого набора тестов нужна немалая изобретательность. Для некоторых языков созданы такие исчерпывающие тесты (Fortran, C, TeX и другие). Многие разработчики компиляторов добавляют к регрессивным тестам программы, которые выявляют ошибки в предыдущих версиях их компиляторов; весьма грустно, когда старая ошибка появляется вновь вследствие новых исправлений...

Важны также тесты на производительность. Некоторые разработчики компиляторов проверяют, чтобы новые версии компиляторов генерировали код не хуже, чем предыдущие версии, включая время работы как часть регрессионного теста.

Сопровождение компилятора представляет собой отдельную важную задачу, в особенности если компилятор создается для работы в различных средах или в ситуации постоянной смены кадров в команде, работающей над проектом. Критичным элементом возможности сопровождения является хороший стиль программирования и хорошая документация. Авторам знаком один компилятор, в исходных текстах которого целых семь комментариев, причем один из них звучит как “Это проклятое место...”. Что и говорить, эту программу непросто сопровождать кому угодно — разве что кроме ее автора.

¹ В скользь заметим, что даже при таком подходе требуется решать проблему глобальных имен — в особенности при использовании в одной программе нескольких автоматически генерируемых анализаторов. — Прим. ред.

Кнут в работе [261] разработал систему под названием WEB (не путать с WWW!), предназначенную для решения задачи документации больших программ, написанных на языке Pascal. WEB упрощает грамотное программирование; документация разрабатывается одновременно с кодом, а не постфактум. Многие идеи WEB применимы и к другим языкам программирования².

² В настоящее время имеется ряд программных продуктов, облегчающих создание документации, а в некоторых языках (типа Java) такие возможности являются встроенными. — Прим. ред.

ГЛАВА 12

Некоторые компиляторы

В этой главе обсуждается структура некоторых существующих компиляторов для языка форматирования текста, языков Pascal, C, Fortran, Bliss и Modula 2. Наша задача — не объяснить, почему было отдано предпочтение тому или иному подходу в разработке компиляторов, а проиллюстрировать широту, возможную в реализациях компиляторов¹.

Компиляторы Pascal выбраны в связи с их влиянием на сам язык; компиляторы С — поскольку С является основным языком программирования в UNIX; Fortran H — в связи с его влиянием на разработку технологий оптимизации. BLISS/11 иллюстрирует создание компилятора, задача которого — оптимизация размера программы. Компилятор Modula 2 для DEC выбран в связи с использованием относительно простых технологий для создания превосходного кода, что позволило создать этот компилятор за несколько месяцев усилиями одного человека.

12.1. Препроцессор математических формул EQN

Для многих компьютерных программ множество возможных входных потоков можно рассматривать как небольшой язык. Структура такого множества может быть описана грамматикой, а для точного определения выполняемых программой действий можно использовать синтаксически управляемую трансляцию. После такого решения для реализации программы можно использовать технологии, применяемые при разработке компиляторов.

Одним из первых компиляторов для небольших языков в среде программирования UNIX был EQN [246]. Как вкратце описано в разделе 1.2, EQN получает во входном потоке выражения типа “ $E \text{ sub } 1$ ” и генерирует команды для программы форматирования текста TROFF, которая выводит в конечном счете “ E_1 ”.

Схема реализации EQN показана на рис. 12.1. Макропрепроцессор (см. раздел 1.4) и лексический анализатор работают совместно. Поток токенов, полученный после лексического анализа, транслируется в процессе синтаксического анализа в команды форматирования текста. Транслятор построен с применением генератора синтаксических анализаторов Yacc, описанного в разделе 4.9.

Подход, состоящий в рассмотрении множества входных потоков для EQN как языка и применении технологий построения компиляторов для создания транслятора, имеет ряд преимуществ, замеченных авторами.

1. *Простота реализации.* “Построение работающей системы, успешно справляющейся со сложными примерами, потребовало примерно человека-месяца работы”.

¹ К изложенному в этой главе следует подходить как к учебному материалу, имеющему к тому же определенную историческую ценность, а не как к описанию реально используемых компиляторов языков программирования. — Прим. ред.

2. **Эволюция языка.** Синтаксически управляемое определение упрощает внесение изменений во входной язык. С течением времени EQN эволюционировал, обеспечивая более полное соответствие нуждам пользователей.

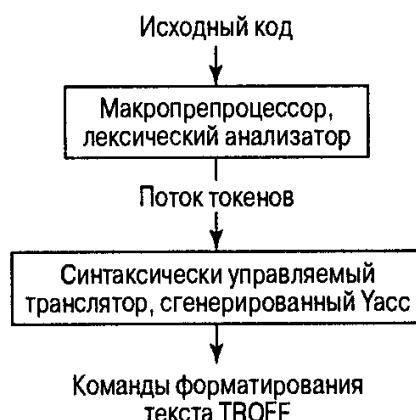


Рис. 12.1. Реализация EQN

Авторы заключают, что “определение языка и построение компилятора для него с помощью компилятора компиляторов является практически единственным разумным подходом”.

12.2. Компиляторы Pascal

Разработка языка Pascal и создание первого компилятора для него были, по словам Вирта [456], взаимозависимыми задачами. Поэтому весьма поучительно рассмотреть структуру компиляторов языка, разработанного Виртом и его коллегами. Первый [456] и второй [31, 32] компиляторы генерировали абсолютный машинный код для машин серии CDC 6000. Эксперименты по переносимости со вторым компилятором привели к компилятору Pascal-P, который генерировал специальный код, именуемый Р-кодом, для абстрактной стековой машины [335].

Каждый из указанных компиляторов представляет собой однопроходный компилятор на основе синтаксического анализатора с рекурсивным спуском, наподобие начальной стадии, описанной в главе 2, “Простой однопроходный компилятор”. По словам Вирта, создание языка, соответствующего ограничениям метода, оказалось относительно простым. Организация компилятора Pascal-P показана на рис. 12.2.



Рис. 12.2. Компилятор Pascal-P

Основные операции абстрактной стековой машины, используемой компилятором Pascal-P, отражают требования Pascal. Память машины разделена на четыре области:

1. код процедур,
2. константы,
3. стек для записей активации,
4. куча для данных, распределяемых с помощью оператора new.²

Поскольку процедуры в Pascal могут быть вложенными, запись активации процедуры содержит как связь доступа, так и связь управления. Вызов процедуры транслируется в инструкцию абстрактной машины “пометить стек” со связями доступа и управления в качестве параметров. Код процедуры обращается к памяти для локальных переменных посредством указания смещения относительно конца записи активации. Обращение к памяти, выделенной нелокальным переменным, выполняется с помощью пар, состоящих из числа проходимых связей доступа и смещения (способ, рассмотренный в разделе 7.4). Первый из рассматриваемых компиляторов для эффективного доступа к нелокальным переменным использовал дисплей.

Из опыта написания второго из рассматриваемых компиляторов в [32] сделан вывод, что, с одной стороны, однопроходный компилятор прост в реализации и требует умеренного использования операций ввода-вывода (код тела процедуры компилируется в памяти и записывается в виде модуля в файл). С другой стороны, однопроходная организация ограничивает качество генерируемого кода и требует для работы относительно много памяти.

12.3. Компиляторы С

С является языком программирования общего назначения, разработанным Д. Ритчи (D. M. Ritchie), и используется как основной язык программирования в операционной системе UNIX [374]. UNIX сама написана на С и распространена на множество машин (от микропроцессоров до больших универсальных компьютеров) путем первоначального распространения компиляторов С. Здесь мы вкратце опишем структуру компилятора Ритчи для PDP-11 [373] и рассмотрим семейство переносимых компиляторов РСС [217]. Три четверти кода РСС не зависят от целевой машины. Все эти компиляторы двухпроходные; компилятор для PDP-11 использует необязательный третий проход для оптимизации вывода на языке ассемблера, как показано на рис. 12.3. Эта фаза локальной оптимизации устраняет избыточный и бесполезный код.

Первый проход каждого компилятора выполняет лексический и синтаксический анализ, а также генерацию промежуточного кода. Для разбора всего, кроме выражений (при разборе которых используется приоритет операторов), компилятор для PDP-11 использует метод рекурсивного спуска. Промежуточный код состоит из выражений в постфиксной записи и ассемблерного кода для других конструкций. РСС использует синтаксический LALR(1)-анализатор, сгенерированный Yacc. Его промежуточный код содержит префиксную запись выражений и ассемблерный код для прочих конструкций. В каждом случае распределение памяти для локальных имен выполняется в первом проходе, так что обращение к этим именам осуществляется с помощью смещений в записи активации.

² Раскрутка облегчается тем, что компилятор, написанный на подмножестве компилируемого языка, использует кучу как стек, так что изначально может применяться простейший диспетчер кучи.

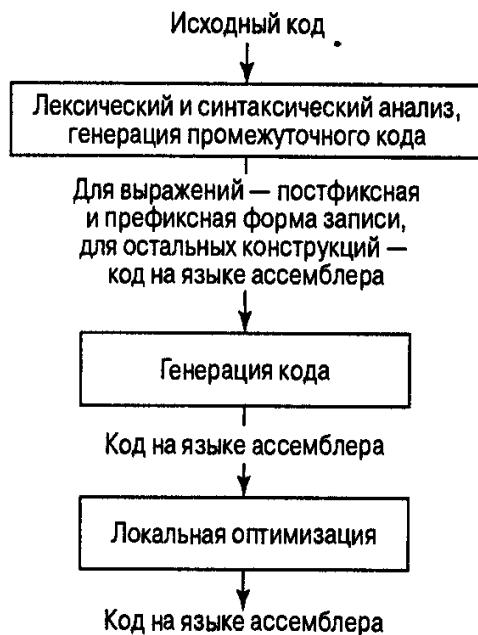


Рис. 12.3. Структура проходов компиляторов С

На заключительном этапе выражения представлены синтаксическими деревьями. В компиляторе PDP-11 генерация кода реализуется путем прохода по дереву с применением стратегии, схожей с алгоритмом назначения меток из раздела 9.10. Для гарантии использования в операциях, где это необходимо, пар регистров, а также для получения возможных преимуществ константных операндов в этот алгоритм были внесены модификации.

В [216] приведен обзор влияния теории компиляции на реализацию РСС. Как в РСС, так и в РСС2 (следующей версии этого компилятора) код для выражений генерируется посредством преобразования дерева. Генератор кода РСС рассматривает инструкции исходного языка по одной, находит максимальные поддеревья, которые могут быть вычислены без сохранения, с использованием доступных регистров. Метки, вычисляемые методом, аналогичным описанному в разделе 9.10, определяют подвыражения, которые вычисляются и сохраняются во временных переменных. Код для вычисления и сохранения значений, представленных этими поддеревьями, генерируется компилятором в процессе выбора поддеревьев. Преобразование деревьев более ясно представлено в РСС2, генератор кода которого основан на алгоритме динамического программирования из раздела 9.11.

В [220] описано влияние целевой машины на конструкцию записей активации и последовательность вызовов/возвратов из процедур. Функция стандартной библиотеки `printf` может иметь переменное количество аргументов, так что разработка последовательности вызова на некоторых машинах определяется требованиями допустимости списка аргументов переменной длины.

12.4. Компиляторы Fortran H

Исходный компилятор Fortran H [294] был объемистым и весьма мощным оптимизирующим компилятором, который использовал методы оптимизации, впоследствии описанные в данной книге. Предпринимались попытки увеличения производительности; были разработаны “расширенная” версия компилятора для IBM/370 и “улучшенная” версия

[390]. Fortran H предоставляет пользователю выбор компиляции без оптимизации, с оптимизацией только использования регистров или с полной оптимизацией. Схема компилятора в случае полной оптимизации приведена на рис. 12.4.

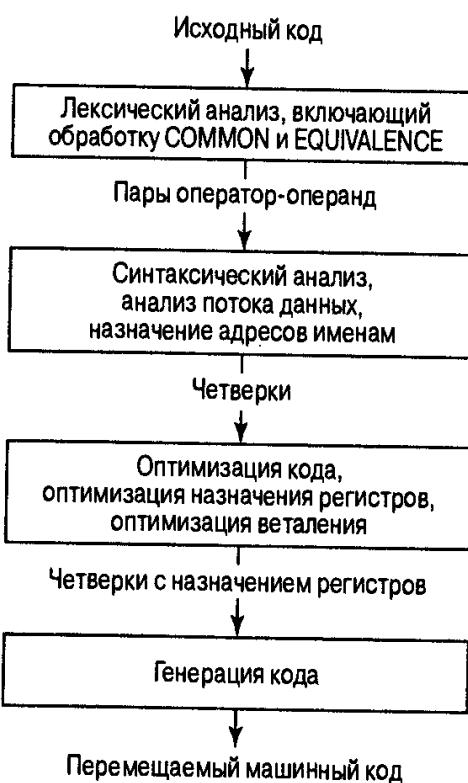


Рис. 12.4. Схема компилятора Fortran H

Исходный текст разбирается в четыре прохода. Первые два выполняют лексический и синтаксический анализ, выдавая промежуточное представление в виде четверок. Следующий проход включает в себя оптимизацию кода и использования регистра, а последний генерирует объектный код на основе четверок и распределения регистров.

Фаза лексического анализа несколько необычна, поскольку ее выход представляет собой не поток токенов, а поток пар оператор-операнд, которые примерно эквивалентны токену операнда с предшествующим токеном не-операнда. Следует заметить, что в Fortran, как и в большинстве языков программирования, мы никогда не встречаемся с двумя идущими подряд токенами операндов, таких как идентификаторы или константы; два таких токена всегда разделяются как минимум одним токеном пунктуации.

Например, инструкция присвоения $A = B(I) + C$ транслируется в следующую последовательность пар.

“Инструкция присвоения”		A
=		B
(I
)		-
+		C

Фаза лексического анализа различает левые скобки, которые служат ограничителем списка параметров или индексов, и скобки, группирующие операнды; приведенный ранее символ “(s ” представляет левую скобку, используемую в качестве оператора индексирования. Так как за правой скобкой никогда не идет операнд, Fortran H не различает две роли правых скобок.

Обработка инструкций COMMON и EQUIVALENCE связана с лексическим анализом. На этой стадии возможна разметка блоков памяти COMMON и блоков, связанных с подпрограммами, и определение расположения каждой переменной, упоминаемой программой в одной из этих статических областей памяти.

Поскольку в Fortran нет структурированных управляющих инструкций типа `while`, синтаксический анализ (за исключением выражений) достаточно прост, а для выражений Fortran Н использует разбор с использованием приоритета операторов. В процессе генерации четверок выполняется некоторая простая локальная оптимизация; например, умножение на степень двойки заменяется операцией битового сдвига влево.

Оптимизация кода в Fortran Н

Каждая подпрограмма разбивается на базовые блоки, и циклическая структура находится путем поиска дуг графа потока, головы которых доминируют над их хвостами (см. раздел 10.4). Компилятор выполняет следующую оптимизацию.

1. *Устранение общих подвыражений.* Компилятор ищет локальные общие подвыражения и выражения, общие для блока B и одного или нескольких блоков, над которыми доминирует B . Прочие вхождения общих подвыражений не рассматриваются. Кроме того, поиск общих подвыражений выполняется по одному, а не с использованием метода битовых векторов из раздела 10.6. Интересно, что в “улучшенной” версии компилятора авторы пришли к выводу о возможности существенного ускорения при применении метода битовых векторов.
2. *Перемещение кода.* Инвариантные относительно цикла инструкции удаляются из цикла так, как описано в разделе 10.7.
3. *Распространение копий.* Эта задача выполняется по одной инструкции копирования за один раз.
4. *Устранение переменных индукции.* Эта оптимизация выполняется только для переменных, присвоение которым производится в цикле один раз. Вместо подхода, основанного на семействах переменных (см. раздел 10.7), для поиска переменных, принадлежащих некоторому семейству другой переменной индукции, выполняются многократные проходы по коду.

Хотя анализ потока данных выполняется в стиле “по одному элементу”, значения, соответствующие рассмотренным нами множествам *in* и *out*, хранятся как битовые векторы. Однако в исходном компиляторе эти векторы имеют ограниченную длину — 127, так что в больших программах оптимизация касается только наиболее часто используемых переменных. Улучшенная версия увеличивает предельное значение, но не устраняет ограничение.

Алгебраическая оптимизация

Поскольку Fortran часто используется для числовых вычислений, алгебраическая оптимизация достаточно опасна; преобразования выражения в компьютерной арифметике могут привести к переполнению или потере точности, которые не видны при идеализированном взгляде на алгебраические упрощения. Однако алгебраические преобразования целочисленной арифметики, как правило, безопасны, и улучшенная версия компилятора выполняет оптимизацию такого вида только при работе с массивами.

В общем случае при обращении к элементу массива типа A(I, J, K) производится вычисление смещения вида $aI+bJ+cK+d$; точные значения констант зависят от расположения A и размерностей массива. Если, скажем, I и K представляют собой константы (либо числовые, либо переменные, инвариантные относительно цикла), то компилятор применяет коммутативный и ассоциативный законы для генерации выражения $bJ+e$, где $e = aI+cK+d$.

Оптимизация регистров

Fortran H разделяет регистры на три класса. Эти множества регистров используются для локальной оптимизации регистров, глобальной оптимизации регистров и оптимизации ветвления. Точное число регистров в каждом классе может быть уточнено компилятором в допустимых пределах.

Глобальные регистры распределяются по циклам на основе частоты использования переменных в циклах. Переменная, которая должна храниться в регистре в цикле L, но не в цикле, непосредственно следующем за L, загружается в предзаголовке L и сохраняется при выходе из L.

Локальные регистры используются в базовом блоке для хранения результатов одной инструкции до их использования в последующей инструкции (или инструкциях). Временные значения сохраняются в памяти только в том случае, когда локальных регистров недостаточно. Компилятор пытается вычислять новые значения в регистре, хранящем один из операндов, если после этого операнд оказывается неактивен. В усовершенствованной версии делаются попытки распознать ситуации, когда глобальные регистры можно обменять с другими регистрами для увеличения количества операций, результат которых сохраняется на месте одного из их операндов.

Оптимизация ветвления является артефактом набора инструкций IBM/370, который дает значительный выигрыш при переходах в адреса, выражаемые содержимым некоторого регистра плюс константа в диапазоне от 0 до 4095. В соответствии с этим Fortran H выделяет некоторые регистры для хранения адресов в кодовом пространстве с интервалом в 4096 байт для повышения эффективности переходов во всех программах, за исключением очень больших.

12.5. Компилятор Bliss/11

Этот компилятор реализует язык программирования Bliss на машине PDP-11 [460]. В каком-то смысле это оптимизирующий компилятор из “уходящего” мира, в котором так высоко ценилась память, что уменьшение размера программы было куда важнее повышения ее быстродействия. Однако большинство оптимизирующих преобразований сохраняет как пространство, так и время работы, и наследники этого компилятора используются и сегодня.

Этот компилятор заслуживает нашего внимания по ряду причин. Он обладает мощным оптимизатором, причем многие из его оптимизаций не встречаются почти nowhere. Кроме того, он является пионером синтаксически управляемого подхода к оптимизации, рассмотренного в разделе 10.5. Соответственно, язык Bliss был разработан так, что производит только приводимые графы потока (в нем нет безусловных переходов). Таким образом, анализ потока данных может выполняться непосредственно над деревом разбора, а не над графиком потока.

Компилятор однопроходен, причем он полностью завершает работу над одной процедурой, прежде чем приступить к другой. С точки зрения разработчика компилятор состоит из пяти модулей, показанных на рис. 12.5.

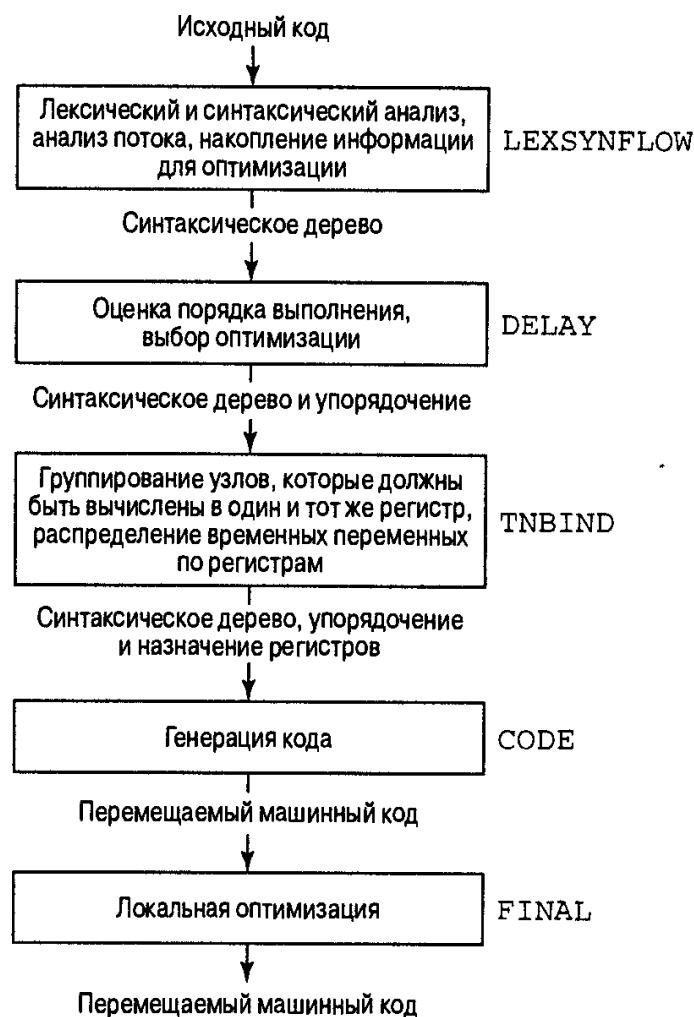


Рис. 12.5. Компилятор Bliss/11

LEXSYNFLOW выполняет лексический и синтаксический анализ с использованием метода рекурсивного спуска. Поскольку в Bliss нет оператора безусловного перехода, все графы потоков процедур Bliss приводимы. Синтаксис языка позволяет нам строить граф потока и определять циклы и их входы в процессе синтаксического анализа. LEXSYNFLOW так и делает, а также находит общие подвыражения и разновидность ои- и ио-цепочек, пользуясь преимуществами структуры приводимых графов потока. Еще одна важная задача LEXSYNFLOW состоит в поиске групп подобных выражений, являющихся кандидатами для замещения единой подпрограммой. Заметим, что такое замещение при экономии размера целевого кода делает ее несколько более медленной.

Модуль DELAY рассматривает синтаксическое дерево для определения того, какие из обычных методов оптимизации, таких как перемещение инвариантного кода или устранение общих подвыражений, могут привести к уменьшению размера программы. В это же время определяется порядок вычисления выражений на основе стратегии назначения меток (см. раздел 9.10), модифицированной таким образом, чтобы учсть недоступные регистры (используемые для хранения значений общих подвыражений). Для определения того, должно ли выполняться переупорядочение вычислений, используются алгебраические законы. Условные выражения вычисляются либо численно, либо с помощью

потока управления (см. раздел 8.4); при этом модуль *DELAY* определяет, какой из методов более выгоден в каждом конкретном случае.

TNBIND определяет, какие временные переменные должны быть связаны с регистрами, и распределяет как регистры, так и память. Используемая стратегия сначала группирует узлы синтаксического дерева, которые должны быть назначены одному и тому же регистру. Как обсуждалось в разделе 9.6, имеется определенная выгода от вычисления узла в тот же регистр, что и у одного из его предков. Выигрыш, получаемый в результате хранения временных переменных в регистрах, особенно велик там, где такие переменные многократно используются на небольшом интервале. В первую очередь регистры назначаются узлам, для которых выигрыш наиболее велик. Модуль *CODE* преобразует дерево вместе с его упорядочением и информацией о назначении регистров в перемещаемый машинный код.

Затем этот код изучается модулем *FINAL*, который выполняет его локальную оптимизацию до тех пор, пока она не перестает давать результат. Выполняемые преобразования включают устранение (условных или безусловных) переходов к переходам и дополнение условных переходов, рассмотренные в разделе 9.9.

Избыточный и бесполезный код (который может оказаться результатом других методов оптимизации модуля *FINAL*) устраняется. Предпринимаются также попытки объединения аналогичных последовательностей кода по двум путям ветвления, а также локального распространения констант. Выполняются и другие локальные оптимизации, часть из которых машинно-зависима (например, для PDP-11 инструкции перехода по возможности заменяются “ветвями”, которые состоят из одного слова, но ограничены переходом в диапазоне 128 слов).

12.6. Оптимизирующий компилятор Modula-2

Этот компилятор, описанный в [351], разработан специально для генерации хорошего кода с использованием оптимизации, обеспечивающей высокую отдачу при небольших усилиях. Автор описывает свою стратегию как поиск “наилучшей простой (best simple)” оптимизации. Эта идея трудна в реализации — без экспериментов и измерений трудно заранее решить, какая именно оптимизация наиболее выгодна. Некоторые из решений, реализованных в компиляторе Modula-2, вероятно, не подходят для компилятора, обеспечивающего максимальную оптимизацию. Тем не менее, эта стратегия позволила автору добиться цели — получение превосходного кода с помощью компилятора, написанного одним человеком за несколько месяцев. Пять проходов начальной стадии компиляции показаны на рис. 12.6.

Синтаксический анализатор сгенерирован с использованием Yacc и производит синтаксические деревья в два прохода, поскольку переменные в Modula не обязаны быть объявлены до их использования. Была предпринята попытка сделать данный компилятор совместимым с существующими. Так, промежуточный код представляет собой Р-код, обеспечивающий совместимость со многими компиляторами Pascal. Формат вызова процедур этого компилятора согласуется с форматом Pascal и компилятором С в Berkeley UNIX³, так что процедуры, написанные на трех языках, могут быть легко интегрированы в одной программе.

³ Имеется в виду порядок передачи параметров вызываемой процедуре (слева направо, как в С, или справа налево, как в Pascal) и очистка стека вызываемой процедурой (как в Pascal) или вызывающей (как в С) — что, например, указывается в С с помощью описания функций как `_pascal` или `_cdecl`. — Прим. ред.

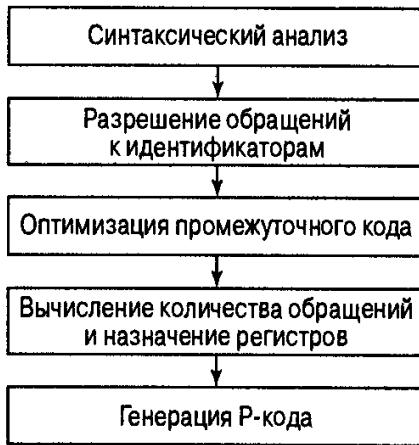


Рис. 12.6. Проходы компилятора Modula-2

Компилятор не выполняет анализа потока данных. Поскольку Modula-2, как и Bliss, представляет собой язык, производящий исключительно приводимые графы потока, в нем может использоваться методология из раздела 10.5. В действительности компилятор Modula продолжает путь Bliss-11 по использованию преимуществ синтаксиса. Циклы идентифицируются в соответствии с их синтаксисом, т.е. компилятор просматривает `while` и `for` конструкции в программе. Инвариантные выражения находятся на основе того факта, что никакие из их переменных не определяются в цикле, и выносятся в заголовок цикла. Определение переменных индукции выполняется только для переменных из семейства индекса цикла `for`. Глобальные общие подвыражения находятся, когда одно из них располагается в блоке, доминирующем над блоком с другим подвыражением, но такой анализ выполняется для каждого подвыражения отдельно, без битовых векторов.

Стратегия распределения регистров вполне разумна, хотя и не всеобъемлюща. В частности, в качестве кандидатов на размещение в регистрах она рассматривает только

1. временные переменные, используемые в процессе вычисления выражений (получающие наивысший приоритет при распределении регистров);
2. значения общих подвыражений;
3. индексы и предельные значения циклов `for`;
4. адреса E в выражениях вида `with E do`;
5. простые переменные (символьные, целые и т.п.), локальные по отношению к рассматриваемой процедуре.

Делается попытка оценить значения выигрыша при хранении каждой переменной классов (2)–(5) в регистре в предположении, что инструкция выполняется 10^d раз в случае вложенности в d циклов. Однако переменные, обращение к которым выполняется не более двух раз, не рассматриваются как возможные кандидаты на хранение в регистрах. Прочие переменные упорядочиваются в соответствии с выполненной оценкой и назначаются регистрам, доступным после присвоения временных переменных в процессе вычисления выражений и переменных более высокого приоритета.

ПРИЛОЖЕНИЕ А

Программный проект

A.1. Введение

В этом приложении вашему вниманию предлагается программное упражнение для практических занятий, соответствующих учебному курсу по разработке компиляторов, основанному на данной книге. Упражнение состоит в реализации базовых компонентов компилятора для подмножества языка программирования Pascal. Это подмножество минимально, но обеспечивает возможность создания таких программ, как использованная в разделе 7.1 процедура рекурсивной сортировки. То, что данный язык является подмножеством существующего, дает определенные преимущества. Смысл программы, написанной с использованием данного подмножества, определяется семантикой Pascal [214]. При доступности компилятора Pascal он может использоваться для проверки работы компилятора, создаваемого в качестве упражнения. Конструкции данного подмножества встречаются во многих языках программирования, так что соответствующее упражнение может быть легко переформулировано для использования другого языка программирования в случае недоступности компилятора Pascal.

A.2. Структура программы

Программа состоит из последовательности объявлений глобальных данных, последовательности процедур и объявлений функций, а также составной инструкции, представляющей собой “основную программу”. Глобальные данные хранятся в статически выделяемой памяти. Данные, локальные по отношению к процедурам и функциям, размещаются в стеке. Рекурсия разрешена, и параметры передаются по ссылке. Процедуры `read` и `write` обеспечивает компилятор.

На рис. A.1 приведен пример программы. Имя программы — `example`, `a input` и `output` — имена файлов, используемые процедурами соответственно `read` и `write`.

```
program example(input, output);
var x, y: integer;
function gcd(a, b: integer): integer;
begin
  if b = 0 then gcd := a;
  else gcd := gcd(b, a mod b)
end;

begin
  read(x, y);
  write(gcd(x, y))
end.
```

Рис. A.1. Пример программы

A.3. Синтаксис подмножества Pascal

Ниже приведена LALR(1)-грамматика подмножества Pascal. Данная грамматика может быть модифицирована для синтаксического анализа методом рекурсивного спуска путем устранения левой рекурсии, как описано в разделах 2.4 и 4.3. Синтаксический анализатор на основе приоритета операторов для выражений может быть построен замещением *relop*, *addop* и *mulop* и устранением ϵ -продукций.

Добавление продукции

statement \rightarrow if *expression* then *statement*

приводит к неоднозначности “кочующего else”, которую можно устраниить описанным в разделе 4.3 способом (см. также пример 4.19 в случае применения предиктивного синтаксического анализатора).

Синтаксического отличия между простой переменной и вызовом функции без параметров нет. Оба генерируются продукцией

factor \rightarrow id

Таким образом, присвоение a := b устанавливает a равным значению, возвращаемому функцией b, если b представляет собой функцию.

program \rightarrow

program id (*identifier_list*) ;

declarations

subprogram_declarations

compound_statement

identifier_list \rightarrow

id

| *identifier_list* , id

declarations \rightarrow

declarations var identifier_list : type ;

| ϵ

type \rightarrow

standard_type

| array [num .. num] of *standard_type*

standard_type \rightarrow

integer

| real

subprogram_declarations \rightarrow

subprogram_declarations subprogram_declaration ;

| ϵ

subprogram_declaration \rightarrow

subprogram_head declarations compound_statement

subprogram_head \rightarrow

function id *arguments* : *standard_type* ;

| procedure id *arguments* ;

$arguments \rightarrow$
 (parameter_list)
 | ϵ
 $parameter_list \rightarrow$
 identifier_list : type
 | parameter_list ; identifier_list : type
 $compound_statement \rightarrow$
begin
 optional_statements
end
 $optional_statements \rightarrow$
 statement_list
 | ϵ
 $statement_list \rightarrow$
 statement
 | statement_list ; statement
 $statement \rightarrow$
 variable assignop expression
 | procedure_statement
 | compound_statement
 | if expression then statement else statement
 | while expression do statement
 $variable \rightarrow$
 id
 | id [expression]
 $procedure_statement \rightarrow$
 id
 | id (expression_list)
 $expression_list \rightarrow$
 expression
 | expression_list , expression
 $expression \rightarrow$
 simple_expression
 | simple_expression relop simple_expression
 $simple_expression \rightarrow$
 term
 | sign term
 | simple_expression addop term
 $term \rightarrow$
 factor
 | term mulop factor
 $factor \rightarrow$
 id
 | id (expression_list)

```

| num
| ( expression )
| not factor

sign →
+ | -

```

A.4. Лексические соглашения

Система записи токенов взята из раздела 3.3.

- Комментарии заключаются в фигурные скобки { и }. Они не могут содержать {. Комментарии могут располагаться после любого токена.
- Пробелы между токенами необязательны, однако ключевые слова должны быть окружены пробелами, символами новой строки, находиться в начале программы или завершаться заключительной точкой.
- Токен id идентификатора представляет собой букву, за которой следуют буквы или цифры.

```

letter → [a-zA-Z]
digit → [0-9]
id → letter ( letter | digit ) *

```

При реализации может быть ограничена допустимая длина идентификатора.

- Токен num соответствует беззнаковым числам (см. пример 3.5).

```

digits → digit digit*
optional_fraction → . digits | ε
optional_exponent → ( E ( + | - | ε ) digits ) | ε
num → digits optional_fraction optional_exponent

```

- Ключевые слова зарезервированы и в грамматике обозначаются полужирным шрифтом.
- Операторы отношения (relop) представляют собой =, <>, <, <=, >= и >. Обратите внимание, что <> означает ≠.
- Операторами addop являются +, - и or.
- Операторы mulop представляют собой *, /, div, mod и and.
- Лексемой токена assignop является :=.

A.5. Предлагаемые упражнения

Программный практикум для односеместрового курса состоит в написании интерпретатора для определенного выше языка или для аналогичного подмножества другого языка высокого уровня. Проект включает трансляцию исходной программы в промежуточное представление (в виде четверок или кода стековой машины) и интерпретацию промежуточного представления. Мы предлагаем следующий порядок создания модулей, отличающийся от порядка их работы при компиляции, поскольку удобно иметь работающий интерпретатор для отладки других компонентов компилятора.

1. *Разработка механизма таблицы символов.* Решение об организации таблицы символов должно обеспечивать сбор информации об именах, оставляя структуру записи таблицы символов по возможности более гибкой. Напишите подпрограммы для следующих действий.
 - i) Поиск имени в таблице символов, создание новой записи для отсутствующего имени и возврата указателя на запись для имени в обоих случаях.
 - ii) Удаление из таблицы символов всех имен, локальных для данной процедуры.
2. *Создание интерпретатора для четверок.* Вопрос о точном множестве четверок в настоящий момент может оставаться открытым, но оно должно включать арифметические операторы и инструкции условных переходов, соответствующие множеству операторов языка. Кроме того, множество включает логические операторы, если условия вычисляются арифметически, а не по положениям в программе. Кроме того, необходимы “четверки” для преобразования целых чисел в действительные, для маркировки начала и конца процедур, а также для передачи параметров и вызова процедур. Необходимо также разработать вызывающую последовательность и организацию среды времени выполнения интерпретируемой программы. Простая стековая организация, рассмотренная в разделе 7.3, вполне подходит для нашего языка, поскольку в нем не разрешены вложенные объявления процедур, так что все переменные либо являются глобальными, объявленными на уровне всей программы, либо локальны по отношению к простой процедуре. Для простоты вместо интерпретатора может быть использован другой язык высокого уровня. Каждая четверка может являться инструкцией языка высокого уровня типа С или даже Pascal. Вывод компилятора в таком случае представляет собой последовательность инструкций С, которая может быть скомпилирована существующим компилятором С.
3. *Создание лексического анализатора.* Выберите внутренние коды для токенов. Решите, как константы будут представляться в компиляторе. Решите задачу подсчета считанных строк для корректного отображения места обнаружения ошибки. Учтите, что при необходимости лексический анализатор должен выводить листинг исходной программы. Разработайте программу для внесения ключевых слов в таблицу символов. Разработайте лексический анализатор как подпрограмму, вызываемую синтаксическим анализатором, и возвращающую пару (токен, значение атрибута). На этом этапе при обнаружении лексическим анализатором ошибки ее обработка может сводиться к выводу соответствующего сообщения и прекращению работы программы.
4. *Разработка семантических действий.* Напишите семантические программы для генерации четверок. Грамматика потребует внесения изменений для упрощения трансляции — обратитесь к разделам 5.5 и 5.6 за примерами таких изменений грамматики. Обеспечьте на этом этапе выполнение семантического анализа, преобразуя при необходимости целые числа в действительные.
5. *Разработка синтаксического анализатора.* Если у вас имеется генератор LALR-анализаторов, это значительно упростит вашу задачу. Если такой генератор способен обрабатывать неоднозначности грамматики, как Yacc, то нетерминалы, означающие выражения, могут быть объединены. Кроме того, неоднозначность “кочующего else” может быть разрешена путем переноса при возникновении конфликта переноса/свертки.
6. *Создание подпрограмм обработки ошибок.* Разработайте восстановление после лексических и синтаксических ошибок. Обеспечьте вывод диагностики для лексических, синтаксических и семантических ошибок.

7. *Вычисления.* Программа на рис. А.1 может служить в качестве простой тестовой программы. Другой тестовой программой может быть программа на рис. 7.1. Код для функции `partition` на этом рисунке соответствует помеченному фрагменту на рис. 10.2. Если у вас есть такая возможность, пропустите ваш компилятор через профайлер. Определите подпрограммы, занимающие основное время работы. Какие модули следует модифицировать для ускорения работы вашего компилятора?

A.6. Эволюция интерпретатора

Альтернативный подход к построению интерпретатора для нашего языка состоит в реализации настольного калькулятора, т.е. интерпретатора для выражений. Постепенное добавление конструкций к языкам этого калькулятора приведет к получению интерпретатора для всего языка. Аналогичный подход используется в книге [247]. Предлагается следующий порядок добавления конструкций.

1. *Трансляция выражений в постфиксную запись.* Используя метод рекурсивного спуска, описанный в главе 2, “Простой однопроходный компилятор”, либо генератор синтаксических анализаторов, освойтесь со средой программирования и напишите транслятор простых арифметических выражений в постфиксную запись.
2. *Добавление лексического анализатора.* Обеспечьте работу транслятора из предыдущего пункта с ключевыми словами, идентификаторами и числами. Переделайте транслятор для генерации кода стековой машины или четверок.
3. *Создание интерпретатора для промежуточного представления.* Как говорилось в разделе А.5, вместо интерпретатора может использоваться язык высокого уровня. В настоящий момент интерпретатор должен поддерживать только арифметические операции, присвоение и операции ввода-вывода. Расширьте язык путем добавления объявлений глобальных переменных, присвоений и вызовов процедур `read` и `write`. Наличие этих конструкций обеспечивает возможность тестирования интерпретатора.
4. *Добавление инструкций.* Программа на нашем языке теперь состоит из основной программы без объявлений подпрограмм. Протестируйте транслятор и интерпретатор.
5. *Добавление процедур и функций.* Таблица символов должна теперь обеспечивать ограничение области видимости идентификаторов телом процедуры. Разработайте последовательность вызова (простая организация стека из раздела 7.3 вполне адекватна поставленной задаче). Расширьте интерпретатор для поддержания последовательности вызова.

A.7. Расширения

Имеется ряд возможностей, которые могут быть добавлены в язык без существенного повышения сложности компиляции.

1. Многомерные массивы.
2. Инструкции `for` и `case`.
3. Блочная структура.
4. Структуры записей.

Если позволяет время — добавьте одно или несколько из этих расширений в ваш компилятор.

ПРИЛОЖЕНИЕ Б

Спецификации языков программирования

Материал данного приложения любезно предоставлен Е. А. Зуевым.

Приведенные в приложении описания представляют строгие, непротиворечивые и компактные грамматики языков программирования C++ и C#. Эти описания могут служить дополнительным материалом для изучения языков, оценки их синтаксической сложности по сравнению с другими языками программирования, а также быть основой для создания синтаксических анализаторов и аналогичных инструментов, ориентированных на рассматриваемые языки программирования.

Грамматики записаны с использованием входного языка генераторов синтаксических анализаторов семейства YACC/Bison. Грамматики непротиворечивы в смысле указанного входного языка (т.е. не содержат неразрешенных конфликтов типа свертка/свертка); все конфликты типа перенос/свертка явно разрешены посредством задания приоритетов лексем.

Являясь чисто синтаксическим описанием языков, данные грамматики фактически определяют некоторые их надмножества. Ряд ограничений семантического характера не нашли отражения в грамматиках, что позволило сделать их существенно более компактными. Кроме того, в описаниях представлен только синтаксис — иными словами, лексика языка не определяется, и лексемы языка (разделители, служебные слова, литералы) задаются в данном описании как предопределенные сущности. Синтаксис директив препроцессора и соответствующие лексемы также не определяются.

Б.1. Язык программирования C++

Источник: International Standard - Programming Languages - C++, ISO/IEC 14882, 1998 (E)

C++ — один из наиболее распространенных в мире языков программирования. Будучи разработанным сотрудником Bell Labs Бьерном Страуструпом на основе языка C, он вобрал в себя практически все известные к настоящему моменту достижения теории и практики программирования. C++ — мощный общеселевой язык программирования, который поддерживает различные парадигмы программирования: традиционное (процедурное) программирование, объектно-ориентированный подход к разработке программ, механизмы обобщенного программирования (generic programming), средства безопасного программирования (механизм исключительных ситуаций), а также (в некоторой степени) модульное программирование (пространства имен). Модель выполнения C++ ориентирована прежде всего на высокую эффективность создаваемых программ.

В 1990 году комитетом ISO была начата стандартизация языка, в процессе которой в него было внесено большое количество важных нововведений. В конце 1998 года был принят международный стандарт языка C++ (ISO/IEC 14882).

Необходимо отметить некоторые особенности представленного описания грамматики языка C++. Во-первых, в описание не вошли дополнительные служебные слова-эквиваленты операций (например, `and` как синоним знака операции `&&` или `or` как синоним `||`). Во-

вторых, поскольку изначальная (явно оговоренная в стандарте языка) неоднозначность грамматики C++ преодолевается распознаванием смысла идентификаторов в процессе лексического анализа, в представленном описании это решение моделируется введением четырех категорий идентификаторов:

- имя типа (лексема `lxmTypeName`): идентификатор, обозначающий имя ранее объявленного класса или перечисления либо введенный некоторым `typedef`-объявлением;
- имя шаблона (лексема `lxmTemplateName`): идентификатор, обозначающий имя ранее объявленного шаблона класса или функции;
- имя пространства имен (лексема `lxmNamespaceName`): идентификатор, обозначающий имя ранее объявленного пространства имен;
- идентификатор (лексема `lxmIdentifier`): любой идентификатор, не входящий в предыдущие категории.

Описанное решение не является единственно возможным. Известны успешные попытки преодоления неоднозначностей C++ без введения дополнительных категорий идентификаторов, однако результирующий синтаксис получается весьма громоздким и крайне трудным для понимания.

```
// Reference:  
// International Standard - Programming Languages - C++,  
// ISO/IEC 14882, 1998 (E)  
//  
// Copyright (c) 2001  
// Eugene A. Zueff  
// ETH Zurich, Switzerland  
// zueff@inf.ethz.ch.  
// All rights reserved.  
//  
// Copyright (c) 2001  
// Евгений А. Зуев,  
// ETH Zurich, Switzerland,  
// zueff@inf.ethz.ch.  
// Все права защищены.
```

<i>/////// Зарезервированные слова ///////////////</i>		
<code>%token lxmAsm</code>	<code>lxmFalse</code>	<code>lxmSizeof</code>
<code>%token lxmAuto</code>	<code>lxmFloat</code>	<code>lxmStatic</code>
<code>%token lxmBool</code>	<code>lxmFor</code>	<code>lxmStaticCast</code>
<code>%token lxmBreak</code>	<code>lxmFriend</code>	<code>lxmStruct</code>
<code>%token lxmCase</code>	<code>lxmGoto</code>	<code>lxmSwitch</code>
<code>%token lxmCatch</code>	<code>lxmIf</code>	<code>lxmTemplate</code>
<code>%token lxmChar</code>	<code>lxmInline</code>	<code>lxmThis</code>
<code>%token lxmClass</code>	<code>lxmInt</code>	<code>lxmThrow</code>
<code>%token lxmConst</code>	<code>lxmLong</code>	<code>lxmTrue</code>
<code>%token lxmConstCast</code>	<code>lxmMutable</code>	<code>lxmTry</code>
<code>%token lxmContinue</code>	<code>lxmNamespace</code>	<code>lxmTypedef</code>
<code>%token lxmDefault</code>	<code>lxmNew</code>	<code>lxmTypeid</code>
<code>%token lxmDelete</code>	<code>lxmOperator</code>	<code>lxmTypename</code>
<code>%token lxmDo</code>	<code>lxmPrivate</code>	<code>lxmUnion</code>
<code>%token lxmDouble</code>	<code>lxmProtected</code>	<code>lxmUnsigned</code>
<code>%token lxmDynamicCast</code>	<code>lxmPublic</code>	<code>lxmUsing</code>

```

%token lxmElse           lxmRegister          lxmVirtual
%token lxmEnum            lxmReinterpretCast   lxmVoid
%token lxmExplicit         lxmReturn           lxmVolatile
%token lxmExport           lxmShort            lxmWcharT
%token lxmExtern           lxmSigned           lxmWhile
// Литералы /////////////////////////////////
%token lxmCharacterLiteral
%token lxmFloatingLiteral
%token lxmIntegerLiteral
%token lxmStringLiteral
// Прочие лексемы ///////////////////////////////
%token lxmAmpersand        lxmAmpersandAmpersand lxmAmpersandEqual
%token lxmArrow             lxmArrowAsterisk    lxmAsterisk lxmAsteriskEqual
%token lxmCaret             lxmCaretEqual      lxmColon lxmColonColon lxmComma
%token lxmDot               lxmDotAsterisk     lxmDotDotDot lxmEqual
%token lxmEqualEqual       lxmExclamation    lxmGreater lxmGreaterEqual
%token lxmGreaterGreaterEqual lxmGreaterGreaterEqual
%token lxmLeftBrace         lxmLeftBracket   lxmLeftParenth
%token lxmLess              lxmLessEqual      lxmLessLess lxmLessLessEqual
%token lxmMinus             lxmMinusEqual     lxmMinusMinus lxmNonEqual
%token lxmPercent           lxmPercentEqual   lxmPlus lxmPlusEqual
%token lxmPlusPlus          lxmQuestion       lxmRightBrace lxmRightBracket
%token lxmRightParenth     lxmSemicolon     lxmSlash lxmSlashEqual
%token lxmTilde             lxmVertical      lxmVerticalEqual
%token lxmVerticalVertical
// Имена /////////////////////////////////
%token lxmIdentifier
%token lxmTypeName          // class-, enum- or typedef-name
%token lxmTemplateName       // template-name
%token lxmNamespaceName     // namespace-name
// Приоритеты лексем ///////////////////////////////
%left lxmEqual lxmAsteriskEqual lxmSlashEqual
      lxmPercentEqual lxmPlusEqual lxmMinusEqual
      lxmGreaterGreaterEqual lxmLessLessEqual lxmThrow
      lxmAmpersandEqual lxmCaretEqual lxmVerticalEqual
%left lxmQuestion
%left lxmVerticalVertical
%left lxmAmpersandAmpersand
%left lxmVertical
%left lxmCaret
%left lxmAmpersand
%left lxmEqualEqual lxmNonEqual
%left lxmLess lxmGreater lxmLessEqual lxmGreaterEqual
%left lxmLessLess lxmGreaterGreater
%left lxmPlus lxmMinus
%left lxmAsterisk lxmSlash lxmPercent
%left lxmDotAsterisk lxmArrowAsterisk lxmArrow lxmDot
%left priUnAsterisk priUnAmpersand priUnPlus priUnMinus
      lxmExclamation lxmTilde
%left lxmPlusPlus lxmMinusMinus
%left lxmLeftBracket lxmLeftParenth
%left lxmColonColon lxmColon lxmSemicolon lxmElse
%left lxmAuto lxmRegister lxmStatic lxmExtern lxmMutable

```

```

lxmInline lxmVirtual lxmExplicit lxmFriend
lxmTypedef lxmConst lxmVolatile lxmChar lxmWcharT
lxmBool lxmShort lxmInt lxmLong lxmSigned
lxmUnsigned lxmFloat lxmDouble lxmVoid lxmClass
lxmEnum lxmTemplate lxmIdentifier lxmTypeName
lxmNamespaceName lxmTemplateName lxmStruct
lxmTypename lxmUnion
%left lxmLeftBrace
%left priHighest
// Начальное правило /////////////////////////////////
%start TranslationUnit
%%
///////////////////////////////
// A.3 Basic concepts
TranslationUnit
: // empty
| DeclarationSeq
;
// A.4 Expressions
IdExpression
: UnqualifiedId
| QualifiedId
;
UnqualifiedId
: lxmIdentifier %prec lxmQuestion
| lxmOperator Operator
| lxmOperator ConversionTypeId %prec lxmQuestion
// | lxmTilde lxmIdentifier
// | lxmTilde TemplateId
;
QualifiedId
: QualifierNamed lxmTemplate UnqualifiedId
| QualifierNamed UnqualifiedId
| lxmColonColon lxmIdentifier
| lxmColonColon lxmOperator Operator
;
NestedNameSpecifier
: NestedNameTail
| NestedNameSpecifier NestedNameTail
| NestedNameSpecifier lxmTemplate NestedNameTailTemplate
;
NestedNameTail
: lxmNamespaceName lxmColonColon
| lxmTypeName lxmColonColon
| TemplateId lxmColonColon
;
NestedNameTailTemplate
: lxmTemplate lxmTemplateName
| lxmTemplate TemplateId
;
QualifierNamed // added
: lxmColonColon NestedNameSpecifier
| NestedNameSpecifier %prec lxmEqual
;

```

```

;
Qualifier // added
: QualifierNamed
| lxmColonColon
;
GeneralExpression
: lxmIntegerLiteral
| lxmCharacterLiteral
| lxmFloatingLiteral
| lxmStringLiteral
| lxmTrue // boolean-literal
| lxmFalse // boolean-literal
| lxmThis
| ParenthesizedExpression
| IdExpression
// postfix-expression
| GeneralExpression ArrayTail %prec lxmLeftBracket
| GeneralExpression CallOrConvTail %prec lxmLeftParenth
| SimpleTypeSpecifier CallOrConvTail %prec lxmLeftParenth
| lxmTypename QualifierNamed lxmTypeName CallOrConvTail
| lxmTypename QualifierNamed lxmTemplate
TemplateId CallOrConvTail
| lxmTypename QualifierNamed TemplateId CallOrConvTail
| GeneralExpression lxmDot lxmTemplate IdExpression
| GeneralExpression lxmDot IdExpression
| GeneralExpression lxmArrow lxmTemplate IdExpression
| GeneralExpression lxmArrow IdExpression
| GeneralExpression lxmDot PseudoDestructorName
| GeneralExpression lxmArrow PseudoDestructorName
| GeneralExpression lxmPlusPlus %prec lxmPlusPlus
| GeneralExpression lxmMinusMinus %prec lxmMinusMinus
CastOperator TargetType ParenthesizedExpression
lxmTypeid ParenthesizedExpression
lxmTypeid ParenthesizedTypeId
// unary-expression
| lxmPlusPlus GeneralExpression %prec lxmPlusPlus
| lxmMinusMinus GeneralExpression %prec lxmMinusMinus
| lxmAsterisk GeneralExpression %prec priUnAsterisk
| lxmAmpersand GeneralExpression %prec priUnAmpersand
| lxmPlus GeneralExpression %prec priUnPlus
| lxmMinus GeneralExpression %prec priUnMinus
| lxmExclamation GeneralExpression %prec lxmExclamation
| lxmTilde GeneralExpression %prec lxmTilde
| lxmSizeof GeneralExpression %prec priHighest
| lxmSizeof ParenthesizedTypeId %prec priHighest
NewExpression
DeleteExpression
// cast-expression
| ParenthesizedTypeId GeneralExpression %prec priHighest
// pm-expression
| GeneralExpression lxmDotAsterisk GeneralExpression
| GeneralExpression lxmArrowAsterisk GeneralExpression
// multiplicative-expression

```

```

| GeneralExpression lxmAsterisk           GeneralExpression
| GeneralExpression lxmSlash              GeneralExpression
| GeneralExpression lxmPercent           GeneralExpression
| // additive-expression
| GeneralExpression lxmPlus              GeneralExpression
| GeneralExpression lxmMinus             GeneralExpression
| // shift-expression
| GeneralExpression lxmLessLess         GeneralExpression
| GeneralExpression lxmGreaterGreater   GeneralExpression
| // relational-expression
| GeneralExpression lxmLess              GeneralExpression
| GeneralExpression lxmGreater           GeneralExpression
| GeneralExpression lxmLessEqual        GeneralExpression
| GeneralExpression lxmGreaterEqual     GeneralExpression
| // equality-expression
| GeneralExpression lxmEqualEqual       GeneralExpression
| GeneralExpression lxmNonEqual         GeneralExpression
| // and-expression
| GeneralExpression lxmAmpersand        GeneralExpression
| // exclusive-or-expression
| GeneralExpression lxmCaret             GeneralExpression
| // inclusive-or-expression
| GeneralExpression lxmVertical         GeneralExpression
| // logical-and-expression
| GeneralExpression lxmAmpersandAmpersand GeneralExpression
| // logical-or-expression
| GeneralExpression lxmVerticalVertical GeneralExpression
| // conditional-expression
| GeneralExpression lxmQuestion Expression
|           lxmColon    GeneralExpression
| // assignment-expression
| GeneralExpression lxmEqual            GeneralExpression
| GeneralExpression lxmAsteriskEqual    GeneralExpression
| GeneralExpression lxmSlashEqual      GeneralExpression
| GeneralExpression lxmPercentEqual    GeneralExpression
| GeneralExpression lxmPlusEqual       GeneralExpression
| GeneralExpression lxmMinusEqual      GeneralExpression
| GeneralExpression lxmGreaterGreaterEqual GeneralExpression
| GeneralExpression lxmLessLessEqual    GeneralExpression
| GeneralExpression lxmAmpersandEqual  GeneralExpression
| GeneralExpression lxmCaretEqual      GeneralExpression
| GeneralExpression lxmVerticalEqual   GeneralExpression
| // throw-expression
| lxmThrow  GeneralExpression
| lxmThrow
;
ParensizedExpression
: lxmLeftParenth Expression lxmRightParenth
;
ParensizedTypeId
: lxmLeftParenth TypeId lxmRightParenth
;

```

```

ArrayTail
  : lxmLeftBracket Expression lxmRightBracket
;
CallOrConvTail
  : lxmLeftParenth Expression lxmRightParenth
  | lxmLeftParenth           lxmRightParenth
;
PseudoDestructorName
  : Qualifier          SimpleDestructorName
//  |           SimpleDestructorName
//           .
//           -- is covered by
//           -- 'lxmTilde GeneralExpression'
//           -- from GeneralExpression.
| QualifierNamed lxmTemplate TemplateId lxmColonColon
  SimpleDestructorName
;
SimpleDestructorName
  : lxmTilde lxmTypeName ?
  | lxmTilde TemplateId
;
CastOperator
  : lxmDynamicCast
  | lxmStaticCast
  | lxmReinterpretCast
  | lxmConstCast
;
TargetType
  : lxmLess TypeId lxmGreater
;
NewExpression
  : New ParenthesizedExpression NewTypeId CallOrConvTail
  | New ParenthesizedExpression NewTypeId %prec lxmQuestion
  | New                           NewTypeId CallOrConvTail
  | New                           NewTypeId %prec lxmQuestion
;
New
  : lxmColonColon lxmNew
  |           lxmNew
;
NewTypeId
  : ParenthesizedTypeId
  |           TypeId // Instead of original NewTypeId
;
DeleteExpression
  : DeleteHeader GeneralExpression %prec priHighest
;
DeleteHeader
  : lxmColonColon lxmDelete
  |           lxmDelete
  | lxmColonColon lxmDelete lxmLeftBracket lxmRightBracket
  |           lxmDelete lxmLeftBracket lxmRightBracket
;
// Expression nonterminal INCLUDES LIST of expressions!

```

```

Expression
:
| Expression lxmComma GeneralExpression
;
// A.5 Statements
Statement
    // labeled-statement
:
| lxmIdentifier lxmColon Statement
| lxmCase GeneralExpression lxmColon Statement
| lxmDefault lxmColon Statement
    // expression-statement
| Expression lxmSemicolon
| CompoundStatement
    // selection-statement
| lxmIf Condition Statement           %prec lxmQuestion
| lxmIf Condition Statement lxmElse Statement
                                         %prec lxmElse
| lxmSwitch Condition Statement
    // iteration-statement
| lxmWhile Condition Statement
| lxmDo Statement lxmWhile ParenthesizedExpression
| lxmFor lxmLeftParenth ForHeader lxmRightParenth Statement
    // jump-statement
| lxmBreak lxmSemicolon
| lxmContinue lxmSemicolon
| lxmReturn Expression lxmSemicolon
| lxmReturn lxmSemicolon
| lxmGoto lxmIdentifier lxmSemicolon
| BlockDeclaration // declaration-statement
| TryBlock
                                         lxmSemicolon // empty-statement
;
CompoundStatement
:
| lxmLeftBrace StatementSeq lxmRightBrace
| lxmLeftBrace
                                         lxmRightBrace
;
StatementSeq
:
    Statement
| StatementSeq Statement
;
Condition
:
| lxmLeftParenth ForCondition lxmRightParenth
;
ForCondition
:
| Expression
| TypeSpecifierSeq Declarator lxmEqual GeneralExpression
;
ForHeader
:
| ForInitStatement ForCondition lxmSemicolon Expression
| ForInitStatement ForCondition lxmSemicolon
| ForInitStatement
                                         lxmSemicolon Expression
| ForInitStatement
                                         lxmSemicolon
;

```

```

ForInitStatement
  : Expression lxmSemicolon // expression-statement
  |           lxmSemicolon // expression-statement
  | SimpleDeclaration
  ;
// A.6 Declarations
DeclarationSeq
  : Declaration
  | DeclarationSeq Declaration
  ;
Declaration
  : BlockDeclaration
  | FunctionDefinition
  | TemplateDeclaration
    // explicit-instantiation
  | lxmTemplate Declaration
    // explicit-specialization
  | lxmTemplate lxmLess lxmGreater Declaration
    // linkage-specification
  | lxmExtern lxmStringLiteral BracedDeclarations
  | lxmExtern lxmStringLiteral Declaration
    // named-namespace-definition
  | lxmNamespace lxmIdentifier BracedDeclarations
  | lxmNamespace lxmNamespaceName BracedDeclarations
    // unnamed-namespace-definition
  | lxmNamespace BracedDeclarations
  ;
BracedDeclarations
  : lxmLeftBrace DeclarationSeq lxmRightParenth
  | lxmLeftBrace lxmRightParenth
  ;
BlockDeclaration
  : SimpleDeclaration
    // asm-definition
  | lxmAsm
    lxmLeftParenth lxmStringLiteral lxmRightParenth
    lxmSemicolon
    // namespace-alias-definition
  | lxmNamespace
    lxmIdentifier lxmEqual Qualifier
      lxmNamespaceName lxmSemicolon
  | lxmNamespace
    lxmIdentifier lxmEqual lxmNamespaceName lxmSemicolon
  | UsingDeclaration
  | UsingDirective
  ;
SimpleDeclaration
  : DeclSpecifierSeq InitDeclaratorList lxmSemicolon
  | DeclSpecifierSeq lxmSemicolon
  // | InitDeclaratorList lxmSemicolon
    // Standard, Section 7, $7:
    // "Only in function declarations for constructors,
    // destructors, and type conversions can the

```

```

// decl-specifier-seq be omitted. The 'implicit int'
// rule of C is no longer supported."
// So we excluded 'InitDeclaratorList lxmSemicolon'
// alternative from the rule and added the following
// alternatives which cover just constructors,
// destructors, and type conversions:
| DeclSpecifierSeq DirectAbstractDeclarator lxmSemicolon
| lxmOperator ConversionTypeId
| | DirectAbstractDeclarator lxmSemicolon
;
DeclSpecifier
    // storage-class-specifier
    : lxmAuto      %prec lxmAuto
    | lxmRegister  %prec lxmRegister
    | lxmStatic    %prec lxmStatic
    | lxmExtern    %prec lxmExtern
    | lxmMutable   %prec lxmMutable
    TypeSpecifier
        // function-specifier
        | lxmInline    %prec lxmInline
        | lxmVirtual   %prec lxmVirtual
        | lxmExplicit  %prec lxmExplicit
        // other specifiers
        | lxmFriend   %prec lxmFriend
        | lxmTypedef   %prec lxmTypedef
;
DeclSpecifierSeq
    :
    | DeclSpecifierSeq DeclSpecifier
;
TypeSpecifier
    : SimpleTypeSpecifier  %prec lxmQuestion
        // class-specifier
        | ClassKey          lxmIdentifier BaseClause ClassBody
    // | ClassKey          lxmTypeName     BaseClause ClassBody
    // | ClassKey          lxmIdentifier BaseClause ClassBody
    | ClassKey NestedNameSpecifier
        lxmTypeName     BaseClause ClassBody
    | ClassKey NestedNameSpecifier
        TemplateId      BaseClause ClassBody
    | ClassKey          TemplateId      BaseClause ClassBody
        // enum-specifier
    | lxmEnum lxmIdentifier EnumeratorBody
    | lxmEnum          EnumeratorBody
        // elaborated-type-specifier
    | ClassKey Qualifier      lxmTypeName
    | ClassKey Qualifier      lxmTypeName
    | lxmEnum  Qualifier      lxmTypeName
    | lxmEnum          lxmTypeName
    | lxmTypename QualifierNamed lxmTemplate TemplateId
    | lxmTypename QualifierNamed lxmTemplate TemplateId
        // cv-qualifier

```

```

| lxmConst      %prec lxmConst
| lxmVolatile   %prec lxmVolatile
;
SimpleTypeSpecifier
: TypeName
| lxmChar        %prec lxmChar
| lxmWcharT     %prec lxmWcharT
| lxmBool       %prec lxmBool
| lxmShort      %prec lxmShort
| lxmInt        %prec lxmInt
| lxmLong       %prec lxmLong
| lxmSigned     %prec lxmSigned
| lxmUnsigned   %prec lxmUnsigned
| lxmFloat      %prec lxmFloat
| lxmDouble     %prec lxmDouble
| lxmVoid       %prec lxmVoid
;
TypeName
: QualifierNamed lxmTypeName
| QualifierNamed TemplateId
| lxmColonColon lxmTypeName
| lxmColonColon TemplateId
|                   lxmTypeName
;
EnumeratorBody
: lxmLeftBrace EnumeratorList lxmRightBrace
| lxmLeftBrace           lxmRightBrace
;
EnumeratorList
:                           EnumeratorDefinition
| EnumeratorList lxmComma EnumeratorDefinition
;
EnumeratorDefinition
: lxmIdentifier
| lxmIdentifier lxmEqual GeneralExpression
;
UsingDeclaration
: lxmUsing lxmTypename
    QualifierNamed UnqualifiedId lxmSemicolon
| lxmUsing          QualifierNamed UnqualifiedId lxmSemicolon
| lxmUsing          lxmColonColon UnqualifiedId lxmSemicolon
;
UsingDirective
: lxmUsing lxmNamespace Qualifier
    lxmNamespaceName lxmSemicolon
| lxmUsing lxmNamespace          lxmNamespaceName lxmSemicolon
;
// A.7 Declarators
InitDeclaratorList
:                               InitDeclarator
| InitDeclaratorList lxmComma InitDeclarator
;
InitDeclarator

```

```

: Declarator
| Declarator Initializer
;
Declarator
: DirectDeclarator %prec lxmQuestion
| PtrOperator Declarator %prec priHighest
;
DirectDeclarator
: IdExpression // DeclaratorId
| DirectDeclarator ParameterDeclarationClause
| DirectDeclarator ArrayTail
| lxmLeftParenth Declarator lxmRightParenth
;
PtrOperator
: lxmAsterisk
| lxmAsterisk CvQualifierSeq
| lxmAmpersand
| QualifierNamed lxmAsterisk CvQualifierSeq
| QualifierNamed lxmAsterisk
;
DeclaratorTailOpt
: // empty
| CvQualifierSeq ExceptionSpecification
| CvQualifierSeq
| CvQualifierSeq ExceptionSpecification
;
CvQualifierSeq
: lxmConst lxmVolatile
| lxmConst
| lxmVolatile
;
TypeId
: TypeSpecifierSeq AbstractDeclarator %prec priHighest
| TypeSpecifierSeq %prec lxmQuestion
;
TypeSpecifierSeq
: TypeSpecifier
| TypeSpecifierSeq TypeSpecifier
;
AbstractDeclarator
: DirectAbstractDeclarator
| DirectAbstractDeclarator %prec lxmQuestion
| PtrOperator %prec lxmQuestion
| AbstractDeclarator PtrOperator .
;
DirectAbstractDeclarator
: DirectAbstractDeclarator
| DirectAbstractDeclarator ParameterDeclarationClause DeclaratorTailOpt
| DirectAbstractDeclarator ParameterDeclarationClause DeclaratorTailOpt
| DirectAbstractDeclarator BracketedExpression
| DirectAbstractDeclarator BracketedExpression
| lxmLeftParenth AbstractDeclarator lxmRightParenth
;
```

```

;
BracketedExpression
: lxmLeftBracket GeneralExpression lxmRightBracket
;
ParameterDeclarationClause
: lxmLeftParenth Parameters lxmRightParenth
;
Parameters
: // empty
| ParameterDeclarationList
| ParameterDeclarationList           lxmDotDotDot
|                               lxmDotDotDot
| ParameterDeclarationList lxmComma lxmDotDotDot
;
ParameterDeclarationList
:                                         ParameterDeclaration
| ParameterDeclarationList lxmComma ParameterDeclaration
;
ParameterDeclaration
: DeclSpecifierSeq Declarator
| DeclSpecifierSeq Declarator      lxmEqual GeneralExpression
| DeclSpecifierSeq
| DeclSpecifierSeq AbstractDeclarator
| DeclSpecifierSeq AbstractDeclarator
|                                         lxmEqual GeneralExpression
| DeclSpecifierSeq                  lxmEqual GeneralExpression
;
FunctionDefinition
: DeclSpecifierSeq Declarator FunctionBody
|             Declarator FunctionBody
;
FunctionBody
: CtorInitializer CompoundStatement
|                 CompoundStatement
|                 FunctionTryBlock
;
Initializer
: lxmEqual InitializerClause
// | lxmLeftParenth ExpressionList lxmRightParenth
| lxmLeftParenth Expression      lxmRightParenth
|                                         // instead of previous
;
InitializerClause
: GeneralExpression
| lxmLeftBrace InitializerList lxmComma lxmRightBrace
| lxmLeftBrace InitializerList           lxmRightBrace
| lxmLeftBrace                   lxmRightBrace
;
InitializerList
:                                         InitializerClause
| InitializerList lxmComma InitializerClause
;
// A.8 Classes

```

```

ClassKey
: lxmClass      %prec lxmClass
| lxmStruct     %prec lxmStruct
| lxmUnion      %prec lxmUnion
;
ClassBody    // added
: lxmLeftBrace MemberSpecification           lxmRightBrace
| lxmLeftBrace                           lxmRightBrace
| lxmLeftBrace           AccessSpecifiers lxmRightBrace
| lxmLeftBrace MemberSpecification           AccessSpecifiers lxmRightBrace
;
MemberSpecification
:                               MemberDeclaration
|           AccessSpecifiers MemberDeclaration
| MemberSpecification               MemberDeclaration
| MemberSpecification AccessSpecifiers MemberDeclaration
;
MemberDeclaration
: DeclSpecifierSeq MemberDeclaratorList lxmSemicolon
| DeclSpecifierSeq                 lxmSemicolon
|           MemberDeclaratorList lxmSemicolon
|           lxmSemicolon
| FunctionDefinition             lxmSemicolon
| FunctionDefinition   %prec lxmQuestion
// Access-modifier's syntax is covered by
// 'DeclSpecifierSeq lxmSemicolon' rule (see above).
// | QualifierNamed lxmTemplate UnqualifiedId lxmSemicolon
// | QualifierNamed           UnqualifiedId lxmSemicolon
| UsingDeclaration
| TemplateDeclaration
;
MemberDeclaratorList
:                               MemberDeclarator
| MemberDeclaratorList lxmComma MemberDeclarator
;
MemberDeclarator
// : Declarator PureSpecifierOpt
: Declarator
| Declarator lxmEqual GeneralExpression
| lxmIdentifier lxmColon GeneralExpression %prec lxmColon
|           lxmColon GeneralExpression
;
// A.9 Derived classes
BaseClause
: // empty
| lxmColon BaseSpecifierList
;
BaseSpecifierList
:                               BaseSpecifier
| BaseSpecifierList lxmComma BaseSpecifier
;
BaseSpecifier

```

```

: AccessToBase Qualifier lxmTypeName
| AccessToBase           lxmTypeName
|             Qualifier lxmTypeName
|                 lxmTypeName
;
AccessToBase
: lxmPrivate
| lxmProtected
| lxmPublic
| lxmPrivate   lxmVirtual
| lxmProtected lxmVirtual
| lxmPublic    lxmVirtual
| lxmVirtual
| lxmVirtual   lxmPrivate
| lxmVirtual   lxmProtected
| lxmVirtual   lxmPublic
;
AccessSpecifier
: lxmPrivate
| lxmProtected
| lxmPublic
;
AccessSpecifiers // added
:                         AccessSpecifier lxmColon
| AccessSpecifiers AccessSpecifier lxmColon
;
// A.10 Special member functions
ConversionTypeId
: TypeSpecifierSeq           %prec lxmQuestion
| TypeSpecifierSeq ConversionDeclarator %prec priHighest
;
ConversionDeclarator
:                         PtrOperator
| ConversionDeclarator PtrOperator
;
CtorInitializer
: lxmColon MemInitializerList
;
MemInitializerList
: MemInitializer
| MemInitializer lxmComma MemInitializerList
;
MemInitializer
: MemInitializerId CallOrConvTail
| MemInitializerId
;
MemInitializerId
: Qualifier lxmTypeName
|           lxmTypeName
// | lxmIdentifier -- is included into TypeName
;
// A.11 Overloading
Operator

```

```

: lxmNew      %prec lxmQuestion
| lxmDelete   %prec lxmQuestion
| lxmNew      lxmLeftBracket lxmRightBracket
| lxmDelete   lxmLeftBracket lxmRightBracket
| lxmPlus
| lxmMinus
| lxmAsterisk
| lxmSlash
| lxmPercent
| lxmCaret
| lxmAmpersand
| lxmVertical
| lxmTilde
| lxmExclamation
| lxmEqual
| lxmLess
| lxmGreater
| lxmPlusEqual
| lxmMinusEqual
| lxmAsteriskEqual
| lxmSlashEqual
| lxmPercentEqual
| lxmCaretEqual
| lxmAmpersandEqual
| lxmVerticalEqual
| lxmLessLess
| lxmGreaterGreater
| lxmGreaterGreaterEqual
| lxmLessLessEqual
| lxmEqualEqual
| lxmNonEqual
| lxmLessEqual
| lxmGreaterEqual
| lxmAmpersandAmpersand
| lxmVerticalVertical
| lxmPlusPlus
| lxmMinusMinus
| lxmArrowAsterisk
| lxmArrow
| lxmLeftParenth lxmRightParenth
| lxmLeftBracket lxmRightBracket
;
// A.12 Templates
TemplateDeclaration
: lxmExport TemplateHeader Declaration
|           TemplateHeader Declaration
;
TemplateHeader
: lxmTemplate lxmLess TemplateParameterList lxmGreater
;
TemplateParameterList
:                               TemplateParameter
| TemplateParameterList lxmComma TemplateParameter

```

```

;
TemplateParameter
: TypeParameter
| ParameterDeclaration
;
TypeParameter
: lxmClass
| lxmClass lxmIdentifier
| lxmClass lxmIdentifier lxmEqual TypeId
| lxmClass lxmEqual TypeId
| lxmTypename
| lxmTypename lxmIdentifier
| lxmTypename lxmIdentifier lxmEqual TypeId
| lxmTypename lxmEqual TypeId
| TemplateHeader lxmClass
| TemplateHeader lxmClass lxmIdentifier
;
TemplateId
: lxmTemplateName lxmLess TemplateArgumentList lxmGreater
| lxmTemplateName lxmLess lxmGreater
;
TemplateArgumentList
: TemplateArgument
| TemplateArgumentList lxmComma TemplateArgument
;
TemplateArgument
: GeneralExpression %prec lxmQuestion
| TypeId
// | IdExpression -- covered by GeneralExpression
;
// A.13 Exception handling
TryBlock
: lxmTry CompoundStatement HandlerSeq
;
FunctionTryBlock
: lxmTry CtorInitializer CompoundStatement HandlerSeq
| lxmTry CompoundStatement HandlerSeq
;
HandlerSeq
: Handler
| HandlerSeq Handler
;
Handler
: lxmCatch
| lxmLeftParenth ExceptionDeclaration lxmRightParenth
| CompoundStatement
;
ExceptionDeclaration
: TypeSpecifierSeq Declarator
| TypeSpecifierSeq AbstractDeclarator
;
```

```

| TypeSpecifierSeq
| lxmDotDotDot
;
ExceptionSpecification
: lxmThrow lxmLeftParenth TypeIdList lxmRightParenth
| lxmThrow lxmLeftParenth           lxmRightParenth
;
TypeIdList
:
    TypeId
| TypeIdList lxmComma TypeId
;
%%
////////// Синтаксис языка C# //////////

```

Б.2. Язык программирования C#

Источник: C# Language Specification, Draft 01. ECMA/TC39/TG2/2000/3 Draft Standard ECMA-xxxx. — October 2000.

C# — язык программирования, предложенный компанией Microsoft как базовый язык для новой программной платформы Microsoft .NET.

C# — полнофункциональный язык общего назначения, дизайн которого аналогичен дизайну языка Java. Модели выполнения C#- и Java-программ также имеют много общего (в частности, отсутствие указателей, автоматическая сборка мусора, тесная интеграция со стандартным библиотечным окружением).

Вместе с тем C# допускает ряд низкоуровневых возможностей, характерных для языков C/C++. Соответствующие конструкции явно оформляются в виде отдельных “небезопасных” (unsafe) фрагментов программ.

Объектная модель C# практически идентична объектной модели Java: единичное наследование для классов с общим базовым классом, концепция интерфейса с возможностью множественных реализаций интерфейсов.

Каждое синтаксическое правило в данной грамматике сопровождается ссылкой на раздел спецификации языка, в котором содержится описание соответствующей языковой конструкции.

```

////////// Синтаксис языка C# //////////
// Reference:
// C# Language Specification, Draft 01.
// ECMA/TC39/TG2/2000/3 Draft Standard ECMA-xxxx.-.
// October 2000.
// Copyright (c) 2001
// Eugene A. Zueff
// ETH Zurich, Switzerland
// zueff@inf.ethz.ch.
// All rights reserved.
//
// Copyright (c) 2001
// Евгений А. Зуев,
// ETH Zurich, Switzerland,
// zueff@inf.ethz.ch.
// Все права защищены.

```

```

/////// Зарезервированные слова /////////////////
%token lxmAbstract           // abstract
%token lxmAs                 // as
%token lxmBase               // base
%token lxmBool               // bool
%token lxmBreak              // break
%token lxmByte               // byte
%token lxmCase               // case
%token lxmCatch              // catch
%token lxmChar               // char
%token lxmChecked            // checked
%token lxmClass              // class
%token lxmConst               // const
%token lxmContinue            // continue
%token lxmDecimal             // decimal
%token lxmDefault             // default
%token lxmDelegate            // delegate
%token lxmDo                  // do
%token lxmDouble              // double
%token lxmElse                // else
%token lxmEnum                // enum
%token lxmEvent               // event
%token lxmExplicit             // explicit
%token lxmExtern              // extern
%token lxmFalse               // false
%token lxmFinally             // finally
%token lxmFixed               // fixed
%token lxmFloat               // float
%token lxmFor                 // for
%token lxmForeach             // foreach
%token lxmGoto                // goto
%token lxmIf                  // if
%token lxmImplicit             // implicit
%token lxmIn                  // in
%token lxmInt                 // int
%token lxmInterface            // interface
%token lxmInternal             // internal
%token lxmIs                  // is
%token lxmLock                // lock
%token lxmLong                // long
%token lxmNamespace            // namespace
%token lxmNew                 // new
%token lxmObject               // object
%token lxmOperator              // operator
%token lxmOut                 // out
%token lxmOverride              // override
%token lxmParams               // params
%token lxmPrivate              // private
%token lxmProtected             // protected
%token lxmPublic               // public
%token lxmReadOnly              // readonly

```

```

%token lxmRef           // ref
%token lxmReturn        // return
%token lxmSbyte         // sbyte
%token lxmSealed        // sealed
%token lxmShort         // short
%token lxmSizeof        // sizeof
%token lxmStackalloc    // stackalloc
%token lxmStatic         // static
%token lxmString         // string
%token lxmStruct         // struct
%token lxmSwitch         // switch
%token lxmThis           // this
%token lxmThrow          // throw
%token lxmTrue           // true
%token lxmTry            // try
%token lxmTypeof         // typeof
%token lxmUint           // uint
%token lxmUlong          // ulong
%token lxmUnchecked      // unchecked
%token lxmUnsafe         // unsafe
%token lxmUshort          // ushort
%token lxmUsing          // using
%token lxmVirtual        // virtual
%token lxmVoid           // void
%token lxmWhile          // while

/////// Псевдо зарезервированные слова ///////////////
%token lxmGet           // get
%token lxmSet           // set
%token lxmAdd           // add
%token lxmRemove         // remove

/////// Литералы /////////////////////////////////
%token lxmCharLiteral    // 'c'
%token lxmDecimalLiteral // 12.345M
%token lxmDoubleLiteral  // 12.345
%token lxmFloatLiteral   // 12.345F
%token lxmIntLiteral     // 12345
%token lxmLongLiteral    // 12345L
%token lxmStringLiteral  // "abcd"
%token lxmUintLiteral    // 12345U
%token lxmUlongLiteral   // 12345UL

/////// Прочие токены ///////////////////////////////
%token lxmAmpersand      // &
%token lxmAmpersandAmpersand // &&
%token lxmAmpersandEqual  // &=
%token lxmAsterisk        // *
%token lxmAsteriskEqual   // *=
%token lxmCaret           // ^
%token lxmCaretEqual      // ^=
%token lxmColon           // :
%token lxmComma           // ,
%token lxmDot             // .

```

```

%token lxmEqual           // =
%token lxmEqualEqual      // ==
%token lxmExclamation     // !
%token lxmGreater          // >
%token lxmGreaterEqual     // >=
%token lxmGreaterGreater   // >>
%token lxmGreaterGreaterEqual // >>=
%token lxmIdentifier
%token lxmLeftBrace        // {
%token lxmLeftBracket       // [
%token lxmLeftParenth      // (
%token lxmLess              // <
%token lxmLessEqual         // <=
%token lxmLessLess          // <<
%token lxmLessLessEqual     // <<=
%token lxmMinus             // -
%token lxmMinusEqual        // --=
%token lxmMinusGreater      // ->
%token lxmMinusMinus        // ---
%token lxmNonEqual          // !==
%token lxmPercent           // %
%token lxmPercentEqual      // %=
%token lxmPlus              // +
%token lxmPlusEqual         // +==
%token lxmPlusPlus          // ++
%token lxmQuestion          // ?
%token lxmRankSpecifier      // [ ] [,] [,,] ...
%token lxmRightBrace        // }
%token lxmRightBracket       // ]
%token lxmRightParenth      // )
%token lxmSemicolon          // ;
%token lxmSlash              // /
%token lxmSlashEqual         // /=
%token lxmTilde              // ~
%token lxmVertical           // |
%token lxmVerticalEqual      // |=
%token lxmVerticalVertical    // ||

////// Приоритеты операторов ///////////////
%left lxmQuestion           // ?
lxmColon                  // :
lxmEqual                   // =
lxmPlusEqual               // +==
lxmMinusEqual              // --=
lxmAsteriskEqual          // *=
lxmSlashEqual              // /==
lxmPercentEqual            // %=
lxmAmpersandEqual          // &=
lxmVerticalEqual           // |=
lxmCaretEqual              // ^=
lxmLessLessEqual            // <<=
lxmGreaterGreaterEqual     // >>=

```

```

%left lxmVerticalVertical      //    ||
%left lxmAmpersandAmpersand   //    &&
%left lxmVertical             //    |
%left lxmCaret                //    ^
%left lxmAmpersand            // binary &
%left lxmEqualEqual           //    ==
%left lxmNonEqual              //    !=
%left lxmLess                 //    <
%left lxmGreater               //    >
%left lxmLessEqual             //    <=
%left lxmGreaterEqual          //    >=
%left lxmIs                   //    is
%left lxmAs                   //    as
%left lxmLessLess              //    <<
%left lxmGreaterGreater        //    >>
%left lxmPlus                 // binary +
%left lxmMinus                 // binary -
%left lxmAsterisk              // binary *
%left lxmSlash                 //    /
%left lxmPercent               //    %
%right priUnaryMinus           // unary -
%right priUnaryPlus            // unary +
%left lxmExclamation           //    !
%left lxmTilde                 //    ~
%right priDereference          // unary *
%right priAddress               // unary &
%left lxmPlusPlus              //    ++
%left lxmMinusMinus             //    --
%left lxmMinusGreater           //    ->
%left lxmLeftBracket            //    [
%left lxmLeftParenth            //    (
%left lxmRightParenth           //    )
%left lxmBool
%left lxmByte
%left lxmChar
%left lxmDecimal
%left lxmDouble
%left lxmFloat
%left lxmIdentifier
%left lxmInt
%left lxmLong
%left lxmObject
%left lxmSbyte
%left lxmShort
%left lxmString
%left lxmUint
%left lxmUlong
%left lxmUshort
%left lxmVoid
%left lxmElse
%left lxmDot                  //



//////////////// Стартовая продукция ///////////////////

```

```

%start CompilationUnit
/////////// Продукции /////////////////
%%
///////// 9.1 Compilation Units ///////////////
CompilationUnit
    : NamespaceInternals
    ;
NamespaceInternals
    : // empty
    | UsingDirectives NamespaceMemberDeclarations
    | UsingDirectives
    | NamespaceMemberDeclarations
    ;
///////// 9.2 Namespace declarations ///////////////
NamespaceDeclaration
    : lxmNamespace QualifiedIdentifier lxmLeftBrace
      NamespaceInternals RightBraceSemicolon
    ;
QualifiedIdentifier
    : lxmIdentifier
    | QualifiedIdentifier lxmDot lxmIdentifier
      %prec lxmDot
    ;
///////// 9.3 Using directives ///////////////
UsingDirectives
    : UsingDirective
    | UsingDirectives UsingDirective
    ;
UsingDirective
    : // 9.3.1 Using alias directives
      lxmUsing lxmIdentifier lxmEqual
      QualifiedIdentifier lxmSemicolon
      // 9.3.2 Using namespace directives
    | lxmUsing QualifiedIdentifier lxmSemicolon
    ;
///////// 9.4 Namespace members ///////////////
NamespaceMemberDeclarations
    : NamespaceMemberDeclaration
    | NamespaceMemberDeclarations
      NamespaceMemberDeclaration
    ;
NamespaceMemberDeclaration
    : NamespaceDeclaration
    | TypeDeclaration
    ;
///////// 9.5 Type declarations ///////////////
TypeDeclaration
    : // 10.1 Class declarations
      AttributesModifiers lxmClass lxmIdentifier
      BasesOpt ClassBody
    | lxmClass lxmIdentifier BasesOpt ClassBody
      // 11.1 Struct declarations
    | AttributesModifiers lxmStruct lxmIdentifier

```

```

    BasesOpt ClassBody
    | lxmStruct lxmIdentifier BasesOpt ClassBody
    // 13.1 Interface declarations
    | AttributesModifiers lxmInterface lxmIdentifier
    BasesOpt InterfaceBody
    | lxmInterface lxmIdentifier BasesOpt
    InterfaceBody
    // 14.1 Enum declarations
    | AttributesModifiers lxmEnum lxmIdentifier
    EnumBaseOpt EnumBody
    | lxmEnum lxmIdentifier EnumBaseOpt EnumBody
    // 15.1 Delegate declarations
    | AttributesModifiers lxmDelegate ReturnType
    lxmIdentifier FormalParameterPart lxmSemicolon
    | lxmDelegate ReturnType lxmIdentifier
    FormalParameterPart lxmSemicolon
    ;
AttributesModifiers
    : Attributes Modifiers
    | Attributes
    | Modifiers
    ;
ReturnType
    : Type
    | lxmVoid
    ;
////////// 10.1.2 Class base specification ///////
BasesOpt
    : // empty
    | lxmColon TypeNames
    ;
TypeNames
    : // one base class + several interfaces
    | QualifiedIdentifier
    | TypeNames lxmComma QualifiedIdentifier
    ;
////////// 10.1.3 Class body /////////////
ClassBody
    : lxmLeftBrace RightBraceSemicolon
    | lxmLeftBrace ClassMemberDeclarations
    RightBraceSemicolon
    ;
RightBraceSemicolon
    : lxmRightBrace
    | lxmRightBrace lxmSemicolon
    ;
////////// 10.2 Class members /////////////
ClassMemberDeclarations
    : ClassMemberDeclaration
    | ClassMemberDeclarations ClassMemberDeclaration
    ;
ClassMemberDeclaration
    // 10.3 Constants
    // 10.4 Fields

```

```

: ConstantFieldDeclaration
// 10.5 Methods
| AttributesModifiers ReturnType
QualifiedIdentifier FormalParameterPart Body
| ReturnType QualifiedIdentifier
FormalParameterPart Body
// 10.6 Properties
| AttributesModifiers Type QualifiedIdentifier
AccessorDeclarationPart
| Type QualifiedIdentifier
AccessorDeclarationPart
// 10.7 Events
| AttributesModifiers lxmEvent Type Declarators
lxmSemicolon
| lxmEvent Type Declarators lxmSemicolon
AttributesModifiers lxmEvent Type
QualifiedIdentifier EventAccessorDeclarations
| lxmEvent Type QualifiedIdentifier
EventAccessorDeclarations
// 10.8 Indexers
| AttributesModifiers Type Indexer
IndexerParameterPart AccessorDeclarationPart
| Type Indexer IndexerParameterPart
AccessorDeclarationPart
// 10.9 Operators
| AttributesModifiers OperatorDeclarator
Block // Modifiers: should be 'public static'
| OperatorDeclarator Block
// 10.10 Instance constructors
// 10.11 Static constructors
| AttributesModifiers ConstructorDeclarator Block
| ConstructorDeclarator Block
// 10.12 Destructors
| AttributesOpt lxmTilde lxmIdentifier
lxmLeftParenth lxmRightParenth Block
| TypeDeclaration
;

Body
: Block
| lxmSemicolon
;
///////// 10.3 Constants /////////////
///////// 10.4 Fields ///////////
ConstantFieldDeclaration
: AttributesModifiers Type Declarators
lxmSemicolon
| Type Declarators lxmSemicolon
;
Declarators
: Declarator
| Declarators lxmComma Declarator
;
Declarator
;
```

```

        : lxmIdentifier
        | lxmIdentifier lxmEqual VariableInitializer
;
VariableInitializer
        : GeneralExpression
        | ArrayInitializer
        | StackallocInitializer //only in unsafe context
;
StackallocInitializer // A.7 Stack allocation
        : lxmStackalloc Type lxmLeftBracket
          GeneralExpression lxmRightBracket
          // unmanaged-type [ expression ]
;
//////////////// 10.5.1 Method parameters /////////////////////
FormalParameterPart
        : lxmLeftParenth FormalParameterList
          lxmRightParenth
        | lxmLeftParenth lxmRightParenth
;
FormalParameterList
        : FormalParameter
        | FormalParameterList lxmComma FormalParameter
;
FormalParameter
        : AttributesOpt ParameterModifierOpt Type
          lxmIdentifier
;
ParameterModifierOpt
        : // empty
        | lxmRef
        | lxmOut
        | lxmParams
;
//////////////// 10.6.2 Accessors /////////////////////
AccessorDeclarationPart
        : lxmLeftBrace AccessorDeclarations lxmRightBrace
;
AccessorDeclarations
        : GetAccessorDeclaration
        | SetAccessorDeclaration
        | GetAccessorDeclaration SetAccessorDeclaration
        | SetAccessorDeclaration GetAccessorDeclaration
;
GetAccessorDeclaration
        : AttributesOpt lxmGet Body
;
SetAccessorDeclaration
        : AttributesOpt lxmSet Body
;
//////////////// 10.7 Events /////////////////////
EventAccessorDeclarations
        : lxmLeftBrace AddAccessorDeclaration
          RemoveAccessorDeclaration lxmRightBrace
;

```

```

    | lxmLeftBrace RemoveAccessorDeclaration
    | AddAccessorDeclaration lxmRightBrace
    ;
AddAccessorDeclaration
    : AttributesOpt lxmAdd Block
    ;
RemoveAccessorDeclaration
    : AttributesOpt lxmRemove Block
    ;
//////// 10.8 Indexers /////////////
IndexerParameterPart
    : lxmLeftBracket FormalParameterList
    | lxmRightBracket
    ;
Indexer
    : QualifiedIdentifier lxmDot lxmThis
    | lxmThis
    ;
//////// 10.9 Operators ///////////
OperatorDeclarator
    // unary-operator-declarator
    : Type lxmOperator OverloadableOperator
    UnaryOperatorParam
    // binary-operator-declarator
    | Type lxmOperator OverloadableOperator
    BinaryOperatorParams
    // conversion-operator-declarator
    | lxmImplicit lxmOperator Type UnaryOperatorParam
    | lxmExplicit lxmOperator Type UnaryOperatorParam
    ;
UnaryOperatorParam
    : lxmLeftParenth Type lxmIdentifier
    | lxmRightParenth
    ;
BinaryOperatorParams
    : lxmLeftParenth Type lxmIdentifier lxmComma
    Type lxmIdentifier lxmRightParenth
    ;
OverloadableOperator // overloadable-unary-operator OR
                    // overloadable-binary-operator
    : lxmPlus           // +   UNARY or BINARY
    | lxmMinus          // -   UNARY or BINARY
    | lxmExclamation   // !   UNARY
    | lxmTilde          // ~   UNARY
    | lxmPlusPlus      // ++  UNARY
    | lxmMinusMinus    // --  UNARY
    | lxmTrue           // true UNARY
    | lxmFalse          // false UNARY
    | lxmAsterisk       // *   BINARY
    | lxmSlash          // /   BINARY
    | lxmPercent        // %   BINARY
    | lxmAmpersand     // &   BINARY

```

```

| lxmVertical           // | BINARY
| lxmCaret              // ^ BINARY
| lxmLessLess            // << BINARY
| lxmGreaterGreater     // >> / BINARY
| lxmEqualEqual          // == / BINARY
| lxmNonEqual            // != / BINARY
| lxmGreater             // > / BINARY
| lxmLess                // < / BINARY
| lxmGreaterEqual        // >= BINARY
| lxmLessEqual            // <= BINARY
;

////////// 10.10 Instance constructors ///////////
ConstructorDeclarator
: lxmIdentifier FormalParameterPart lxmColon
  lxmBase ArgumentPart
| lxmIdentifier FormalParameterPart
  lxmColon lxmThis ArgumentPart
| lxmIdentifier FormalParameterPart
;

////////// 13.1.3 Interface body ///////////
InterfaceBody
: lxmLeftBrace lxmRightBrace
| lxmLeftBrace InterfaceMemberDeclarations
  lxmRightBrace
;

////////// 13.2 Interface members ///////////
InterfaceMemberDeclarations
: InterfaceMemberDeclaration
| InterfaceMemberDeclarations
  InterfaceMemberDeclaration
;

InterfaceMemberDeclaration
// 13.2.1 Interface methods
: InterfaceMemberPrefix Type lxmIdentifier
  FormalParameterPart lxmSemicolon
// 13.2.2 Interface properties
| InterfaceMemberPrefix Type lxmIdentifier
  InterfaceAccessorPart
// 13.2.3 Interface events
| InterfaceMemberPrefix lxmEvent Type
  lxmIdentifier lxmSemicolon
// 13.2.4 Interface indexers
| InterfaceMemberPrefix Type lxmThis
  IndexerParameterPart InterfaceAccessorPart
;

InterfaceMemberPrefix
: AttributesOpt lxmNew
| AttributesOpt
;

////////// 13.2.2 Interface properties ///////////
InterfaceAccessorPart
: lxmLeftBrace InterfaceAccessors lxmRightBrace
;

```

```

InterfaceAccessors
    : AttributesOpt lxmGet lxmSemicolon
    | AttributesOpt lxmSet lxmSemicolon
    | AttributesOpt lxmGet lxmSemicolon
        AttributesOpt lxmSet lxmSemicolon
    | AttributesOpt lxmSet lxmSemicolon
        AttributesOpt lxmGet lxmSemicolon
    ;
////////// 14.1 Enum declarations ///////////
EnumBaseOpt
    : // empty
    | lxmColon SimpleType
    ;
EnumBody
    : lxmLeftBrace RightBraceSemicolon
    | lxmLeftBrace EnumMemberDeclarations
        RightBraceSemicolon
    | lxmLeftBrace EnumMemberDeclarations
        lxmComma RightBraceSemicolon
    ;
////////// 14.3 Enum members ///////////
EnumMemberDeclarations
    : EnumMemberDeclaration
    | EnumMemberDeclarations lxmComma
        EnumMemberDeclaration
    ;
EnumMemberDeclaration
    : AttributesOpt lxmIdentifier lxmEqual
        GeneralExpression
    | AttributesOpt lxmIdentifier
    ;
////////// 4. Types ///////////
Type
    : NonArrayType %prec lxmQuestion
    | NonArrayType RankSpecifiers
        %prec lxmLeftBracket
    ;
NonArrayType
    : SimpleType %prec lxmQuestion
    | ClassType
    | QualifiedIdentifier %prec lxmQuestion
    | SimpleType lxmAsterisk %prec lxmDot
    | QualifiedIdentifier lxmAsterisk %prec lxmDot
    | lxmVoid lxmAsterisk
    ;
SimpleType
    : lxmSbyte          // Integral types
    | lxmByte
    | lxmShort
    | lxmUshort
    | lxmInt
    | lxmUint
    | lxmLong

```

```

| lxmUlong
| lxmChar
| lxmFloat      // Floating-point types
| lxmDouble
| lxmDecimal
| lxmBool
;
ClassType
: lxmObject
| lxmString
;
///////// 12.1 Array types ///////////
RankSpecifiers
: lxmRankSpecifier
| RankSpecifiers lxmRankSpecifier
;
///////// 12.6 Array initializers ///////////
ArrayInitializerOpt
: // empty
| ArrayInitializer
;
ArrayInitializer
: lxmLeftBrace lxmRightBrace
| lxmLeftBrace VariableInitializerList
  lxmRightBrace
| lxmLeftBrace VariableInitializerList
  lxmComma lxmRightBrace
;
VariableInitializerList
: VariableInitializer
| VariableInitializerList lxmComma
  VariableInitializer
;
///////// 7.4.1 Argument lists ///////////
ArgumentPart
: lxmLeftParenth           lxmRightParenth
| lxmLeftParenth ArgumentList lxmRightParenth
;
ArgumentList
: Argument
| ArgumentList lxmComma Argument
;
Argument
: GeneralExpression
| lxmRef GeneralExpression
| lxmOut GeneralExpression
;
///////// 7. Expressions ///////////
GeneralExpressionOpt
: // empty
| GeneralExpression
;
GeneralExpression

```

```

//          lxmIdentifier
:          lxmFloatLiteral           // 7.5.1 Literals
|          lxmDoubleLiteral
|          lxmDecimalLiteral
|          lxmUlongLiteral
|          lxmUintLiteral
|          lxmLongLiteral
|          lxmIntLiteral
|          lxmStringLiteral
|          lxmCharLiteral
|          lxmBase
|          lxmThis                // 7.5.7 This access
|          QualifiedIdentifier    %prec lxmQuestion
//
|          Type
|          lxmLeftParenth GeneralExpression
|          lxmRightParenth
// 7.5.3 Parenthesized expressions
// 7.6.8 Cast expressions
|          lxmLeftParenth GeneralExpression
|          lxmRightParenth GeneralExpression
|          lxmLeftParenth GeneralExpression
|          lxmAsterisk lxmRightParenth GeneralExpression
|          lxmLeftParenth TargetType lxmRightParenth
|          GeneralExpression
// 7.5.4 Member access
|          GeneralExpression lxmDot lxmIdentifier
|          SimpleType         lxmDot lxmIdentifier
|          ClassType          lxmDot lxmIdentifier
// 7.5.5 Invocation expressions
|          InvocationExpression
// 7.5.6 Element access
|          GeneralExpression lxmLeftBracket
|          ExpressionList lxmRightBracket
// 7.5.9 Postfix increment and decrement
|          IncDecExpression
|          NewExpression      // 7.5.10 new operator
// 7.5.11 typeof operator
|          lxmTypeof lxmLeftParenth Type lxmRightParenth
// 7.5.8 sizeof operator
|          lxmSizeof lxmLeftParenth Type lxmRightParenth
// 7.5.12 checked and unchecked operators
|          lxmChecked lxmLeftParenth GeneralExpression
|          lxmRightParenth
|          lxmUnchecked lxmLeftParenth GeneralExpression
|          lxmRightParenth
// 7.5.2 Pointer member access
|          GeneralExpression lxmMinusGreater
|          lxmIdentifier %prec lxmMinusGreater //expr->id
// 7.6 Unary expressions
|          lxmPlus GeneralExpression
|          %prec priUnaryPlus     // + expr
|          lxmMinus GeneralExpression
|          %prec priUnaryMinus   // - expr

```

```

| lxmExclamation GeneralExpression
| %prec lxmExclamation // ! expr
| lxmTilde GeneralExpression
| %prec lxmTilde // ~ expr
| lxmAsterisk GeneralExpression
| %prec priDereference // * expr
| lxmAmpersand GeneralExpression
| %prec priAddress // & expr
// 7.7 Arithmetic operators
| GeneralExpression lxmAsterisk GeneralExpression
| %prec lxmAsterisk // Multiplication
| GeneralExpression lxmSlash GeneralExpression
| %prec lxmSlash
| GeneralExpression lxmPercent GeneralExpression
| %prec lxmPercent
| GeneralExpression lxmPlus GeneralExpression
| %prec lxmPlus // Addition
| GeneralExpression lxmMinus GeneralExpression
| %prec lxmMinus
// 7.8 Shift operators
| GeneralExpression lxmLessLess GeneralExpression
| %prec lxmLessLess
| GeneralExpression lxmGreaterGreater
| GeneralExpression %prec lxmGreaterGreater
// 7.9 Relational operators
| GeneralExpression lxmLess GeneralExpression
| %prec lxmLess // Relational expression
| GeneralExpression lxmGreater GeneralExpression
| %prec lxmGreater
| GeneralExpression lxmLessEqual
| GeneralExpression %prec lxmLessEqual
| GeneralExpression lxmGreaterEqual
| GeneralExpression %prec lxmGreaterEqual
| GeneralExpression lxmIs Type %prec lxmIs
| GeneralExpression lxmAs Type %prec lxmAs
| GeneralExpression lxmEqualEqual
| GeneralExpression %prec lxmEqualEqual
| GeneralExpression lxmNonEqual
| GeneralExpression %prec lxmNonEqual
// 7.10 Logical operators
| GeneralExpression lxmAmpersand
| GeneralExpression %prec lxmAmpersand // And
| GeneralExpression lxmCaret GeneralExpression
| %prec lxmCaret // Exclusive-or expression
| GeneralExpression lxmVertical GeneralExpression
| %prec lxmVertical // Inclusive-or expression
// 7.11 Conditional logical operators
| GeneralExpression lxmAmpersandAmpersand
| GeneralExpression %prec lxmAmpersandAmpersand
| GeneralExpression lxmVerticalVertical
| GeneralExpression %prec lxmVerticalVertical
// 7.12 Conditional operator
| GeneralExpression lxmQuestion GeneralExpression

```

```

    lxmColon GeneralExpression
    // 7.13 Assignment operators
    | Assignment
    ;
TargetType
    : SimpleType %prec lxmQuestion
    | ClassType
    // | QualifiedIdentifier %prec lxmQuestion
    | SimpleType lxmAsterisk %prec lxmDot
    // | QualifiedIdentifier lxmAsterisk %prec lxmDot
    | lxmVoid lxmAsterisk
    // | NonArrayType RankSpecifiers
    // | %prec lxmLeftBracket
    ;
Assignment
    : GeneralExpression lxmEqual
    | GeneralExpression %prec lxmEqual
    | GeneralExpression lxmPlusEqual
    | GeneralExpression %prec lxmPlusEqual
    | GeneralExpression lxmMinusEqual
    | GeneralExpression %prec lxmMinusEqual
    | GeneralExpression lxmAsteriskEqual
    | GeneralExpression %prec lxmAsteriskEqual
    | GeneralExpression lxmSlashEqual
    | GeneralExpression %prec lxmSlashEqual
    | GeneralExpression lxmPercentEqual
    | GeneralExpression %prec lxmPercentEqual
    | GeneralExpression lxmAmpersandEqual
    | GeneralExpression %prec lxmAmpersandEqual
    | GeneralExpression lxmVerticalEqual
    | GeneralExpression %prec lxmVerticalEqual
    | GeneralExpression lxmCaretEqual
    | GeneralExpression %prec lxmCaretEqual
    | GeneralExpression lxmLessLessEqual
    | GeneralExpression %prec lxmLessLessEqual
    | GeneralExpression lxmGreaterGreaterEqual
    | GeneralExpression %prec lxmGreaterGreaterEqual
    ;
InvocationExpression
    : GeneralExpression ArgumentPart
    ;
IncDecExpression
    : GeneralExpression lxmPlusPlus
    | GeneralExpression lxmMinusMinus
    // 7.6.7 Prefix increment and decrement
    | lxmPlusPlus GeneralExpression
    | %prec lxmPlusPlus
    | lxmMinusMinus GeneralExpression
    | %prec lxmMinusMinus
    ;
ExpressionList // 7.14 Expressions
    : GeneralExpression
    | ExpressionList lxmComma GeneralExpression

```

```

;
////////// 7.5.10 new operator /////////////
NewExpression
    // 7.5.10.1 Object creation expressions
    // 7.5.10.3 Delegate creation expressions
: lxmNew Type ArgumentPart
    // 7.5.10.2 Array creation expressions
| lxmNew NonArrayType NewArrayClause
  RankSpecifiers ArrayInitializerOpt
| lxmNew NonArrayType NewArrayClause
  ArrayInitializerOpt
| lxmNew Type ArrayInitializer
;
NewArrayClause
: lxmLeftBracket ExpressionList lxmRightBracket
;
////////// 8. Statements /////////////
Statement
: Labels EmbeddedStatement
| EmbeddedStatement
| ConstantFieldDeclaration
;
EmbeddedStatement
: Block
| lxmSemicolon // 8.3 The empty statement
| ExpressionStatement lxmSemicolon
  // 8.6 Expression statements
  // 8.7 Selection statements
  // 8.7.1 The if statement
| lxmIf lxmLeftParenth GeneralExpression
  lxmRightParenth EmbeddedStatement
  %prec lxmQuestion
| lxmIf lxmLeftParenth GeneralExpression
  lxmRightParenth EmbeddedStatement lxmElse
  EmbeddedStatement %prec lxmElse
  // 8.7.2 The switch statement
| lxmSwitch lxmLeftParenth GeneralExpression
  lxmRightParenth lxmLeftBrace SwitchSections
  lxmRightBrace
| lxmSwitch lxmLeftParenth GeneralExpression
  lxmRightParenth lxmLeftBrace lxmRightBrace
  // 8.8 Iteration statements
  // 8.8.1 The while statement
| lxmWhile lxmLeftParenth GeneralExpression
  lxmRightParenth EmbeddedStatement
  // 8.8.2 The do statement
| lxmDo EmbeddedStatement lxmWhile lxmLeftParenth
  GeneralExpression lxmRightParenth lxmSemicolon
  // 8.8.3 The for statement
| lxmFor lxmLeftParenth ForHeader lxmRightParenth
  EmbeddedStatement
  // 8.8.4 The foreach statement
| lxmForeach lxmLeftParenth Type lxmIdentifier

```

```

    lxmIn GeneralExpression lxmRightParenth
    EmbeddedStatement
    // 8.9 Jump statements
    // 8.9.1 The break statement
    | lxmBreak lxmSemicolon
    // 8.9.2 The continue statement
    | lxmContinue lxmSemicolon
    // 8.9.3 The goto statement
    | lxmGoto lxmIdentifier lxmSemicolon
    | lxmGoto lxmCase GeneralExpression lxmSemicolon
    | lxmGoto lxmDefault lxmSemicolon
    // 8.9.4 The return statement
    | lxmReturn lxmSemicolon
    | lxmReturn GeneralExpression lxmSemicolon
    // 8.9.5 The throw statement
    | lxmThrow lxmSemicolon
    | lxmThrow GeneralExpression lxmSemicolon
    // 8.10 The try statement
    | lxmTry Block SpecificCatchClauses
    GeneralCatchClause FinallyClause
    | lxmTry Block SpecificCatchClauses FinallyClause
    | lxmTry Block GeneralCatchClause FinallyClause
    | lxmTry Block SpecificCatchClauses
    GeneralCatchClause
    | lxmTry Block SpecificCatchClauses
    | lxmTry Block GeneralCatchClause
    | lxmTry Block FinallyClause
    // 8.11 The checked and unchecked statements
    | lxmChecked Block
    | lxmUnchecked Block
    // 8.12 The lock statement
    | lxmLock lxmLeftParenth GeneralExpression
    lxmRightParenth EmbeddedStatement
    // 8.13 The using statement
    | lxmUsing lxmLeftParenth
    ConstantFieldDeclaration lxmRightParenth
    EmbeddedStatement
    | lxmUsing lxmLeftParenth GeneralExpression
    lxmRightParenth EmbeddedStatement
    // A.6 The fixed statement (in unsafe context)
    | lxmFixed lxmLeftParenth Type
    FixedPointerDeclarators lxmRightParenth
    EmbeddedStatement
;
////////// 8.2 Blocks /////////////
Block
:
    lxmLeftBrace lxmRightBrace
    | lxmLeftBrace StatementList lxmRightBrace
;
////////// 8.2.1 Statement lists ///////////
StatementList
:
    Statement
    | StatementList Statement

```

```

;
////////// 8.4 Labeled statements ///////////
Labels
:
    lxmIdentifier lxmColon
| Labels lxmIdentifier lxmColon
;
////////// 8.5 Declaration statements
////////// //////////// //////////// ////////////
// See the first part of ConstantFieldDeclaration.
////////// 8.6 Expression statements
////////// //////////// //////////// ////////////
ExpressionStatement
:
    InvocationExpression
| NewExpression
| Assignment
| IncDecExpression
;
ExpressionStatementList
:
    ExpressionStatement
| ExpressionStatementList
    lxmComma ExpressionStatement
;
////////// 8.7.2 The switch statement ///////////
SwitchSections
:
    SwitchSection
| SwitchSections SwitchSection
;
SwitchSection
:
    SwitchLabels StatementList
;
SwitchLabels
:
    SwitchLabel
| SwitchLabels SwitchLabel
;
SwitchLabel
:
    lxmCase GeneralExpression lxmColon
| lxmDefault             lxmColon
;
////////// 8.8.3 The for statement ///////////
ForHeader
:
    ForInitializer GeneralExpressionOpt
    lxmSemicolon ExpressionStatementList
| ForInitializer GeneralExpressionOpt
    lxmSemicolon
;
ForInitializer
:
    lxmSemicolon
| ConstantFieldDeclaration
| ExpressionStatementList lxmSemicolon
;
////////// 8.10 The try statement ///////////
SpecificCatchClauses
:
    SpecificCatchClause

```

```

    | SpecificCatchClauses SpecificCatchClause
    ;
SpecificCatchClause
    : lxmCatch lxmLeftParenth Type lxmIdentifier
      lxmRightParenth Block
    | lxmCatch lxmLeftParenth Type lxmRightParenth
      Block
    ;
GeneralCatchClause
    : lxmCatch Block
    ;
FinallyClause
    : lxmFinally Block
    ;
////////// 17.2 Attribute specification //////////
AttributesOpt
    :
    | Attributes
    ;
Attributes
    : AttributeSection
    | Attributes AttributeSection
    ;
AttributeSection
    : lxmLeftBracket lxmIdentifier lxmColon
      AttributeList CommaRightBracket
    | lxmLeftBracket AttributeList CommaRightBracket
    ;
CommaRightBracket
    : lxmRightBracket
    | lxmComma lxmRightBracket
    ;
AttributeList
    : Attribute
    | AttributeList lxmComma Attribute
    ;
Attribute
    : QualifiedIdentifier lxmLeftParenth
      AttributeArguments lxmRightParenth
    ;
AttributeArguments
    : GeneralExpression
    | AttributeArguments lxmComma GeneralExpression
    ;
////////// Modifiers ///////////
Modifiers
    : Modifier
    | Modifiers Modifier
    ;
Modifier
    : lxmNew          %prec lxmQuestion
    | lxmPublic
    | lxmProtected

```

```

| lxmInternal
| lxmPrivate
| lxmAbstract
| lxmSealed
| lxmStatic
| lxmReadOnly
| lxmVirtual
| lxmOverride
| lxmExtern
| lxmConst
| lxmUnsafe
;

////////// A.6 The fixed statement ///////////
FixedPointerDeclarators
:
| FixedPointerDeclarator
| FixedPointerDeclarators lxmComma
    FixedPointerDeclarator
;
FixedPointerDeclarator
:
| lxmIdentifier lxmEqual lxmAmpersand
    GeneralExpression
| GeneralExpression
|
| lxmIdentifier lxmEqual GeneralExpression
;
// %

```

Библиография

1. Abel N. E. and Bell J. R. *Global optimization in compilers*, Proc. First USA-Japan Computer Conf., AFIPS Press, Montvale, N. J., 1972.
2. Abelson H. and Sussman G. J. *Structure and Interpretation of Computer Programs*, MIT Press, Cambridge, Mass, 1985.
3. Adriion W. R., Branstad M. A. and Cherniavsky J. C. *Validation, verification, and testing of computer software*, Computing Surveys **14:2**, 1982, pp. 159–192.
4. Aho A. V. *Pattern matching in strings*, [60], pp. 325–347.
5. Aho A. V. and Corasick M. J. *Efficient string matching: an aid to bibliographic search*, Comm. ACM **18:6**, 1975, pp. 333–340.
6. Aho A. V. and Ganapathi M. *Efficient tree pattern matching: an aid to code generation*, Twelfth Annual ACM Symposium on Principles of Programming Languages, 1985, pp. 334–340.
7. Aho A. V., Hopcroft J. E. and Ullman J. D. *The Design and Analysis of Computer Algorithms*, Addison-Wesley, Reading, Mass, 1974.
8. Aho A. V., Hopcroft J. E. and Ullman J. D. *Data Structures and Algorithms*, Addison-Wesley, Reading, Mass, 1983. (Ахо А.В., Хопкрофт Д. Э., Ульман Д. Д. *Структуры данных и алгоритмы*. — М.: Издательский дом “Вильямс”, 2000.)
9. Aho A. V. and Johnson S. C. *LR parsing*, Computing Surveys **6:2**, 1974, pp. 99–124.
10. Aho A. V. and Johnson S. C. *Optimal code generation for expression trees*, J. ACM **23:3**, 1976, pp. 488–501.
11. Aho A. V., Johnson S. C. and Ullman J. D. *Deterministic parsing of ambiguous grammars*, Comm. ACM **18:8**, 1975, pp. 441–452.
12. Aho A. V., Johnson S. C. and Ullman J. D. *Code generation for expressions with common subexpressions*, J. ACM **24:1**, 1977, pp. 146–160.
13. Aho A. V., Johnson S. C. and Ullman J. D. *Code generation for machines with multiregister operations*, Fourth ACM Symposium on Principles of Programming Languages, 1977, pp. 21–28.
14. Aho A. V., Kernighan B. W. and Weinberger P. J. *Awk — a pattern scanning and processing language*, Software—Practice and Experience **9:4**, 1979, pp. 267–280.
15. Aho A. V. and Peterson T. G. *A minimum distance error-correcting parser for context-free languages*, SIAM J. Computing **1:4**, 1972, pp. 305–312.
16. Aho A. V. and Sethi R. *How hard is compiler code generation?* Lecture Notes in Computer Science **52**, Springer-Verlag, Berlin, 1977, pp. 1–15.
17. Aho A. V. and Ullman J. D. *Optimization of straight line code*, SIAM J. Computing **1:1**, 1972, pp. 1–19.
18. Aho A. V. and Ullman J. D. *The Theory of Parsing, Translation and Compiling*, Vol. I: *Parsing*, Prentice-Hall, Englewood Cliffs, N. J., 1972 (Ахо А.В., Ульман Д.Д. *Теория синтаксического анализа, перевода и компиляции. Т.1. Синтаксический анализ*. — М.:Мир, 1978.)

19. Aho A. V. and Ullman J. D. *The Theory of Parsing, Translation and Compiling*, Vol. II: Compiling, Prentice-Hall, Englewood Cliffs, N. J., 1973 (Ахо А.В., Ульман Д.Д. *Теория синтаксического анализа, перевода и компиляции. Т.2. Компиляция.* — М.:Мир, 1978.)
20. Aho A. V. and Ullman J. D. *A technique for speeding up LR(k) parsers*, SIAM J. Computing 2:2, 1973, pp. 106–127.
21. Aho A. V. and Ullman. J. D. *Principles of Compiler Design*, Addison-Wesley, Reading, Mass, 1977.
22. Aigrain P., Graham S. L., Henry R. R., McKusick M. K. and Pelegri-Llopert E. *Experience with a Graham-Glanville style code generator*, ACM SIGPLAN Notices 19:6, 1984, pp. 13–24.
23. Allen F. E. *Program optimization*, Annual Review in Automatic Programming 5, 1969, pp. 239–307.
24. Allen F. E. *Control flow analysis*, ACM SIGPLAN Notices 5:7, 1970, pp. 1–19.
25. Allen F. E. *Interprocedural data flow analysis*, Information Processing 74, North-Holland, Amsterdam, 1974, pp. 398–402.
26. Allen F. E. *Bibliography on program optimization*, RC-5767, IBM T. J. Watson Research Center, Yorktown Heights, N. Y., 1975.
27. Allen F. E., Carter J. L., Fabri J., Ferrante J., Harrison W. H., Loewner P. G. and Trevillyan L. H. *The experimental compiling system*, IBM. J. Research and Development 24:6, 1980, pp. 695–715.
28. Allen F. E. and Cocke J. *A catalogue of optimizing transformations*, [384], pp. 1–30.
29. Allen F. E. and Cocke J. *A program data flow analysis procedure*, Comm. ACM 19:3, 1976, pp. 137–147.
30. Allen F. E., Cocke J. and Kennedy K. *Reduction of operator strength*, [326], pp. 79–101.
31. Ammann U. *On code generation in a Pascal compiler*, Software—Practice and Experience 7:3, 1977, pp. 391–423.
32. Ammann U. *The Zurich implementation*, [45], pp. 63–82.
33. Anderson J. P. *A note on some compiling algorithms*, Comm. ACM 7:3, 1964, pp. 149–150.
34. Anderson T., Eve J. and Horning J. J. *Efficient LR(1) parsers*, Acta Informatica 2:1, 1973, pp. 12–39.
35. Anklam P., Cutler D., Heinen R., Jr. and MacLaren M. D. *Engineering a Compiler*, Digital Press, Bedford, Mass, 1982.
36. Arden B. W., Galler B. A. and Graham R. M. *An algorithm for equivalence declarations*, Comm. ACM 4:7, 1961, pp. 310–314.
37. Auslander M. A. and Hopkins M. E. *An overview of the PL.8 compiler*, ACM SIGPLAN Notices 17:6, 1982, pp. 22–31.
38. Backhouse R. C. *An alternative approach to the improvement of LR parsers*, Acta Informatica 6:3, 1976, pp. 277–296.
39. Backhouse R. C. *Global data flow analysis problems arising in locally least-cost error recovery*, TOPLAS 6:2, 1984, pp. 192–214.

40. Backus J. W. *Transcript of presentation on the history of Fortran I, II, and III*, [454], pp. 45–66.
41. Backus J. W., Beeber R. J., Best S., Goldberg R., Haibt L. M., Herrick H. L., Nelson R. A., Sayre D., Sheridan P. B., Stern H., Ziller I., Hughes R. A. and Nutt R. *The Fortran automatic coding system*, Western Joint Computer Conference, 1957, pp. 188–198; Reprinted in [381], pp. 29–47.
42. Baker B. S. *An algorithm for structuring programs*, J. ACM 24:1, 1977, pp. 98–120.
43. Baker T. P. *A one-pass algorithm for overload resolution in Ada*, TOPLAS 4:4, 1982, pp. 601–614.
44. Banning J. P. *An efficient way to find the side effects of procedure calls and aliases of variables*, Sixth Annual ACM Symposium on Principles of Programming Languages, 1979, pp. 29–41.
45. Barron D. W. *Pascal — The Language and its Implementation*, Wiley, Chichester, 1981.
46. Barth J. M. *A practical interprocedural data flow analysis algorithm*, Comm. ACM 21:9, 1978, pp. 724–736.
47. Batson A. *The organization of symbol tables*, Comm. ACM 8:2, 1965, pp. 111–112.
48. Bauer A. M. and Saal H. J. *Does APL really need run-time checking?* Software — Practice and Experience 4:2, 1974, pp. 129–138.
49. Bauer F. L. *Historical remarks on compiler construction*, [50], pp. 603–621. Addendum by A. P. Ershov, 1976, pp. 622–626.
50. Bauer F. L. and Eickel J. *Compiler Construction: An Advanced Course*, 2nd Ed., Lecture Notes in Computer Science 21, Springer-Verlag, Berlin, 1976.
51. Bauer F. L. and Wössner H. *The ‘Plankalkül’ of Konrad Zuse: A forerunner of today’s programming languages*, Comm. ACM 15:7, 1972, pp. 678–685.
52. Beatty J. C. *An axiomatic approach to code optimization for expressions*, J. ACM 19:4, 1972, pp. 714–724; Errata 20, 1973, pp. 180 and 538.
53. Beatty J. C. *Register assignment algorithm for generation of highly optimized object code*, IBM J. Research and Development 5:2, 1974, pp. 20–39.
54. Belady L. A. *A study of replacement algorithms for a virtual storage computer*, IBM Systems J. 5:2, 1966, pp. 78–101.
55. Bentley J. L. *Writing Efficient Programs*, Prentice-Hall, Englewood Cliffs, N. J., 1982.
56. Bentley J. L., Cleveland W. S. and Sethi R. *Empirical analysis of hash functions*, manuscript, AT&T Bell Laboratories, Murray Hill, N. J., 1985.
57. Birman A. and Ullman J. D. *Parsing algorithms with backtrack*, Information and Control 23:1, 1973, pp. 1–34.
58. Bochmann G. V. *Semantic evaluation from left to right*, Comm. ACM 19:2, 1976, pp. 55–62.
59. Bochmann G. V. and Ward P. *Compiler writing system for attribute grammars*, Computer J. 21:2, 1978, pp. 144–148.
60. Book R. V. *Formal Language Theory*, Academic Press, New York, 1980.
61. Boyer R. S. and Moore J. S. *A fast string searching algorithm*, Comm. ACM 20:10, 1977, pp. 262–272.

62. Branquart P., Cardinael J.-P., Lewi J., Delescaille J.-P. and Vanbegin M. *An Optimized Translation Process and its Application to Algol 68*, Lecture Notes in Computer Science, **38**, Springer-Verlag, Berlin, 1976.
63. Bratman H. *An alternate form of the 'Uncol diagram*, Comm. ACM **4:3**, 1961, p. 142.
64. Brooker R. A. and Morris D. *A general translation program for phrase structure languages*, J. ACM **9:1**, 1962, pp. 1–10.
65. Brooks F. P., Jr. *The Mythical Man-Month*, Addison-Wesley, Reading, Mass, 1975.
66. Brosgol B. M. *Deterministic Translation Grammars*, Ph. D. Thesis, TR 3-74, Harvard Univ., Cambridge, Mass, 1974.
67. Bruno J. and Lassagne T. *The generation of optimal code for stack machines*, J. ACM **22:3**, 1975, pp. 382–396.
68. Bruno J. and Sethi R. *Code generation for a one-register machine*, J. ACM **23:3**, 1976, pp. 502–510.
69. Burstall R. M., MacQueen D. B. and Sannella D. T. *Hope: an experimental applicative language*, Lisp Conference, P.O. Box 487, Redwood Estates, Calif. 95044, 1980, pp. 136–143.
70. Busam V. A. and Englund D. E. *Optimization of expressions in Fortran*, Comm. ACM **12:12**, 1969, pp. 666–674.
71. Cardelli L. *Basic polymorphic typechecking*, Computing Science Technical Report 112, AT&T Bell Laboratories, Murray Hill, N. J., 1984.
72. Carter L. R. *An Analysis of Pascal Programs*, UMI Research Press, Ann Arbor, Michigan, 1982.
73. Cartwright R. *Types as intervals*, Twelfth Annual ACM Symposium on Principles of Programming Languages, 1985, pp. 22–36.
74. Cattell R. G. G. *Automatic derivation of code generators from machine descriptions*, TOPLAS **2:2**, 1980, pp. 173–190.
75. Chaitin G. J. *Register allocation and spilling via graph coloring*, ACM SIGPLAN Notices **17:6**, 1982, pp. 201–207.
76. Chaitin G. J., Auslander M. A., Chandra A. K., Cocke J., Hopkins M. E. and Markstein P. W. *Register allocation via coloring*, Computer Languages **6**, 1981, pp. 47–57.
77. Cherniavsky J. C., Henderson P. B. and Keohane J. *On the equivalence of URE flow graphs and reducible flow graphs*, Proc. 1976 Conference on Information Sciences and Systems, Johns Hopkins Univ., 1976, pp. 423–429.
78. Cherry L. L. *Writing tools*, IEEE Trans. on Communications **COM-30:1**, 1982, pp. 100–104.
79. Chomsky N. *Three models for the description of language*, IRE Trans. on Information Theory **IT-2:3**, 1956, pp. 113–124. (Хомский Н. *Три модели для описания языка*/Кибернетический сборник. — М.:ИЛ, 1961. — Вып.2. — С. 237–266.)
80. Chow F. *A Portable Machine-Independent Global Optimizer*, Ph. D. Thesis, Computer System Lab., Stanford Univ., Stanford, Calif, 1983.
81. Chow F. and Hennessy J. L. *Register allocation by priority-based coloring*, ACM SIGPLAN Notices **19:6**, 1984, pp. 222–232.

82. Church A. *The Calculi of Lambda Conversion*, Annals of Math. Studies, No. 6, Princeton University Press, Princeton, N. J., 1941.
83. Church A. *Introduction to Mathematical Logic*, Vol. I, Princeton University Press, Princeton, N. J., 1956.
84. Ciesinger J. *A bibliography of error handling*, ACM SIGPLAN Notices **14:1**, 1979, pp. 16–26.
85. Cocke J. *Global common subexpression elimination*, ACM SIGPLAN Notices **5:7**, 1970, pp. 20–24.
86. Cocke J. and Kennedy K. *Profitability computations on program flow graphs*, Computers and Mathematics with Applications **2:2**, 1976, pp. 145–159.
87. Cocke J. and Kennedy K. *An algorithm for reduction of operator strength*, Comm. ACM **20:11**, 1977, pp. 850–856.
88. Cocke J. and Markstein J. *Measurement of code improvement algorithms*, Information Processing 80, 1980, pp. 221–228.
89. Cocke J. and Miller J. *Some analysis techniques for optimizing computer programs*, Proc. 2nd Hawaii Intl. Conf. on Systems Sciences, 1969, pp. 143–146.
90. Cocke J. and Schwartz J. T. *Programming Languages and Their Compilers: Preliminary Notes, Second Revised Version*, Courant Institute of Mathematical Sciences, New York, 1970.
91. Coffman E. G., Jr. and Sethi R. *Instruction sets for evaluating arithmetic expressions*, J. ACM **30:3**, 1983, pp. 457–478.
92. Cohen R. and Harry E. *Automatic generation of near-optimal linear-time translators for non-circular attribute grammars*, Sixth ACM Symposium on Principles of Programming Languages, 1979, pp. 121–134.
93. Conway M. E. *Design of a separable transition diagram compiler*, Comm. ACM **6:7**, 1963, pp. 396–408. (Конвэй М. Проект делимого компилятора, основанного на диаграммах перехода//Сб. “Современное программирование”. — М.:Сов. радио, 1967. — С. 206–246.)
94. Conway R. W. and Maxwell W. L. *CORC — the Cornell computing language*, Comm. ACM **6:6**, 1963, pp. 317–321.
95. Conway R. W. and Wilcox T. R. *Design and implementation of a diagnostic compiler for PL/I*, Comm. ACM **16:3**, 1973, pp. 169–179.
96. Cormack G. V. *An algorithm for the selection of overloaded functions in Ada*, ACM SIGPLAN Notices **16:2** (February, 1981), pp. 48–52.
97. Cormack G. V., Horspool R. N. S. and Kaiserswerth M. *Practical perfect hashing*, Computer J. **28:1**, 1985, pp. 54–58.
98. Courcelle B. *Attribute grammars: definitions, analysis of dependencies, proof methods*, [291], pp. 81–102.
99. Cousot P. *Semantic foundations of program analysis*, [326], pp. 303–342.
100. Cousot P. and Cousot R. *Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints*, Fourth ACM Symposium on Principles of Programming Languages, 1977, pp. 238–252.
101. Curry H. B. And Feys R. *Combinatory Logic*, vol. 1, North-Holland, Amsterdam, 1958.

102. Date C. J. *An Introduction to Database Systems*, 4th Ed., Addison-Wesley, Reading, Mass, 1986. (Дейт К. Введение в системы баз данных. — 7-е изд. — М.: Издательский дом “Вильямс”, 2001.)
103. Davidson J. W. and Fraser C. W. *The design and application of a retargetable peephole optimizer*, TOPLAS 2:2, 1980, pp. 191–202; Errata 3:1, 1981, p. 110.
104. Davidson J. W. and Fraser C. W. *Automatic generation of peephole optimizations*, ACM SIGPLAN Notices 19:6, 1984, pp. 111–116.
105. Davidson J. W. and Fraser C. W. *Code selection through object code optimization*, TOPLAS 6:4, 1984, pp. 505–526.
106. DeRemer F. *Practical Translators for LR(k) Languages*, Ph. D. Thesis, M.I.T., Cambridge, Mass, 1969.
107. DeRemer F. *Simple LR(k) grammars*, Comm. ACM 14:7, 1971, pp. 453–460.
108. DeRemer F. and Pennello T. *Efficient computation of LALR(1) look-ahead sets*, TOPLAS 4:4, 1982, pp. 615–649.
109. Demers A. J. *Elimination of single productions and merging of nonterminal symbols in LR(1) grammars*, J. Computer Languages 1:2, 1975, pp. 105–119.
110. Dencker P., Dürre K. and Heuft J. *Optimization of parser tables for portable compilers*, TOPLAS 6:4, 1984, pp. 546–572.
111. Deransart P., Jourdan M. and Lorho B. *Speeding up circularity tests for attribute grammars*, Acta Informatica 21, 1984, pp. 375–391.
112. Despeyroux T. *Executable specifications of static semantics*, [228], pp. 215–233.
113. Dijkstra E. W. *Recursive programming*, Numerische Math. 2, 1960, pp. 312–318; Reprinted in [381], pp. 221–228.
114. Dijkstra E. W. *An Algol 60 translator for the XI*, Annual Review in Automatic Programming 3, Pergamon Press, New York, 1963, pp. 329–345.
115. Ditzel D. and McLellan H. R. *Register allocation for free: the C machine stack cache*, Proc. ACM Symp. on Architectural Support for Programming Languages and Operating Systems, 1982, pp. 48–56.
116. Downey P. J. and Sethi R. *Assignment commands with array references*, J. ACM 25:4, 1978, pp. 652–666.
117. Downey P. J., Sethi R. and Tarjan R. E. *Variations on the common subexpression problem*, J. ACM 27:4, 1980, pp. 758–771.
118. Earley J. *An efficient context-free parsing algorithm*, Comm. ACM 13:2, 1970, pp. 94–102. (Эрли Дж. Эффективный алгоритм анализа контекстно-свободных языков//Сб. “Языки и автоматы”. — М.:Мир, 1975. — С. 47–70.)
119. Earley J. *Ambiguity and precedence in syntax description*, Acta Informatica 4:2, 1975, pp. 183–192.
120. Earley J. *High level iterators and a method of data structure choice*, J. Computer Languages 1:4, 1975, pp. 321–342.
121. Elshoff J. L. *An analysis of some commercial PL/I programs*, IEEE Trans. Software Engineering SE2:2, 1976, pp. 113–120.
122. Engelfriet J. *Attribute evaluation methods*, [291], pp. 103–138.

123. Ershov A. P. *On programming of arithmetic operations*, Comm. ACM **1:8** (August, 1958), 3-6. Figures 1-3 appear in **1:9** (September, 1958), p. 16.
124. Ershov A. P. *Alpha — an automatic programming system of high efficiency*, J. ACM **13:1**, 1966, pp. 17–24.
125. Ershov A. P. *The Alpha Automatic Programming System*, Academic Press, New York, 1971.
126. Ershov A. P. and Koster C. H. A. *Methods of Algorithmic Language Implementation*, Lecture Notes in Computer Science **47**, Springer-Verlag, Berlin, 1977.
127. Fang I. *FOLDS, a declarative formal language definition system*, STAN-CS-72-329, Stanford Univ, 1972.
128. Farrow R. *Generating a production compiler from an attribute grammar*, IEEE Software **1** (October, 1984), pp. 77–93.
129. Farrow R. and Yellin D. *A comparison of storage optimizations in automatically-generated compilers*, manuscript, Columbia Univ, 1985.
130. Feldman S. I. *Make — a program for maintaining computer programs*, Software—Practice and Experience **9:4**, 1979, pp. 255–265.
131. Feldman S. I. *Implementation of a portable Fortran 77 compiler using modern tools*, ACM SIGPLAN Notices **14:8**, 1979, pp. 98–106.
132. Fischer M. J. *Efficiency of equivalence algorithms*, [315], pp. 153–168.
133. Fleck A. C. *The impossibility of content exchange through the byname parameter transmission technique*, ACM SIGPLAN Notices **11:11** (November, 1976), pp. 38–41.
134. Floyd R. W. *An algorithm for coding efficient arithmetic expressions*, Comm. ACM **4:1**, 1961, pp. 42–51.
135. Floyd R. W. *Syntactic analysis and operator precedence*, J. ACM **10:3**, 1963, pp. 316–333.
136. Floyd R. W. *Bounded context syntactic analysis*, Comm. ACM **7:2**, 1964, pp. 62–67.
137. Fong A. C. *Automatic improvement of programs in very high level languages*, Sixth Annual ACM Symposium on Principles of Programming Languages, 1979, pp. 21–28.
138. Fong A. C. and Ullman J. D. *Induction variables in very high-level languages*, Third Annual ACM Symposium on Principles of Programming Languages, 1976, pp. 104–112.
139. Fosdick L. D. and Osterweil L. J. *Data flow analysis in software reliability*, Computing Surveys **8:3**, 1976, pp. 305–330.
140. Foster J. M. *A syntax improving program*, Computer J. **11:1**, 1968, pp. 31–34.
141. Fraser C. W. *Automatic Generation of Code Generators*, Ph. D. Thesis, Yale Univ., New Haven, Conn, 1977.
142. Fraser C. W. *A compact, machine-independent peephole optimizer*, Sixth Annual ACM Symposium on Principles of Programming Languages, 1979, pp. 1–6.
143. Fraser C. W. and Hanson D. R. *A machine-independent linker*, Software—Practice and Experience **12**, 1982, pp. 351–366.
144. Fredman M. L., Komlós J. and Szemerédi E. *Storing a sparse table with $O(1)$ worst case access time*, J. ACM **31:3**, 1984, pp. 538–544.
145. Frege G. *Begriffsschrift, a formula language, modeled upon that of arithmetic, for pure thought* (1879), Reprinted in [182], pp. 1–82.

146. Freiburghouse R. A. *The Multics PL/I compiler*, AFIPS Fall Joint Computer Conference 35, 1969, pp. 187–208.
147. Freiburghouse R. A. *Register allocation via usage counts*, Comm. ACM 17:11, 1974, pp. 638–642.
148. Freudberger S. M. *On the use of global optimization algorithms for the detection of semantic programming errors*, NSO-24, New York Univ, 1984.
149. Freudberger S. M., Schwartz J. T. and Sharir M. *Experience with the SETL optimizer*, TOPLAS 5:1, 1983, pp. 26–45.
150. Gajewska H. *Some statistics on the usage of the C language*, AT&T Bell Laboratories, Murray Hill, N. J., 1975.
151. Galler B. A. and Fischer M. J. *An improved equivalence algorithm*, Comm. ACM 7:5, 1964, pp. 301–303.
152. Ganapathi M. *Retargetable Code Generation and Optimization using Attribute Grammars*, Ph. D. Thesis, Univ. of Wisconsin, Madison, Wis, 1980.
153. Ganapathi M. and Fischer C. N. *Description-driven code generation using attribute grammars*, Ninth ACM Symposium on Principles of Programming Languages, 1982, pp. 108–119.
154. Ganapathi M., Fischer C. N. and Hennessy J. L. *Retargetable compiler code generation*, Computing Surveys 14:4, 1982, pp. 573–592.
155. Gannon J. D. and Horning J. J. *Language design for programming reliability*, IEEE Trans. Software Engineering SE-1:2, 1975, pp. 179–191.
156. Ganzinger H., Giegerich R., Moncke U. and Wilhelm R. *A truly generative semantics-directed compiler generator*, ACM SIGPLAN Notices 17:6 (June, 1982), pp. 172–184.
157. Ganzinger H. and Ripken K. *Operator identification in Ada*, ACM SIGPLAN Notices 15:2 (February, 1980), pp. 30–42.
158. Garey M. R. and Johnson D. S. *Computers and Intractability: A Guide to the Theory of NP-Completeness*, Freeman, San Francisco, 1979. (Гэри М., Джонсон Д. *Вычислимые машины и труднорешаемые задачи*. — М.: Мир, 1982.)
159. Gear C. W. *High speed compilation of efficient object code*, Comm. ACM 8:8, 1965, pp. 483–488.
160. Geschke C. M. *Global Program Optimizations*, Ph. D. Thesis, Dept. of Computer Science, Carnegie-Mellon Univ, 1972.
161. Giegerich R. *A formal framework for the derivation of machine-specific optimizers*, TOPLAS 5:3, 1983, pp. 422–448.
162. Giegerich R. and Wilhelm R. *Counter-one-pass features in one-pass compilation: a formalization using attribute grammars*, Information Processing Letters 7:6, 1978, pp. 279–284.
163. Glanville R. S. *A Machine Independent Algorithm for Code Generation and its Use in Retargetable Compilers*, Ph. D. Thesis, Univ. of California, Berkeley, 1977.
164. Glanville R. S. and Graham S. L. *A new method for compiler code generation*, Fifth ACM Symposium on Principles of Programming Languages, 1978, pp. 231–240.
165. Graham R. M. *Bounded context translation*, AFIPS Spring Joint Computer Conference 40, 1964, pp. 205–217. Reprinted in [381], pp. 184–205.

166. Graham S. L. *Table-driven code generation*, Computer **13:8**, 1980, pp. 25–34.
167. Graham S. L. *Code generation and optimization*, [291], pp. 251–288.
168. Graham S. L., Haley C. B. and Joy W. N. *Practical LR error recovery*, ACM SIGPLAN Notices **14:8**, 1979, pp. 168–175.
169. Graham S. L., Harrison M. A. and Ruzzo W. L. *An improved context-free recognizer*, TOPLAS **2:3**, 1980, pp. 415–462.
170. Graham S. L. and Rhodes S. P. *Practical syntactic error recovery*, Comm. ACM **18:11**, 1975, pp. 639–650.
171. Graham S. L. and Wegman M. *A fast and usually linear algorithm for global data flow analysis*, J. ACM **23:1**, 1976, pp. 172–202.
172. Grau A. A., Hill U. and Langmaack H. *Translation of Algol 60*, Springer-Verlag, New York, 1967.
173. Hanson D. R. *Is block structure necessary?* Software—Practice and Experience **11**, 1981, pp. 853–866.
174. Harrison M. C. *Implementation of the substring test by hashing*, Comm. ACM **14:12**, 1971, pp. 777–779.
175. Harrison W. *A class of register allocation algorithms*, RC-5342, IBM T. J. Watson Research Center, Yorktown Heights, N. Y., 1975.
176. Harrison W. *Compiler analysis of the value ranges for variables*, IEEE Trans. Software Engineering **3:3**, 1977.
177. Hecht M. S. *Flow Analysis of Computer Programs*, North-Holland, New York, 1977.
178. Hecht M. S. and Shaffer J. B. *Ideas on the design of a 'quad improver' for SIMPL-T, part I: overview and intersegment analysis*, Dept. of Computer Science, Univ. of Maryland, College Park, Md., 1975.
179. Hecht, M. S. and Ullman J. D. *Flow graph reducibility*, SIAM J. Computing **1**, 1972, pp. 188–202.
180. Hecht M. S. and Ullman J. D. *Characterizations of reducible flow graphs*, J. ACM **21**, 1974, pp. 367–375.
181. Hecht M. S. and Ullman J. D. *A simple algorithm for global data flow analysis programs*, SIAM J. Computing **4**, 1975, pp. 519–532.
182. Heijenoort J. van. *From Frege to Gödel*, Harvard Univ. Press, Cambridge, Mass, 1967.
183. Hennessy J. *Program optimization and exception handling*, Eighth Annual ACM Symposium on Principles of Programming Languages, 1981, pp. 200–206.
184. Hennessy J. *Symbolic debugging of optimized code*, TOPLAS **4:3**, 1982, pp. 323–344.
185. Henry R. R. *Graham-Glanville Code Generators*, Ph. D. Thesis, Univ. of California, Berkeley, 1984.
186. Hext J. B. *Compile time type-matching*, Computer J. **9**, 1967, pp. 365–369.
187. Hindley R. *The principal type-scheme of an object in combinatory logic*, Trans. AMS **146**, 1969, pp. 29–60.
188. Hoare C. A. R. *Quicksort*, Computer J. **5:1**, 1962, pp. 10–15.
189. Hoare C. A. R. *Report on the Elliott Algol translator*, Computer J. **5:2**, 1962, pp. 127–129.

190. Hoffman C. M. and O'Donnell M. J. *Pattern matching in trees*, J. ACM **29:1**, 1982, pp. 68–95.
191. Hopcroft J. E. and Karp R. M. *An algorithm for testing the equivalence of finite automata*, TR-71-114, Dept. of Computer Science, Cornell Univ, 1971. See [7], pp. 143–145.
192. Hopcroft J. E. and Ullman J. D. *Formal Languages and Their Relation to Automata*, Addison-Wesley, Reading, Mass, 1969.
193. Hopcroft J. E. and Ullman J. D. *Set merging algorithms*, SIAM J. Computing **2:3**, 1973, pp. 294–303.
194. Hopcroft J. E. and Ullman J. D. *Introduction to Automata Theory, Languages, and Computation*, Addison-Wesley, Reading, Mass, 1979.
195. Horning J. J. *What the compiler should tell the user*, [50].
196. Horwitz L. P., Karp R. M., Miller R. E. and Winograd S. *Index register allocation*, J. ACM **13:1**, 1966, pp. 43–61.
197. Huet G. and Kahn G. (eds.). *Proving and Improving Programs*, Colloque IRIA, Arc-et-Senans, France, 1975.
198. Huet G. and Levy J.-J. *Call-by-need computations in nonambiguous linear term rewriting systems*, Rapport de Recherche 359, INRIA Laboria, Rocquencourt, 1979.
199. Huffman D. A. *The synthesis of sequential machines*, J. Franklin Inst. **257**, 1954, pp. 3–4, p. 161, p. 190, pp. 275–303.
200. Hunt J. W. and McIlroy M. D. *An algorithm for differential file comparison*, Computing Science Technical Report 41, AT&T Bell Laboratories, Murray Hill, N. J., 1976.
201. Hunt J. W. and Szymanski T. G. *A fast algorithm for computing longest common subsequences*, Comm. ACM **20:5**, 1977, pp. 350–353.
202. Huskey H. D., Halstead M. H. and McArthur R. *Neliac — a dialect of Algol*, Comm. ACM **3:8**, 1960, pp. 463–468.
203. Ichbiah J. D. and Morse S. P. *A technique for generating almost optimal Floyd-Evans productions for precedence grammars*, Comm. ACM **13:8**, 1970, pp. 501–508.
204. Ingalls D. H. H. *The Smalltalk-76 programming system design and implementation*, Fifth Annual ACM Symposium on Principles of Programming Languages, 1978, pp. 9–16.
205. Ingerman P. Z. *Panini-Backus form suggested*, Comm. ACM **10:3**, 1967, p. 137.
206. Irons E. T. *A syntax directed compiler for Algol 60*, Comm. ACM **4:1**, 1961, pp. 51–55.
207. Irons E. T. *An error correcting parse algorithm*, Comm. ACM **6:11**, 1963, pp. 669–673.
208. Iverson K. *A Programming Language*, Wiley, New York, 1962.
209. Janas J. M. *A comment on “Operator identification in Ada” by Ganzinger and Ripken*, ACM SIGPLAN Notices **15:9** (September, 1980), pp. 39–43.
210. Jarvis J. F. *Feature recognition in line drawings using regular expressions*, Proc. 3rd Intl. Joint Conf. on Pattern Recognition, 1976, pp. 189–192.
211. Jazayeri M., Ogden W. F. and Rounds W. C. *The intrinsic exponential complexity of the circularity problem for attribute grammars*, Comm. ACM **18:12**, 1975, pp. 697–706.
212. Jazayeri M. and Pozefsky D. *Space-efficient storage management in an attribute grammar evaluator*, TOPLAS **3:4**, 1981, pp. 388–404.

213. Jazayeri M. and Walter K. G. *Alternating semantic evaluator*, Proc. ACM Annual Conference, 1975, 230–234.
214. Jensen K. and Wirth N. *Pascal User Manual and Report*, Springer-Verlag, New York, 1975. (Вирт Н., Йенсен К. *Паскаль: руководство для пользователя и описание языка*. — М.: Финансы и статистика. — 1982.)
215. Johnson S. C. *Yacc — yet another compiler compiler*, Computing Science Technical Report 32, AT&T Bell Laboratories, Murray Hill, N. J., 1975.
216. Johnson S. C. *A portable compiler: theory and practice*, Fifth Annual ACM Symposium on Principles of Programming Languages, 1978, pp. 97–104.
217. Johnson S. C. *A tour through the portable C compiler*, AT&T Bell Laboratories, Murray Hill, N. J., 1979.
218. Johnson S. C. *Code generation for silicon*, Tenth Annual ACM Symposium on Principles of Programming Languages, 1983, pp. 14–19.
219. Johnson S. C. and Lesk M. E. *Language development tools*, Bell System Technical J. 57:6, 1978, pp. 2155–2175.
220. Johnson S. C. and Ritchie D. M. *The C language calling sequence*, Computing Science Technical Report 102, AT&T Bell Laboratories, Murray Hill, N. J., 1981.
221. Johnson W. L., Porter J. H., Ackley S. I. and Ross D. T. *Automatic generation of efficient lexical processors using finite state techniques*, Comm. ACM 11:12, 1968, pp. 805–813.
222. Johnsson R. K. *An Approach to Global Register Allocation*, Ph. D. Thesis, Carnegie-Mellon Univ., Pittsburgh, Pa., 1975.
223. Joliat M. L. *A simple technique for partial elimination of unit productions from LR(k) parser tables*, IEEE Trans. on Computers C-25:7, 1976, pp. 763–764.
224. Jones N. D. *Semantics Directed Compiler Generation*, Lecture Notes in Computer Science 94, Springer-Verlag, Berlin, 1980.
225. Jones N. D. and Madsen C. M. *Attribute-influenced LR parsing*, in Jones, 1980, pp. 393–407.
226. Jones N. D. and Muchnick S. S. *Binding time optimization in programming languages*, Third ACM Symposium on Principles of Programming Languages, 1976, pp. 77–94.
227. Jourdan M. *Strongly noncircular attribute grammars and their recursive evaluation*, ACM SIGPLAN Notices 19:6, 1984, pp. 81–93.
228. Kahn G., MacQueen D. B. and Plotkin G. *Semantics of Data Types*, Lecture Notes in Computer Science 173, Springer-Verlag, Berlin, 1984.
229. Kam J. B. and Ullman J. D. *Global data flow analysis and iterative algorithms*, J. ACM 23:1, 1976, pp. 158–171.
230. Kam J. B. and Ullman J. D. *Monotone data flow analysis frameworks*, Acta Informatica 7:3, 1977, pp. 305–318.
231. Kaplan M. and Ullman J. D. *A general scheme for the automatic inference of variable types*, J. ACM 27:1, 1980, pp. 128–145.
232. Kasami T. *An efficient recognition and syntax analysis algorithm for context-free languages*, AFCRL-65-758, Air Force Cambridge Research Laboratory, Bedford, Mass, 1965.

233. Kasami T., Peterson W. W. and Tokura N. *On the capabilities of while, repeat, and exit statements*, Comm. ACM **16**:8, 1973, pp. 503–512.
234. Kastens U. *Ordered attribute grammars*, Acta Informatica **13**:3, 1980, pp. 229–256.
235. Kastens U., Hutt B. and Zimmermann E. *GAG: A Practical Compiler Generator*, Lecture Notes in Computer Science **141**, Springer-Verlag, Berlin, 1982.
236. Kasyanov V. N. *Some properties of fully reducible graphs*, Information Processing Letters **2**:4, 1973, pp. 113–117.
237. Katayama T. *Translation of attribute grammars into procedures*, TOPLAS **6**:3, 1984, pp. 345–369.
238. Kennedy K. *A global flow analysis algorithm*, InterN. J. Computer Math. Section A **3**, 1971, pp. 5–15.
239. Kennedy K. *Index register allocation in straight line code and simple loops*, [384], pp. 51–64.
240. Kennedy K. *A comparison of two algorithms for global flow analysis*, SIAM J. Computing **5**:1, 1976, pp. 158–180.
241. Kennedy K. *A survey of data flow analysis techniques*, [326], pp. 5–54.
242. Kennedy K. and Ramanathan J. *A deterministic attribute grammar evaluator based on dynamic sequencing*, TOPLAS **1**:1, 1979, pp. 142–160.
243. Kennedy K. and Warren S. K. *Automatic generation of efficient evaluators for attribute grammars*, Third ACM Symposium on Principles of Programming Languages, 1976, pp. 32–49.
244. Kernighan B. W. *Ratfor — a preprocessor for a rational Fortran*, Software—Practice and Experience **5**:4, 1975, pp. 395–406.
245. Kernighan B. W. *PIC — a language for typesetting graphics*, Software—Practice and Experience **12**:1, 1982, pp. 1–21.
246. Kernighan B. W. and Cherry L. L. *A system for typesetting mathematics*, Comm. ACM **18**:3, 1975, pp. 151–157.
247. Kernighan B. W. and Pike R. *The UNIX Programming Environment*, Prentice-Hall, Englewood Cliffs, N. J., 1984. (Керниган Б. В., Пайк Р. *UNIX — универсальная среда программирования*. — М.:Финансы и статистика, 1992.)
248. Kernighan B. W. and Ritchie D. M. *The C Programming Language*, Prentice-Hall, Englewood Cliffs, N. J., 1978. (Керниган Б.В., Ритчи Д.М. *Язык программирования Си*. — М.:Финансы и статистика, 1992.)
249. Kildall G. *A unified approach to global program optimization*, ACM Symposium on Principles of Programming Languages, 1973, pp. 194–206.
250. Kleene S. C. *Representation of events in nerve nets*, in Shannon and McCarthy, 1956, pp. 3–40. (Клини С.К. *Представление событий в нервных сетях*//Сб. “Автоматы”. — М.: ИЛ, 1956. — С. 15–67.)
251. Knuth D. E. *A history of writing compilers*, Computers and Automation (December, 1962), pp. 8–18. Reprinted in [349], pp. 38–56.
252. Knuth D. E. *Backus Normal Form vs. Backus Naur Form*, Comm. ACM **7**:12, 1964, pp. 735–736.

253. Knuth D. E. *On the translation of languages from left to right*, Information and Control **8:6**, 1965, pp. 607–639. (Кнут Д. *О переводе (трансляции) языков слева направо*//Сб. “Языки и автоматы” — М.: Мир, 1975. — С. 9–42.)
254. Knuth D. E. *Semantics of context-free languages*, Mathematical Systems Theory **2:2**, 1968, pp. 127–145; Errata **5:1**, 1971, pp. 95–96.
255. Knuth D. E. *Top-down syntax analysis*, Acta Informatica **1:2**, 1971, pp. 79–110. (Кнут Д. *Нисходящий синтаксический анализ*//Кибернетический сборник. — М.:Мир, 1978. — Вып.15. — С. 101–142.)
256. Knuth D. E. *An empirical study of FORTRAN programs*, Software—Practice and Experience **1:2**, 1971, pp. 105–133.
257. Knuth D. E. *The Art of Computer Programming*: Vol. 1, 2nd. Ed., *Fundamental Algorithms*, Addison-Wesley, Reading, Mass, 1973. (Кнут Д. *Искусство программирования. Т.1. Основные алгоритмы*. — 3-е изд. — М.: Издательский дом “Вильямс”, 2000.)
258. Knuth D. E. *The Art of Computer Programming*: Vol. 3, *Sorting and Searching*, Addison-Wesley, Reading, Mass, 1973. (Кнут Д. *Искусство программирования. Т.3. Сортировка и поиск*. — 2-е изд. — М.: Издательский дом “Вильямс”, 2000.)
259. Knuth D. E. *A generalization of Dijkstra’s algorithm*, Information Processing Letters **6**, 1977, pp. 1–5.
260. Knuth D. E. *The T_EXbook*, Addison-Wesley, Reading, Mass, 1984. (Кнут Д. *Все про T_EX*. — Протвино: АО RDTEX, 1993.)
261. Knuth D. E. *Literate programming*, Computer J. **28:2**, 1984, pp. 97–111.
262. Knuth D. E. *Computers and Typesetting*, vol. 1: *T_EX*, Addison-Wesley, Reading, Mass, 1985, 1986. A preliminary version has been published under the title, *T_EX: The Program*.
263. Knuth D. E., Morris J. H. and Pratt V. R. *Fast pattern matching in strings*, SIAM J. Computing **6:2**, 1977, pp. 323–350.
264. Knuth D. E. and Trabb Pardo L. *Early development of programming languages*, Encyclopedia of Computer Science and Technology **7**, Marcel Dekker, New York, 1977, pp. 419–493.
265. Korenjak A. J. *A practical method for constructing LR(k) processors*, Comm. ACM **12:11**, 1969, pp. 613–623.
266. Kosaraju S. R. *Analysis of structured programs*, J. Computer and System Sciences **9:3**, 1974, pp. 232–255.
267. Koskimies K. and Räihä K.-J. *Modelling of space-efficient one-pass translation using attribute grammars*, Software—Practice and Experience **13**, 1983, pp. 119–129.
268. Koster C. H. A. *Affix grammars*, [344], pp. 95–109.
269. Kou L. *On live-dead analysis for global data flow problems*, J. ACM **24:3**, 1977, pp. 473–483.
270. Kristensen B. B. and Madsen O. L. *Methods for computing LALR(k) lookahead*, TOPLAS **3:1**, 1981, pp. 60–82.
271. Kron H. *Tree Templates and Subtree Transformational Grammars*, Ph. D. Thesis, Univ. of California, Santa Cruz, 1975.

272. LaLonde W. R. *An efficient LALR parser generator*, Tech. Rep. 2, Computer Systems Research Group, Univ. of Toronto, 1971.
273. LaLonde W. R. *On directly constructing LR(k) parsers without chain reductions*, Third ACM Symposium on Principles of Programming Languages, 1976, pp. 127–133.
274. LaLonde W. R., Lee E. S. and Horning J. J. *An LALR(k) parser generator*, Proc. IFIP Congress 71 TA-3, North-Holland, Amsterdam, 1971, pp. 153–157.
275. Lamb D. A. *Construction of a peephole optimizer*, Software—Practice and Experience 11, 1981, pp. 638–647.
276. Lampson B. W. *Fast procedure calls*, ACM SIGPLAN Notices 17:4 (April, 1982), pp. 66–76.
277. Landin P. J. *The mechanical evaluation of expressions*, Computer J. 6:4, 1964, pp. 308–320.
278. Lecarme O. and Peyrolle-Thomas M.-C. *Self-compiling compilers: an appraisal of their implementation and portability*, Software—Practice and Experience 8, 1978, pp. 149–170.
279. Ledgard H. F. *Ten mini-languages: a study of topical issues in programming languages*, Computing Surveys 3:3, 1971, pp. 115–146.
280. Leinius R. P. *Error Detection and Recovery for Syntax Directed Compiler Systems*, Ph. D. Thesis, University of Wisconsin, Madison, 1970.
281. Lengauer T. and Tarjan R. E. *A fast algorithm for finding dominators in a flowgraph*, TOPLAS 1, 1979, pp. 121–141.
282. Lesk M. E. *Lex — a lexical analyzer generator*, Computing Science Technical Report 39, AT&T Bell Laboratories, Murray Hill, N. J., 1975.
283. Leverett B. W. *Topics in code generation and register allocation*, CMU CS-82-130, Computer Science Dept., Carnegie-Mellon Univ., Pittsburgh, Pennsylvania, 1982.
284. Leverett B. W., Cattell R. G. G., Hobbs S. O., Newcomer J. M., Reiner A. H., Schatz B. R. and Wulf W. A. *An overview of the production-quality compiler-compiler project*, Computer 13:8, 1980, pp. 38–40.
285. Leverett, B. W. and Szymanski T. G. *Chaining span-dependent jump instructions*, TOPLAS 2:3, 1980, pp. 274–289.
286. Levy J. P. *Automatic correction of syntax errors in programming languages*, Acta Informatica 4, 1975, pp. 271–292.
287. Lewis P. M., II, Rosenkrantz D. J. and Stearns R. E. *Attributed translations*, J. Computer and System Sciences 9:3, 1974, pp. 279–307.
288. Lewis P. M., II, Rosenkrantz D. J. and Stearns R. E. *Compiler Design Theory*, Addison-Wesley, Reading, Mass, 1976. (Льюис Ф., Розенкранц Д., Стирнз Р. *Теоретические основы проектирования компиляторов*. — М.:Мир, 1979.)
289. Lewis P. M., II and Stearns R. E. *Syntax-directed transduction*, J. ACM 15:3, 1968, pp. 465–488.
290. Lorho B. *Semantic attribute processing in the system Delta*, [126], pp. 21–40.
291. Lorho B. *Methods and Tools for Compiler Construction*, Cambridge Univ. Press, 1984.
292. Lorho B. and Pair C. *Algorithms for checking consistency of attribute grammars*, [197], pp. 29–54.

293. Low J. and Rovner P. *Techniques for the automatic selection of data structures*, Third ACM Symposium on Principles of Programming Languages, 1976, pp. 58–67.
294. Lowry E. S. and Medlock C. W. *Object code optimization*, Comm. ACM **12**, 1969, pp. 13–22.
295. Lucas P. *The structure of formula translators*, Elektronische Rechenanlagen **3**, 1961, pp. 159–166.
296. Lunde A. *Empirical evaluation of some features of instruction set processor architectures*, Comm. ACM **20:3**, 1977, pp. 143–153.
297. Lunell H. *Code Generator Writing Systems*, Ph. D. Thesis, Linköping University, Linköping, Sweden, 1983.
298. MacQueen D. B., Plotkin G. P. and Sethi R. *An ideal model of recursive polymorphic types*, Eleventh Annual ACM Symposium on Principles of Programming Languages, 1984, pp. 165–174.
299. Madsen O. L. *On defining semantics by means of extended attribute grammars*, [224], pp. 259–299.
300. Marill T. *Computational chains and the simplification of computer programs*, IRE Trans. Electronic Computers **EC-11:2**, 1962, pp. 173–180.
301. Martelli A. and Montanari U. *An efficient unification algorithm*, TOPLAS **4:2**, 1982, pp. 258–282.
302. Mauney J. and Fischer C. N. *A forward move algorithm for LL and LR parsers*, ACM SIGPLAN Notices **17:4**, 1982, pp. 79–87.
303. Mayoh B. H. *Attribute grammars and mathematical semantics*, SIAM J. Computing **10:3**, 1981, pp. 503–518.
304. McCarthy J. *Towards a mathematical science of computation*, Information Processing 1962, North-Holland, Amsterdam, 1963, pp. 21–28.
305. McCarthy J. *History of Lisp*, [454], pp. 173–185.
306. McClure R. M. *TMG — a syntax-directed compiler*, Proc. 20th ACM National Conf., 1965, pp. 262–274.
307. McCracken N. J. *An Investigation of a Programming Language with a Polymorphic Type Structure*, Ph. D. Thesis, Syracuse University, Syracuse, N. Y., 1979.
308. McCullough W. S. and Pitts W. *A logical calculus of the ideas immanent in nervous activity*, Bulletin of Math. Biophysics **5**, 1943, pp. 115–133. (Логическое исчисление идей, относящихся к нервной активности//Сб. “Автоматы”. — М.:ИЛ, 1956. — С. 362–384.)
309. McKeeman W. M. *Peephole optimization*, Comm. ACM **8:7**, 1965, pp. 443–444.
310. McKeeman W. M. *Symbol table access*, [50], pp. 253–301.
311. McKeeman W. M., Horning J. J. and Wortman D. B. *A Compiler Generator*, Prentice-Hall, Englewood Cliffs, N. J., 1970.
312. McNaughton R. and Yamada H. *Regular expressions and state graphs for automata*, IRE Trans. on Electronic Computers **EC-9:1**, 1960, pp. 38–47.
313. Meertens L. *Incremental polymorphic type checking in B*, Tenth ACM Symposium on Principles of Programming Languages, 1983, pp. 265–275.

314. Metcalf M. *Fortran Optimization*, Academic Press, New York, 1982.
315. Miller R. E. and Thatcher J. W. (eds.). *Complexity of Computer Computations*, Academic Press, New York, 1972.
316. Milner R. *A theory of type polymorphism in programming*, J. Computer and System Sciences 17:3, 1978, pp. 348–375.
317. Milner R. *A proposal for standard ML*, ACM Symposium on Lisp and Functional Programming, 1984, pp. 184–197.
318. Minker J. and Minker R. G. *Optimization of boolean expressions — historical developments*, A. of the History of Computing 2:3, 1980, pp. 227–238.
319. Mitchell J. C. *Coercion and type inference*, Eleventh ACM Symposium on Principles of Programming Languages, 1984, pp. 175–185.
320. Moore E. F. *Gedanken experiments in sequential machines*, in Shannon and McCarthy, 1956, pp. 129–153. (Мур Э. Умозрительные эксперименты с последовательностными машинами//Сб. “Автоматы”. — М.:ИЛ, 1956. — С. 179–210.)
321. Morel E. and Renvoise C. *Global optimization by suppression of partial redundancies*, Comm. ACM 22, 1979, pp. 96–103.
322. Morris J. H. *Lambda-Calculus Models of Programming Languages*, Ph. D. Thesis, MIT, Cambridge, Mass, 1968.
323. Morris R. *Scatter storage techniques*, Comm. ACM 11:1, 1968, pp. 38–43.
324. Moses J. *The function of FUNCTION in Lisp*, SIGSAM Bulletin 15 (July, 1970), pp. 13–27.
325. Moulton P. G. and Muller M. E. *DITRAN — a compiler emphasizing diagnostics*, Comm. ACM 10:1, 1967, pp. 52–54.
326. Muchnick S. S. and Jones N. D. *Program Flow Analysis: Theory and Applications*, Prentice-Hall, Englewood Cliffs, N. J., 1981.
327. Nakata I. *On compiling algorithms for arithmetic expressions*, Comm. ACM 10:8, 1967, pp. 492–494.
328. Naur P. (ed.). *Revised report on the algorithmic language Algol 60*, Comm. ACM 6:1, 1963, pp. 1–17. (Алгоритмический язык АЛГОЛ 60. — М.:Мир, 1960.)
329. Naur P. *Checking of operand types in Algol compilers*, BIT 5, 1965, 151–163.
330. Naur P. *The European side of the last phase of the development of Algol 60*, in [454], pp. 92–139, pp. 147–161.
331. Newey M. C., Poole P. C. and Waite W. M. *Abstract machine modelling to produce portable software — a review and evaluation*, Software—Practice and Experience 2:2, 1972, pp. 107–136.
332. Newey M. C. and Waite W. M. *The robust implementation of sequence-controlled iteration*, Software—Practice and Experience 15:7, 1985, pp. 655–668.
333. Nicholls J. E. *The Structure and Design of Programming Languages*, Addison-Wesley, Reading, Mass, 1975.
334. Nievergelt J. *On the automatic simplification of computer code*, Comm. ACM 8:6, 1965, pp. 366–370.
335. Nori K. V., Ammann U., Jensen K., Nageli H. H. and Jacobi Ch. *Pascal P implementation notes*, in Barron, 1981, pp. 125–170.

336. Osterweil L. J. *Using data flow tools in software engineering*, in Muchnick and Jones, 1981, pp. 237–263.
337. Pager D. *A practical general method for constructing LR(k) parsers*, Acta Informatica 7, 1977, pp. 249–268.
338. Pager D. *Eliminating unit productions from LR(k) parsers*, Acta Informatica 9, 1977, pp. 31–59.
339. Pai A. B. and Kieburtz R. B. *Global context recovery: a new strategy for syntactic error recovery by table-driven parsers*, TOPLAS 2:1, 1980, pp. 18–41.
340. Paige R. and Schwartz J. T. *Expression continuity and the formal differentiation of algorithms*, Fourth ACM Symposium on Principles of Programming Languages, 1977, pp. 58–71.
341. Palm R. C., Jr. *A portable optimizer for the language C*, M. Sc. Thesis, MIT, Cambridge, Mass, 1975.
342. Park J. C. H., Choe K. M. and Chang C. H. *A new analysis of LALR formalisms*, TOPLAS 7:1, 1985, pp. 159–175.
343. Paterson M. S. and Wegman M. *Linear unification*, J. Computer and System Sciences 16:2, 1978, pp. 158–167.
344. Peck J. E. L. *Algol 68 Implementation*, North-Holland, Amsterdam, 1971.
345. Pennello T. and DeRemer F. *A forward move algorithm for LR error recovery*, Fifth Annual ACM Symposium on Principles of Programming Languages, 1978, pp. 241–254.
346. Pennello T., DeRemer F. and Meyers R. *A simplified operator identification scheme for Ada*, ACM SIGPLAN Notices 15:7 (July–August, 1980), pp. 82–87.
347. Persch G., Winterstein G., Daussmann M. and Drossopoulou S. *Overloading in preliminary Ada*, ACM SIGPLAN Notices 15:11 (November, 1980), pp. 47–56.
348. Peterson W. W. *Addressing for random access storage*, IBM J. Research and Development 1:2, 1957, pp. 130–146.
349. Pollack B. W. *Compiler Techniques*, Auerbach Publishers, Princeton, N. J., 1972.
350. Pollock L. L. and Soffa M. L. *Incremental compilation of locally optimized code*, Twelfth Annual ACM Symposium on Principles of Programming Languages, 1985, pp. 152–164.
351. Powell M. L. *A portable optimizing compiler for Modula-2*, ACM SIGPLAN Notices 19:6, 1984, pp. 310–318.
352. Pratt T. W. *Programming Languages: Design and Implementation*, 2nd Ed., Prentice-Hall, Englewood Cliffs, N. J., 1984. (Перевод первого издания: Пратт Т. Языки программирования: разработка и реализация. — М.:Мир, 1979.)
353. Pratt V. R. *Top down operator precedence*, ACM Symposium on Principles of Programming Languages, 1973, pp. 41–51.
354. Price C. E. *Table lookup techniques*, Computing Surveys 3:2, 1971, pp. 49–65.
355. Prosser R. T. *Applications of boolean matrices to the analysis of flow diagrams*, AFIPS Eastern Joint Computer Conf., Spartan Books, Baltimore, Md., 1959, pp. 133–138.
356. Purdom P. and Brown C. A. *Semantic routines and LR(k) parsers*, Acta Informatica 14:4, 1980, pp. 299–315.
357. Purdom P. W. and Moore E. F. *Immediate predominators in a directed graph*, Comm. ACM 15:8, 1972, pp. 777–778.

358. Rabin M. O. and Scott D. *Finite automata and their decision problems*, IBM J. Research and Development 3:2, 1959, pp. 114–125. (Рабин М., Скотт Д. Конечные автоматы и алгоритмы их разрешения.//Кибернетический сборник. — М.:ИЛ, 1962. — Вып.4. — С. 56–91.)
359. Radin G. and Rogoway H. P. *NPL: Highlights of a new programming language*, Comm. ACM 8:1, 1965, pp. 9–17.
360. Räihä K.-J. *A Space Management Technique for Multi-Pass Attribute Evaluators*, Ph. D. Thesis, Report A-1981-4, Dept. of Computer Science, University of Helsinki, 1981.
361. Räihä K.-J. and Saarinen M. *Testing attribute grammars for circularity*, Acta Informatica 17, 1982, pp. 185–192.
362. Räihä K.-J., Saarinen M., Sarjakoski M., Sippu S., Soisalon-Soininen E. and Tienari M. *Revised report on the compiler writing system HLP78*, Report A-1983-1, Dept. of Computer Science, University of Helsinki, 1983.
363. Randell B. and Russell L. J. *Algol 60 Implementation*, Academic Press, New York, 1964. (Ренделл Б., Рассел Л. Реализация АЛГОЛА 60. — М.: Мир, 1967.)
364. Redziejowski R. R. *On arithmetic expressions and trees*, Comm. ACM 12:2, 1969, pp. 81–84.
365. Reif J. H. and Lewis H. R. *Symbolic evaluation and the global value graph*, Fourth ACM Symposium on Principles of Programming Languages, 1977, pp. 104–118.
366. Reiss S. P. *Generation of compiler symbol processing mechanisms from specifications*, TOPLAS 5:2, 1983, pp. 127–163.
367. Reps T. W. *Generating Language-Based Environments*, MIT Press, Cambridge, Mass, 1984.
368. Reynolds J. C. *Three approaches to type structure*, Mathematical Foundations of Software Development, Lecture Notes in Computer Science 185, Springer-Verlag, Berlin, 1985, pp. 97–138.
369. Richards M. *The portability of the BCPL compiler*, Software—Practice and Experience 1:2, 1971, pp. 135–146.
370. Richards M. *The implementation of the BCPL compiler*, in P. J. Brown (ed.), Software Portability: An Advanced Course, Cambridge University Press, 1977.
371. Ripken K. *Formale beschreibung von maschinen, implementierungen und optimierender maschinen-codeerzeugung aus attributierten programmgraphen*, TUM-INFO-7731, Institut für Informatik, Universität München, Munich, 1977.
372. Ripley G. D. and Druseikis F. C. *A statistical analysis of syntax errors*, Computer Languages 3, 1978, pp. 227–240.
373. Ritchie D. M. *A tour through the UNIX C compiler*, AT&T Bell Laboratories, Murray Hill, N. J., 1979.
374. Ritchie D. M. and Thompson K. *The UNIX time-sharing system*, Comm. ACM 17:7, 1974, pp. 365–375.
375. Robertson E. L. *Code generation and storage allocation for machines with span-dependent instructions*, TOPLAS 1:1, 1979, pp. 71–83.
376. Robinson J. A. *A machine-oriented logic based on the resolution principle*, J. ACM 12:1, 1965, pp. 23–41.

377. Rohl J. S. *An Introduction to Compiler Writing*, American Elsevier, New York, 1975.
378. Röhrich J. *Methods for the automatic construction of error correcting parsers*, Acta Informatica 13:2, 1980, pp. 115–139.
379. Rosen B. K. *High-level data flow analysis*, Comm. ACM 20, 1977, pp. 712–724.
380. Rosen B. K. *Monoids for rapid data flow analysis*, SIAM J. Computing 9:1, 1980, pp. 159–196.
381. Rosen S. *Programming Systems and Languages*, McGraw-Hill, New York, 1967.
382. Rosenkrantz D. J. and Stearns R. E. *Properties of deterministic top-down grammars*, Information and Control 17:3, 1970, pp. 226–256.
383. Rosler L. *The evolution of C — past and future*, AT&T Bell Labs Technical Journal 63:8, 1984, pp. 1685–1699.
384. Rustin R. *Design and Optimization of Compilers*, Prentice-Hall, Englewood Cliffs, N.J., 1972.
385. Ryder B. G. *Constructing the call graph of a program*, IEEE Trans. Software Engineering SE-5:3, 1979, pp. 216–226.
386. Ryder B. G. *Incremental data flow analysis*, Tenth ACM Symposium on Principles of Programming Languages, 1983, pp. 167–176.
387. Saarinen M. *On constructing efficient evaluators for attribute grammars*, Automata, Languages and Programming, Fifth Colloquium, Lecture Notes in Computer Science 62, Springer-Verlag, Berlin, 1978, pp. 382–397.
388. Samelson K. and Bauer F. L. *Sequential formula translation*, Comm. ACM 3:2, 1960, pp. 76–83.
389. Sankoff D. and Kruskal J. B. (eds.). *Time Warps, String Edits, and Macromolecules: The Theory and Practice of Sequence Comparison*, Addison-Wesley, Reading, Mass, 1983.
390. Scarborough R. G. and Kolsky H. G. *Improved optimization of Fortran object programs*, IBM J. Research and Development 24:6, 1980, pp. 660–676.
391. Schaefer M. *A Mathematical Theory of Global Program Optimization*, Prentice Hall, Englewood Cliffs, N.J., 1973.
392. Schonberg E., Schwartz J. T. and Sharir M. *An automatic technique for selection of data representations in SETL Programs*, TOPLAS 3:2, 1981, pp. 126–143.
393. Schorre D. V. *Meta-II: a syntax-oriented compiler writing language*, Proc. 19th ACM National Conf., 1964, D1.3-1 – D1.3-11.
394. Schwartz J. T. *On Programming: An Interim Report on the SETL Project*, Courant Inst., New York, 1973.
395. Schwartz J. T. *Automatic data structure choice in a language of very high level*, Comm. ACM 18:12, 1975, pp. 722–728.
396. Schwartz J. T. *Optimization of very high level languages*, Computer Languages. Part I: Value transmission and its corollaries, 1:2, 1975, pp. 161–194; part II: Deducing relationships of inclusion and membership, 1:3, 1975, pp. 197–218.
397. Sedgewick R. *Implementing Quicksort programs*, Comm. ACM 21, 1978, pp. 847–857.
398. Sethi R. *Complete register allocation problems*, SIAM J. Computing 4:3, 1975, pp. 226–248.

399. Sethi R. and Ullman J. D. *The generation of optimal code for arithmetic expressions*, J. ACM 17:4, 1970, pp. 715–728.
400. Shannon C. and McCarthy J. *Automata Studies*, Princeton University Press, 1956.
401. Sheridan P. B. *The arithmetic translator-compiler of the IBM Fortran automatic coding system*, Comm. ACM 2:2, 1959, pp. 9–21.
402. Shimasaki M., Fukaya S., Ikeda K. and Kiyono T. *An analysis of Pascal programs in compiler writing*, Software—Practice and Experience 10:2, 1980, pp. 149–157.
403. Shustek L. J. *Analysis and performance of computer instruction sets*, SLAC Report 205, Stanford Linear Accelerator Center, Stanford University, Stanford, California, 1978.
404. Sippu S. *Syntax error handling in compilers*, Rep. A-1981-1, Dept. of Computer Science, Univ. of Helsinki, Helsinki, Finland, 1981.
405. Sippu S. and Soisalon-Soininen E. *A syntax-error-handling technique and its experimental analysis*, TOPLAS 5:4, 1983, pp. 656–679.
406. Soisalon-Soininen E. *On the space optimizing effect of eliminating single productions from LR parsers*, Acta Informatica 14, 1980, pp. 157–174.
407. Soisalon-Soininen E. and Ukkonen E. *A method for transforming grammars into LL(k) form*, Acta Informatica 12, 1979, pp. 339–369.
408. Spillman T. C. *Exposing side effects in a PL/I optimizing compiler*, Information Processing 71, North-Holland, Amsterdam, 1971, pp. 376–381.
409. Stearns R. E. *Deterministic top-down parsing*, Proc. 5th Annual Princeton Conf. on Information Sciences and Systems, 1971, pp. 182–188.
410. Steel T. B., Jr. *A first version of Uncol*, Western Joint Computer Conference, 1961, pp. 371–378.
411. Steele G. L., Jr. *Common LISP*, Digital Press, Burlington, Mass, 1984.
412. Stockhausen P. F. *Adapting optimal code generation for arithmetic expressions to the instruction sets available on present-day computers*, Comm. ACM 16:6, 1973, pp. 353–354. Errata: 17:10, 1974, p. 591.
413. Stonebraker M., Wong E., Kreps P. and Held G. *The design and implementation of INGRES*, ACM Trans. Database Systems 1:3, 1976, pp. 189–222.
414. Strong J., Wegstein J., Tritter A., Olsztyń J., Mock O. and Steel T. *The problem of programming communication with changing machines: a proposed solution*, Comm. ACM 1:8 (August, 1958), pp. 12–18; Part 2: 1:9 (September, 1958), pp. 9–15; Report of the SHARE Ad-Hoc committee on Universal Languages.
415. Stroustrup B. *The C++ Programming Language*, Addison-Wesley, Reading, Mass, 1986. (Страуструп Б. Язык программирования C++. — 3-е изд. — СПб.; М.: “Невский Диалект” — “Издательство БИНОМ”, 1999.)
416. Suzuki N. *Inferring types in Smalltalk*, Eighth ACM Symposium on Principles of Programming Languages, 1981, pp. 187–199.
417. Suzuki N. and Ishihata K. *Implementation of array bound checker*, Fourth ACM Symposium on Principles of Programming Languages, 1977, pp. 132–143.
418. Szymanski T. G. *Assembling code for machines with span-dependent instructions*, Comm. ACM 21:4, 1978, pp. 300–308.

419. Tai K. C. *Syntactic error correction in programming languages*, IEEE Trans. Software Engineering **SE-4:5**, 1978, pp. 414–425.
420. Tanenbaum A. S., van Staveren H., Keizer E. G. and Stevenson J. W. *A practical tool kit for making portable compilers*, Comm. ACM **26:9**, 1983, pp. 654–660.
421. Tanenbaum A. S., van Staveren H. and Stevenson J. W. *Using peephole optimization on intermediate code*, TOPLAS **4:1**, 1982, pp. 21–36.
422. Tantzen R. G. *Algorithm 199: Conversions between calendar date and Julian day number*, Comm. ACM **6:8**, 1963, p. 443.
423. Tarhio J. *Attribute evaluation during LR parsing*, Report A-1982-4, Dept. of Computer Science, University of Helsinki, 1982.
424. Tarjan R. E. *Finding dominators in directed graphs*, SIAM J. Computing **3:1**, 1974, pp. 62–89.
425. Tarjan R. E. *Testing flow graph reducibility*, J. Computer and System Sciences **9:3**, 1974, pp. 355–365.
426. Tarjan R. E. *Efficiency of a good but not linear set union algorithm*, J. ACM **22:2**, 1975, pp. 215–225.
427. Tarjan R. E. *A unified approach to path problems*, J. ACM **28:3**, pp. 577–593. And *Fast algorithms for solving path problems*, J. ACM **28:3**, 1981, pp. 594–614.
428. Tarjan R. E. and Yao A. C. *Storing a sparse table*, Comm. ACM **22:11**, 1979, pp. 606–611.
429. Tennenbaum A. M. *Type determination in very high level languages*, NSO-3, Courant Institute of Math. Sciences, New York Univ, 1974.
430. Tennent R. D. *Principles of Programming Languages*, Prentice-Hall International, Englewood Cliffs, N. J., 1981.
431. Thompson K. *Regular expression search algorithm*, Comm. ACM **11:6**, 1968, pp. 419–422.
432. Tjiang S. W. K. *Twig language manual*, Computing Science Technical Report 120, AT&T Bell Laboratories, Murray Hill, N. J., 1986.
433. Tokuda T. *Eliminating unit reductions from LR(k) parsers using minimum contexts*, Acta Informatica **15**, 1981, pp. 447–470.
434. Trickey H. W. *Compiling Pascal Programs into Silicon*, Ph. D. Thesis, Stanford Univ, 1985.
435. Ullman J. D. *Fast algorithms for the elimination of common subexpressions*, Acta Informatica **2**, 1973, pp. 191–213.
436. Ullman J. D. *Principles of Database Systems*, 2nd Ed., Computer Science Press, Rockville, Md., 1982. (Ульман Д.Д. *Основы систем баз данных*. — М.:Финансы и статистика, 1993.)
437. Ullman J. D. *Computational Aspects of VLSI*, Computer Science Press, Rockville, Md., 1984.
438. Vyssotsky V. and Wegner P. *A graph theoretical Fortran source language analyzer*, manuscript, AT&T Bell Laboratories, Murray Hill, N. J., 1963.
439. Wagner R. A. *Order-n correction for regular languages*, Comm. ACM **16:5**, 1974, pp. 265–268.
440. Wagner R. A. and Fischer M. J. *The string-to-string correction problem*, J. ACM **21:1**, 1974, pp. 168–174.

441. Waite W. M. *Code generation*, [50], pp. 302–332.
442. Waite W. M. *Optimization*, [50], pp. 549–602.
443. Waite W. M. and Carter L. R. *The cost of a generated parser*, Software—Practice and Experience **15:3**, 1985, pp. 221–237.
444. Wasilew S. G. *A Compiler Writing System with Optimization Capabilities for Complex Order Structures*, Ph. D. Thesis, Northwestern Univ., Evanston, 1971, p. 111.
445. Watt D. A. *The parsing problem for affix grammars*, Acta Informatica **8**, 1977, pp. 1–20.
446. Wegbreit B. *The treatment of data types in ELI*, Comm. ACM **17:5**, 1974, pp. 251–264.
447. Wegbreit B. *Property extraction in well-founded property sets*, IEEE Trans. on Software Engineering **1:3**, 1975, pp. 270–285.
448. Wegman M. N. *Summarizing graphs by regular expressions*, Tenth Annual ACM Symposium on Principles of Programming Languages, 1983, pp. 203–216.
449. Wegman M. N. and Zadeck F. K. *Constant propagation with conditional branches*, Twelfth Annual ACM Symposium on Principles of Programming Languages, 1985, pp. 291–299.
450. Wegstein J. H. *Notes on Algol 60*, in Wexelblat, 1981, pp. 126–127.
451. Weihl W. E. *Interprocedural data flow analysis in the presence of pointers, procedure variables, and label variables*, Seventh Annual ACM Symposium on Principles of Programming Languages, 1980, pp. 83–94.
452. Weingart S. W. *An Efficient and Systematic Method of Code Generation*, Ph. D. Thesis, Yale University, New Haven, Connecticut, 1973.
453. Welsh J., Sneeringer W. J. and Hoare C. A. R. *Ambiguities and insecurities in Pascal*, Software—Practice and Experience **7:6**, 1977, pp. 685–696.
454. Wexelblat R. L. *History of Programming Languages*, Academic Press, New York, 1981.
455. Wirth N. *PL 360 — a programming language for the 360 computers*, J. ACM **15:1**, 1968, pp. 37–74.
456. Wirth N. *The design of a Pascal compiler*, Software—Practice and Experience **1:4**, 1971, pp. 309–333.
457. Wirth N. *Pascal-S: A subset and its implementation*, in Barron, 1981, pp. 199–259.
458. Wirth N. and Weber H. *Euler: a generalization of Algol and its formal definition: Part I*, Comm. ACM **9:1**, 1966, pp. 13–23.
459. Wood D. *The theory of left factored languages*, Computer J. **12:4**, 1969, pp. 349–356.
460. Wulf W. A., Johnsson R. K., Weinstock C. B., Hobbs S. O. and Geschke C. M. *The Design of an Optimizing Compiler*, American Elsevier, New York, 1975.
461. Yannakakis M. Private communication, 1985.
462. Younger D. H. *Recognition and parsing of context-free languages in time n^3* , Information and Control **10:2**, 1967, pp. 189–208. (Янгер Д. *Распознавание и анализ контекстно-свободных языков за время n^3* //Сб. “Проблемы математической логики”. — М.:Мир, 1970. — С.344–362.)
463. Zelkowitz M. V. and Bail W. G. *Optimization of structured programs*, Software—Practice and Experience **4:1**, 1974, pp. 51–57.

Предметный указатель

L

Lex, 97; 119; 139; 156; 157; 265
прогностический оператор, 123

LL(k)-грамматика, 226

LR(k)-грамматика, 225

LR-грамматика, 225

L-атрибутное определение, 295

l-значение, 80

R

r-значение, 80

S

SLR-анализ, 226

S-атрибутное определение, 281; 292

Y

Yacc, 221; 260; 292
восстановление после ошибки, 266

A

Абстрактная стековая машина, 79

Адресация массива, 459
схема трансляции, 461

Активация, 376

время жизни, 378
дерево, 379

Активные переменные, 594

Алфавит, 105

Альтернатива, 175

Альтернативное имя, 610

Анализ, 23

SLR, 226

иерархический, 26

лексический, 26; 71; 97–155

линейный, 26

методом рекурсивного

спуска, 61; 188

нисходящий, 58; 188

потока данных, 641
предиктивный, 61; 185
семантический, 26; 28
синтаксический, 26; 48; 167

Анализатор

предиктивный, 63

Ассемблерный код, 37

Ассоциативность, 49; 108; 251

Атрибут, 51; 100

время жизни, 315; 319

наследуемый, 280; 282

рекурсивное вычисление, 325

синтезируемый, 52; 280; 281; 292

Атрибутная грамматика, 280

Б

Базовый блок, 500

алгебраические

преобразования, 503

нормального вида, 503

преобразование, 502

Буферизация ввода, 102

ограничители, 104

Бэкуса–Наура, форма, 44; 96; 167

В

Висячие ссылки, 394

Внешняя ссылка, 39

Внутренний цикл, 571

Восстановление после ошибки, 169; 199; 216; 257; 266; 340

в режиме паники, 172; 199; 257

глобальная коррекция, 172

на уровне фразы, 201

продукции ошибок, 172

уровень фразы, 172

Временная переменная, 458

Время жизни, 378

Выведение типа, 356

Вывод, 175

Выравнивание, 385

Выражение типа, 338

Вычисление, инвариантное относительно цикла, 563

Г

Генератор компиляторов, 41

Генерация кода, 35; 487

выбор инструкций, 489

выбор порядка вычислений, 491

распределение регистров, 490; 512

управление памятью, 489

целевая машина, 492

Глубина вложенности, 401

Грамматика

LL(1), 198

LL(k), 226

LR, 225

LR(k), 207; 225

атрибутная, 280

контекстно-свободная, 27; 221; 280

леворекурсивная, 183

неоднозначная, 48; 179; 250; 263

операторная, 209

расширенная, 226

Грамматический символ, 174

Граф

взаимодействия регистров, 516

зависимости, 280; 283

переходов, 125

раскраска, 516

Граф потока, 500; 503

тор-решение, 648

глубина, 626

декомпозиция, 635

интервал, 626

область, 577

пределочный, 628

приводимый, 572; 628

разделение узлов, 629

упорядочение вглубь, 622

цикл, 505

Д

Даг, 289; 339; 444; 517; 520
Декартово произведение, 338
Дерево
активации, 379
доминаторов, 569
охватывающее вглубь, 624
синтаксическое, 286
Дерево разбора, 23; 47; 48; 58; 177
анnotated, 52; 280
скелетное, 212
Детерминированный конечный автомат, 127; 144
построение, 149
с минимальным числом состояний, 150
Диаграмма переходов, 112
Динамическое программирование, 537
Доминатор, 568
поиск, 631
Допустимая строка, 126
Достигающие определения, 576; 588; 639
локальные, 584
Доступное выражение, 591
Дублирование констант, 562
Дуга, 112

Е

Естественный цикл, 570

З

Загрузчик, 38
Замыкание
Клини, 106; 132; 157
позитивное, 106
Запись, 338; 455
активации, 384

И

Идемпотентность, 109
Идентификация операторов, 352
Идиома, 553
Инвариант цикла, 601
Индуктивная переменная, 605

Интерпретатор, 24
Использование имени, 505
Итерация, 106

К

Кадр, 384
Квантор общности, 358
Класс символов, 105; 110
дополнение, 157
Класс эквивалентности, 318
Ключевые слова, 73
зарезервированные, 73; 100

Коммутативность, 108
Компилятор, 22
Конечный автомат, 125
детерминированный, 127; 144
построение, 149
с минимальным числом состояний, 150
недетерминированный, 125; 180
моделирование, 136
построение, 132
сжатие таблиц, 153

Конкатенация, 106
Конкретный синтаксис, 66
Конструктор типа, 338
Контекстно-свободная грамматика, 45; 55; 173; 179; 221; 280
Конфликт перенос/свертка, 207
Конфликт свертка/свертка, 207
Кросс-компилятор, 681
Куча, 383

Л

Левая факторизация, 185
Левоассоциативность, 49
Лексема, 33; 71; 99; 414
Лексический анализ, 26; 71
Лексический анализатор, 44; 74; 97
Лидер, 501
Логические выражения, 465
сокращенные вычисления, 467
числовое представление, 466

Локальная оптимизация, 524

М

Макрос, 25; 35
Маркер, 305
Массив, 338; 459
Машинная идиома, 553
Мусор, 424
сборка, 424

Н

Наиболее общий унификатор, 365
Наследуемые атрибуты, 282
Недетерминированный конечный автомат, 125; 180
моделирование, 136
построение, 132
Неоднозначная грамматика, 179
Неоднозначность, 48; 250
удаление, 182
Нетерминал, 45; 173
Нормальная форма Грейбах, 273
Нормальная форма Хомского, 277

О

Область видимости, 381; 420; 453
динамическая, 396; 406
лексическая, 396
Область значений, 339
Область определения, 339
Обратная дуга, 570
Обратная поправка, 40; 476
Обход дерева, 55
вглубь, 55
Общие подвыражения, 559
Объединение, 106
Объявление, 452
Окончайшая рекурсия, 69
Оператор
перегруженный, 337
Операторная грамматика, 209
Определение переменной, 575
достигающее, 576
Оптимизация, 34

использование
алгебраических тождеств,
567
локальная, 524; 559
перемещение кода, 562; 602
распространение
копирований, 561; 598
удаление бесполезного
кода, 562
удаление общих
подвыражений, 559
устранение глобальных
общих подвыражений,
596
устранение индуктивной
переменной, 562; 564;
605
Отношения приоритетов, 210
Оценка типов, 653
обратная схема, 658
прямая схема, 657

П

Перегруженный оператор, 337
Перегрузка, 352
Передача параметров
значение-результат, 410
копирование-
восстановление, 410
по адресу, 409
по значению, 408
по имени, 411
по ссылке, 409
Переменная индукции, 564;
605
Перемещаемый код, 38; 602
Поддержка времени
исполнения, 376
Подпоследовательность, 106
Подстрока, 105
Полиморфные функции, 355
Порождение, 175
каноническое, 177
левое, 176
правое, 177
Порожденная строка, 48
Последовательное вычисление,
538
Последовательность возврата,
391

Последовательность вызова,
391; 481
Постфиксная запись, 51
Правоассоциативность, 49
Предиктивный анализ, 61; 189
Предиктивный анализатор, 63
Предложение, 105; 176
Предпросмотр, 234
Преобразование типов, 350;
463
неявное, 350
Препроцессор, 25; 35
Предфикс, 105
Приоритет, 251
Приоритет операторов, 50
Проверка
динамическая, 336
единственности, 336
имен, 336
статическая, 336
типа выражения, 342
типов, 28; 336
управления, 336
Прогностический оператор,
123; 143
Продукция, 45
Промежуточный код, 34
Профайлер, 416
Проход, 40
Процедура, 376; 452; 481
вызов, 377
определение, 376
рекурсивная, 378
фактические параметры, 377
формальные параметры, 377
Псевдоним, 610; 617
вычисление, 617

P

Разбор, 26; 48
Разделяемые узлы, 536
Раскрутка компилятора, 681
Распознаватель, 125
Распределение памяти
в куче, 395
статическое, 387
стековое, 389
Распределение регистров, 490
Расстояние редактирования,
164
Расширенная грамматика, 226

Регулярное выражение, 107;
180
сокращения, 110
эквивалентность, 108
Регулярное множество, 108
Регулярное определение, 109
Решеточная диаграмма, 645

С

Сборка мусора, 424
Свертка, 201
Связь доступа, 384; 401
Связь управления, 384
Семантика, 44
Семантические действия, 55
Сентенциальная форма, 176
Символьный отладчик, 661
Синтаксис, 44
Синтаксически управляемая
трансляция, 28; 42; 44; 51;
447
Синтаксически управляемое
определение, 51; 52; 55;
280; 288
простое, 56
строго нециклическое, 327
циклическое, 286
Синтаксический анализ, 26
LALR, 221
LR(k), 220
восходящий, 168; 201
нисходящий, 168
перенос/свертка, 201
предиктивный, 189
приоритета операторов, 209
Синтаксическое дерево, 28; 286
Синтез, 23
Синтезируемые атрибуты, 281;
292
Система типов, 339
надежная, 340
Сканирование, 26
Слово, 105
Состояние, 112
важное, 144
Стартовый символ, 45
Статические проверяющие
программы, 24
Стек, 80
вызовов, 380
управления, 380

Стоимость инструкции, 493
Строка, 105
 кратчайший повторяющийся префикс, 161
 основа, 202
 период, 161
 пустая, 46
 Фибоначчи, 161
Структурный редактор, 23
Суффикс, 105
Схема трансляции, 55; 296
Счетчик ссылок, 427

Т

Таблица переходов, 126
Таблица символов, 28; 30; 73; 76; 100; 336; 412; 456
Терминал, 45; 173
Тип
 базовый, 337
 восстановление после ошибки, 340
 выведение, 356
 выражение, 338
 запись, 338
 именная эквивалентность, 347
 имя, 338
 инструкции, 343
 конструктор, 338
 массив, 338
 структурная эквивалентность, 344; 347
 указатель, 338

фундаментальный, 337
функция, 339
 эквивалентность, 344
Токен, 26; 45; 71; 99; 105; 173
Топологическая сортировка, 285
Трехадресный код, 34; 443; 445
 косвенные тройки, 450
реализация, 449
 тройки, 450
четверки, 449

У

Указатель, 338
Унификация, 359
 наиболее общий унификатор, 360
Упаковка данных, 385
Уравнения потока данных, 574; 588
Устранение левой рекурсии, 184
Устранение неоднозначности, 182

Ф

Фазы компилятора, 30
Фактические параметры, 36; 377
Форма Бэкуса–Наура, 44; 96; 167

Формальные параметры, 36; 377
Функция, 339; 376
Функция отказа, 160
Функция приоритетов, 214

Х

Хеширование, 416

Ц

Цепочки использований определений, 596
Цепочки определений использований, 585
Цикл
 вложенный, 571
 внутренний, 571
 естественный, 570
 предзаголовок, 571

III

Шаблон, 99

Я

Язык
 исходный, 22
 контекстно-свободный, 176
 целевой, 22
Языковые расширения, 36

Научно-популярное издание

Альфред В. Ахо, Рави Сети, Джейфри Д. Ульман

**Компиляторы: принципы,
технологии и инструменты**

Литературный редактор *Е.Д. Давидян*

Верстка *М.А. Удалов*

Художественный редактор *С.А. Чернокозинский*

Технический редактор *Г.Н. Горобец*

Корректоры *Л.А. Гордиенко, О.В. Мишутина,
Л.В. Чернокозинская*

Издательский дом “Вильямс”.

101509, Москва, ул. Лесная, д. 43, стр. 1.

Изд. лиц. ЛР № 090230 от 23.06.99

Госкомитета РФ по печати.

Подписано в печать 23.04.2003. Формат 70×100/16.

Гарнитура Times. Печать офсетная.

Усл. печ. л. 61,92. Уч.-изд. л. 46,1.

Доп. тираж 3000 экз. Заказ № 2819.

Отпечатано с диапозитивов в ФГУП “Печатный двор”

Министерства РФ по делам печати,
телерадиовещания и средств массовых коммуникаций.
197110, Санкт-Петербург, Чкаловский пр., 15.

Компиляторы

Принципы, технологии, инструменты

Kаждый, кто всерьез занимался разработкой компиляторов, знаком с "Книгой дракона", *Principles of Compiler Design*, Альфреда Ахо и Джейфри Ульмана. Эта книга сыграла огромную роль в быстро развивающейся области разработки компиляторов. Со времени ее издания данная область науки значительно продвинулась вперед. Поэтому сейчас у вас в руках новый "дракон" — книга *Компиляторы: Принципы, технологии, инструменты*, написанная замечательной командой авторов — Альфредом Ахо, Рави Сети и Джейфри Ульманом.

Книга начинается с введения в принципы работы и построения компиляторов, продемонстрированные на примере создания простейшего однопроходного компилятора. Остальная часть книги раскрывает идеи, представленные в первых двух главах, и обсуждает более сложные вопросы синтаксического анализа, проверки типов, генерации и оптимизации кода.

В эту книгу вошли базовые материалы из предыдущего издания, обогащенные новейшими достижениями в данной области науки. Ее отличает следующее.

- Полнее раскрыто предназначение компиляторов
- Ориентация на практическую разработку компиляторов
- Большее внимание уделено синтаксически управляемой трансляции, проверке типов, генерации и оптимизации кода
- Широкий спектр примеров и упражнений

ОБ АВТОРАХ

Альфред Ахо — руководитель отдела AT&T Bell Laboratories в Мюррей Хилл, штат Нью-Джерси. Он получил степень бакалавра физики в университете Торонто и степень доктора в Принстонском университете.

Рави Сети — сотрудник AT&T Bell Laboratories в Мюррей Хилл, штат Нью-Джерси. В Индийском технологическом институте г. Капуре ему была присвоена степень бакалавра, а в Принстонском университете — степень доктора.

Джейфри Ульман — профессор компьютерных наук Станфордского университета. Степень бакалавра он получил в Колумбийском университете, а степень доктора — в Принстонском университете.

ISBN 5-8459-0189-8



01059

9 785845 901897



www.williamspublishing.com